174
13

PROLOG AND ARTIFICIAL INTELLIGENCE IN CHEMICAL ENGINEERING

by

Thomas E. Quantrille

Dissertation submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy
in
Chemical Engineering

APPROVED:

Y. A. Liu
Dr. Y.A. Liu, Chairman

William L. Conger
Dr. William L. Conger

Peter R. Rony
Dr. Peter R. Rony

John W. Roach
Dr. John W. Roach

Dr. Henry A. McGee, Jr.

May, 1991
Blacksburg, Virginia

PROLOG AND ARTIFICIAL INTELLIGENCE IN CHEMICAL ENGINEERING

by

Thomas E. Quantrille

Dr. Y.A. Liu, Chairman

Chemical Engineering

(ABSTRACT)

This dissertation deals with applications of Prolog and Artificial Intelligence (AI) to chemical engineering, and in particular, to the area of chemical process synthesis. We introduce the language Prolog (chapters 1-9), discuss AI techniques (chapters 10-11), discuss EXSEP, the EXpert System for SEParation Synthesis (chapters 12-15), and summarize applications of both AI and Artificial Neural Networks (ANNs) to chemical engineering (chapters 16-17).

We have developed EXSEP, a knowledge-based system that performs separation process synthesis. EXSEP is a computer-aided design tool that can generate flowsheets using any combination of high-recovery (sharp) and low-recovery (nonsharp) separations, using a variety of separation methods with energy and mass separating agents.

EXSEP generates separation process flowsheets using a unique plan-generate-test approach that incorporates computer-aided tools and techniques for problem representation and simplification, feasibility analysis of separation tasks, and heuristic synthesis and evolutionary improvement.

A difficult problem in knowledge-based approaches to chemical engineering is the "quantitative or deep knowledge dilemma." Experience has shown that a strictly qualitative knowledge approach to chemical process synthesis is insufficient. However, including rigorous quantitative analysis into an expert system is cumbersome and impractical.

EXSEP overcomes this deep-knowledge dilemma through a unique knowledge representation and problem-solving strategy that includes shortcut design calculations. These calculations are used as a feasibility test for all separations; no separation is chosen by EXSEP unless it is deemed as thermodynamically feasible through this quantitative, deep-knowledge, engineering analysis.

We apply EXSEP for the flowsheet synthesis of several industrial separations problems. The results show that EXSEP successfully generates technically feasible and economically attractive process flowsheets accurately and efficiently. EXSEP is also user-friendly, and can be readily applied by practicing engineers using a personal computer. In addition, EXSEP is developed modularly, and can be easily expanded in the future to include additional separation methods.

ACKNOWLEDGEMENTS

It is a pleasure to thank a number of very special persons who contributed to the preparation of this dissertation.

First, I would like to thank the enduring patience of my wife, Sharon Quantrille. Her support through the laborious process of research and dissertation preparation and editing was exemplary.

I would also like to thank the members of my advisory committee, in particular: Professor Y.A. Liu, who is my principal advisor and chairman of the committee; Professor Peter Rony, whose detailed review and comments proved invaluable; Professor John Roach, who introduced the subject of artificial intelligence to me; Professor William Conger, the Department Head who provided both technical and logistic support; and finally, Professor Henry McGee, who served on my committee even while on leave at the National Science Foundation as Director of the Division of Chemical and Thermal Systems.

In addition, I wish to express our gratitude to Professor Michael Mavrovouniotis of the University of Maryland for his prompt and thorough review of chapter 17.

To Mrs. Marie Hetherington, I owe a deep gratitude for her tireless and skillful copy-editing of the entire manuscript.

I would also like to express my appreciation to the Dow Chemical

Foundation, whose financial support made this work possible.

In addition, I would like to thank Mr. Karl Rony, Mr. David Schenk, and Mr. Jean Christian Brunet for help in performing Prolog programming. Mr. Karl Rony initiated this work and performed some of the initial Prolog programming. Mr. David Schenk and Mr. Jean Christian Brunet performed some of the Prolog programming on a specific module developed. All of their contributions are sincerely appreciated.

Lastly, I wish to thank Mrs. Diane Cannaday, who patiently and graciously helped all aspects of the preparation of this research.

TABLE OF CONTENTS

CHAPTER ONE

CHAPTER TWO

CHAPTER THREE

CHAPTER FOUR

CHAPTER FIVE

CHAPTER SIX

CHAPTER SEVEN

CHAPTER EIGHT

CHAPTER NINE

CHAPTER TEN

CHAPTER ELEVEN

CHAPTER TWELVE

CHAPTER THIRTEEN

CHAPTER FOURTEEN

CHAPTER FIFTEEN

CHAPTER SIXTEEN

CHAPTER SEVENTEEN

1

INTRODUCTION TO PROLOG

This chapter is an overview of Prolog. It introduces important concepts about the language through the use of examples. Section 1.1 describes how to represent facts in Prolog. Section 1.2 illustrates a Prolog program containing facts and questions. In section 1.3, we look at the use of rules in Prolog. Finally, we introduce recursive rules in section 1.4.

1.1 REPRESENTING FACTS IN PROLOG

Prolog is an ideal language for solving engineering problems. It is particularly useful for analyzing information about objects and their relationships. In solving engineering problems, we frequently deal with facts. Prolog represents facts through the use of *clauses*. For example, we can represent the fact that the normal boiling point of water is 100 °C by:

```
normal_boiling_point(water,100).
```

We use the same expression to describe the normal boiling point of any material:

```
normal_boiling_point(isobutane,-11.7).
normal_boiling_point(n_butane,-0.5).
normal_boiling_point(ethyl_acetate, 77).
normal_boiling_point(acrylonitrile,77).
```

Polyethylene has no boiling point. It decomposes before it vaporizes. We represent this fact as follows:

```
normal_boiling_point(polyethylene,decomposes).
```

Throughout this book, bold type, e.g., **normal_boiling_point(water,100)**, represents input *from the user to the computer.*

Each of the previous **normal_boiling_point** statements is a Prolog *clause*. In particular, it is a specific type of clause called a *fact*. Each clause ends with a "." , or period. The period represents a full stop. Prolog clauses must be written with lower-case letters, but can contain certain symbols such as _, **&**, **#**, and **+**.

In the **normal_boiling_point** statement, we have used the underscore character (_) to improve legibility. We can, of course, write the statement as:

```
normalboilingpoint(water,100).
```

but including the underscore between words makes the statement easier to read.

Each clause above declares a fact about the normal boiling point of a material. In English, we would read the clause

```
normal_boiling_point(isobutane,-11.7).
```

as "the normal boiling point of isobutane is minus 11.7 °C."

As in English, every clause in Prolog must contain a predicate. A predicate is the "action word" that declares a quality or an attribute about an object or a variety of objects. In Prolog, we call **normal_boiling_point** the *predicate*. Water, n_butane, 100, -11.7, -0.5, decomposes, etc. are all objects of the predicate. In Prolog, they are called *arguments*. A complete statement, e.g., **normal_boiling_point(water,100)**, contains the predicate and its arguments in parenthesis. It is a Prolog *clause*.

The general format for a Prolog fact is:

predicate(a_1,a_2,a_3, ..., a_n).

where a_1, a_2, a_3, etc. are all arguments. A predicate can have as many arguments as desired. Or, it can have no arguments at all. The following clauses, consisting of predicates with no arguments, are all acceptable Prolog facts:

```
proton.
electron.
neutron.
```

A predicate can relate many arguments. We know that if water is at a pressure of 1 atmosphere and a temperature of 20 °C, it will be a liquid. We can write a clause to relate the arguments water, pressure, temperature, and phase. We identify the clause with the predicate **phase**. The statement "water, at 1 atmosphere of pressure and 20 °C, is in the liquid phase" is written in Prolog as the clause:

```
phase(water,1,20,liquid).
```

We can generalize this clause to describe the phase of any material as follows:

```
phase(material, pressure, temperature, phase condition).
```

When we encounter the **phase(water,1,20,liquid)** statement in a program, it may not be clear what each argument represents. In addition, we may not know the units of the numbers 1 and 20. To remedy this situation and improve understanding of the program, we can put *comments* into the Prolog program. In the statement below, comments appear preceded by **%** . Prolog ignores all information between **%** and the end of the line.

```
phase(water,1,20,liquid)      % argument one = material
                              % argument two = pressure, atmospheres
                              % argument three = temperature, °C
                              % argument four = phase of the
                              % material at the given conditions.
```

These facts are essential to solving problems in Prolog. We can write facts about:

- Material properties

- Process data
- Cost information
- Problem specifications
- Problem constraints

Examples of different types of facts are shown below.

Material properties

```
flammable(acetone,yes).
flammable(water, no).

phase(oxygen,1,20,gas).
phase(acrylonitrile,1,20,liquid).
phase(acrylonitrile,1,80,gas).
phase(polyethylene,1,600,decomposes).
```

Process data

```
flow_rate(water,10000).
flow_rate(raffinate,789).
flow_rate(benzene,253).

alarm(low_level,on).
alarm(high_level,off).

pump(p2,vacuum_pump,on).
pump(p5,water_pump,on).
```

Cost Information

```
cost(raw_material,polyethylene,1.40).
cost(capital,compressor,800000).
cost(operating,natural_gas,0.85).
```

Problem specifications

```
capacity(pump,40,gpm).
capacity(reactor,900,ft3)
```

EXERCISES

1.1.1 Write a Prolog fact for each of the following statements.

a. The viscosity of carbon tetrachloride is 0.86 centipoise.

b. The specific gravity of water is 1.0.

c. The molecular weight of ammonia is 17.031.

1.1.2 Write a Prolog fact for each of the following statements. Include a comment statement using % to indicate units of each quantity.

a. Water is an inorganic compound with a molecular weight of 18.015, a freezing point of 273.2 K, a normal boiling point of 373.2 K, and a critical compressibility factor of 0.229.

b. Phosgene is an organic compound with a molecular weight of 98.916, a freezing point of 145 K, a boiling point of 280.8 K, and a critical compressibility factor of 0.280.

c. 1-Heptene is an olefin with a molecular weight of 98.189, a freezing point of 154.3 K, a boiling point of 677.9 K, and a critical compressibility factor of 0.262.

1.1.3 Translate the following Prolog facts into English sentences.

a. phase(oxygen,pressure,1,atmosphere,temperature,20,celsius,gas).

b. phase(oxygen,1,20,gas).

Comment on the advantages and disadvantages of way we represent each fact in Prolog.

1.2 A SIMPLE PROGRAM INCLUDING FACTS AND QUESTIONS

A. Prolog Programs

Programs in Prolog consist of three basic elements or actions:

- *Declaring facts*- about objects and their relationships,
- *Defining rules*- about objects and their relationships, and
- *Asking questions*- about these same objects and their relationships to solve complex problems.

When we ask a question, Prolog will dutifully attempt to answer it. Programming with the appropriate relationships and asking the right questions will enable Prolog to solve complex problems.

As an introduction to Prolog programming, let us look at a program that contains facts and questions only. We shall introduce rules in section 1.3. Consider again the facts about the normal boiling point of materials. We include all six facts below in a Prolog program:

```
normal_boiling_point(water,100).
normal_boiling_point(isobutane,-11.7).
normal_boiling_point(n_butane,-0.5).                    % dbla
normal_boiling_point(polyethylene,decomposes).
normal_boiling_point(ethyl_acetate, 77).
normal_boiling_point(acrylonitrile,77).
```

In Prolog, a collection of clauses is called a ***database***. Prolog uses the database to answer questions and solve problems. The above database, called database 1a, or simply *db1a*, will be used throughout this chapter. We also need to clarify some notation. As described in section 1.1, bold type, such as:

```
normal_boiling_point(water,100)
```

represents input *from the user to the computer*. Italic bold type, such as:

normal_boiling_point(water,100)

represents output *from the computer to the user*.

Now that we have our database set up, we can ask Prolog some questions. In Prolog, a question is just like a fact, except that we put a question mark followed by a hyphen (i.e., **?-**) before it. "Is the normal boiling point of water 100 °C?" in Prolog, is:

```
?- normal_boiling_point(water,100).
```

Note that all questions also must be followed by a period, which again represents a full stop. Because this fact exists in the preceding program or database, Prolog responds:

yes

If we ask:

```
?- normal_boiling_point(water,25).
```

Prolog responds

no

because the only fact on the normal boiling point of water in the database is 100 °C.

When we ask the question **normal_boiling_point(water,25)**, this question becomes the Prolog *goal*. A goal is essentially a question posed to the database. We try to find if the answer to this question is positive (i.e., "yes") or negative (i.e., "no") based on relationships in the database. Prolog solves problems by attempting to satisfy the goal.

Let us increase the complexity of our program by introducing another type of Prolog object, called a *variable*. If we designate capital letter **X** as our variable, we can now ask, "what material has a normal boiling point of 77 °C ?" In Prolog, we write:

```
?- normal_boiling_point(X,77).
```

Here, Prolog will not respond with a simple "yes" or "no". Instead, it replies with:

```
X = ethyl_acetate
```

If we want additional answers, we can get them easily. With most versions of Prolog, entering a semicolon ; calls the next Prolog answer to the screen:

```
X = ethyl_acetate ;
X = acrylonitrile
```

To ask Prolog again, we simply type in another semicolon. Prolog responds, and we see ***no*** on the screen since all facts that match the question have been exhausted:

```
X = ethyl_acetate ;
X = acrylonitrile ;
no
```

B. Data Objects

In a Prolog program, the arguments in relationships can be fixed concrete objects, or variables. In the above example, **X** is a variable. *All variables in Prolog begin with a capital letter.* The arguments **ethyl_acetate**, **acrylonitrile**, **77**, and **100** are fixed concrete objects, called *constants*. Constants are fixed objects that never change in a Prolog program.

The terms **77** and **100** belong to the first type of Prolog constant, called *numbers*. Numbers are, simply, *numerical*, fixed concrete objects. The terms **ethyl_acetate** and **acrylonitrile** are another type of Prolog constant, called *atoms*. Atoms are *non-numerical*, fixed concrete objects. We can tell that the terms are atoms because they *begin with a lower-case letter*.

We can summarize as follows:

Numbers- *numerical*, fixed concrete objects
↗
Constants
↘
Atoms- *non-numerical*, fixed concrete objects

To better understand Prolog data objects, let us consider the analogy to atoms in chemistry. In chemistry, atoms are the building blocks of more complex chemical structures, i.e., molecules. The same principle applies in Prolog. Atoms and numbers are the building blocks of Prolog. They are used to build more complex data objects such as facts and clauses.

C. Matching

When we ask the question

```
?- normal_boiling_point(X,77).
```

Prolog looks for a match in the database. The first time through, Prolog locates a potential match with the fact **normal_boiling_point(ethyl_acetate,77).** To complete the match, Prolog binds the free variable X to the atom **ethyl_acetate**. Variable X is no longer free. It is said to be *instantiated* to **ethyl_acetate**. When a variable is instantiated, it is locked-in and it represents a constant.

The program progresses when we enter ;. This tells Prolog to make variable X a free variable again and look for additional solutions. The second time through the database, Prolog finds a potential match with the fact **normal_boiling_point(acrylonitrile,77).** Free variable X again becomes bound, or instantiated, to another non-numerical constant, the atom **acrylonitrile**.

We should emphasize that Prolog assumes all facts in its database are true. Prolog cannot tell if a fact does not make sense. For instance, we place the following "fact" into the Prolog database:

```
made_of(moon,green_cheese).
```

This statement says in English, "the moon is made of green cheese," and is obviously incorrect. Nevertheless, when we ask Prolog the question:

```
?- made_of(moon,green_cheese).
```

Prolog answers:

yes

When writing Prolog programs, it is up to the programmer to ensure that facts placed into the database are accurate.

Now let us summarize what we have learned so far:

- Prolog uses *clauses* to define relationships. A clause consists of a *predicate* followed in parenthesis by *arguments*. All clauses must come to a full stop, indicated by a period at the end.

- A *fact* is a special type of Prolog clause, and has the general format:

    $\text{predicate}(a_1,a_2,a_3,\ \ldots,\ a_n).$

    where a_1, a_2, a_3, etc. are all arguments. Rules are another type of

Prolog clause, and they have a different format. We will discuss rules in the next section.

- A Prolog program consists of clauses and at least one *goal*. When we ask a question, it becomes the goal that Prolog attempts to satisfy.

- Arguments of relationships can be fixed concrete objects known as *constants*, or they can be flexible general objects called *variables*.

- Variables must begin with a capital letter. For example, **X**, **Y**, **Z**, **Num**, and **RefNum** are all acceptable Prolog variables.

- Non-numerical constants such as acrylonitrile and water are called *atoms*. Atoms begin with a lower-case letter and are used to build clauses.

- Numerical constants such as 100, 77, and -11.7 are *numbers*. Numbers are also used to build Prolog clauses.

- When matching a free variable, Prolog binds it to a constant. Once this step is complete, the variable is no longer an unknown, but represents a constant. At this point, the variable is said to be *instantiated*.

- When Prolog responds to a question with ***no***, the goal cannot be satisfied and therefore *fails*.

- When Prolog successfully satisfies a goal, it responds with ***yes***. If the question also includes a variable, Prolog will return the matched term.

D. Exercises

1.2.1 Suppose that we have the following **normal_boiling_point** relationships in our Prolog program database, designated *db1a*:

```
normal_boiling_point(water,100).
normal_boiling_point(isobutane,-11.7).
normal_boiling_point(n_butane,-0.5).
normal_boiling_point(polyethylene,decomposes).       % db1a
normal_boiling_point(ethyl_acetate, 77).
normal_boiling_point(acrylonitrile,77).
```

How will Prolog respond to the following queries?

```
a. ?- normal_boiling_point(water,77).
b. ?- normal_boiling_point(water,X).
c. ?- normal_boiling_point(Water,77).
```

d. ?- normal_boiling_point(X,decompose).

1.2.2 Assume that we have the following relationships in our Prolog program database:

```
alarm(low_level,on).
alarm(high_level,off).
alarm(high_temperature,off).
alarm(flame_failure,off).

pump(p2,vacuum_pump).
pump(p5,vacuum_pump).
pump(p7,product_transfer_pump).
```

Formulate Prolog questions about the following relationships:

a. What alarms are currently on?

b. What alarms are currently off?

c. What is the status of the flame-failure alarm?

d. What pump(s) is a vacuum pump(s)?

e. What purpose does pump p7 serve?

1.3 AN INTRODUCTION TO RULES IN PROLOG

A. A Description of Rules

The examples in the previous section all used facts. We can ask questions about facts, and Prolog will answer them. Answering questions about facts can be very valuable. However, solving complex engineering problems requires more of Prolog. We can increase the complexity, and thus increase the usefulness, of Prolog by adding *rules*. To learn rules, let us consider the pumping network shown in Figure 1.1.

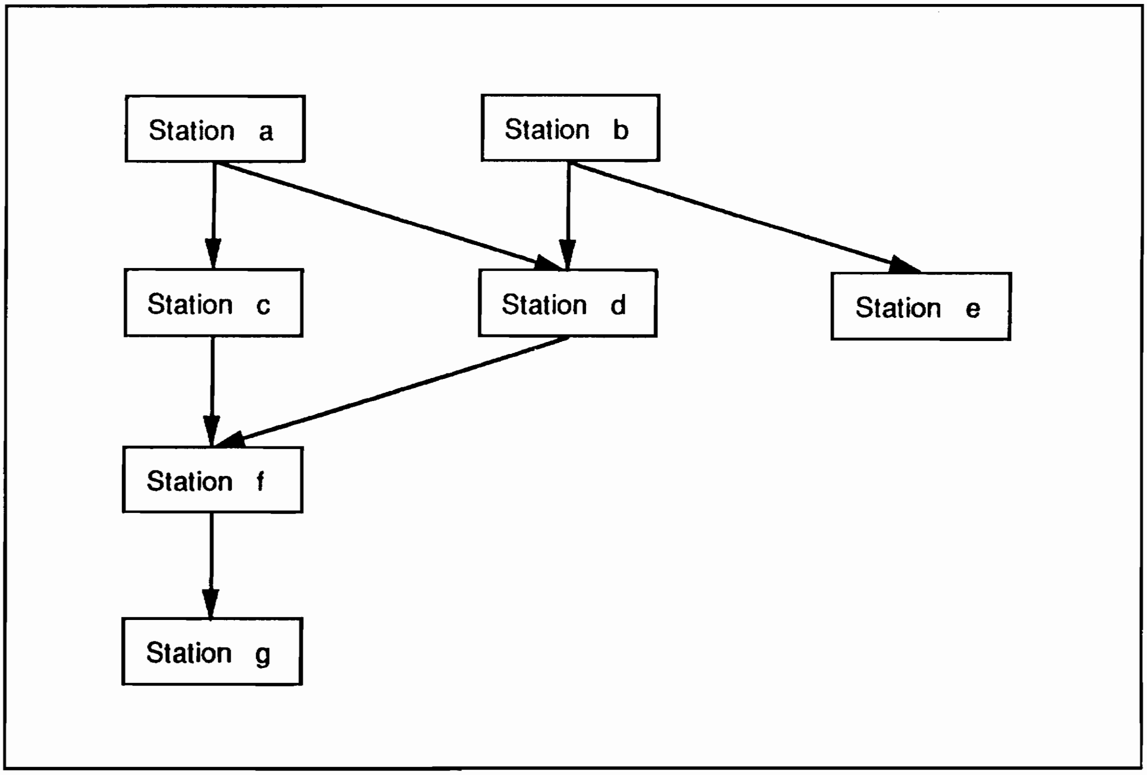


Figure 1.1. A network of pumping stations.

In this network, the product from pumping station a feeds into stations **c** and **d**. Likewise, product from station **b** feeds into stations **d** and **e**. We represent the entire network by the following facts (database *db1b*):

```
feeds_into(a,c).
feeds_into(a,d).
feeds_into(b,d).                               % db1b
feeds_into(b,e).
feeds_into(c,f).
feeds_into(d,f).
feeds_into(f,g).
```

The first fact, **feeds_into(a,c)**, corresponds to the English statement "station a feeds into station **c**". Now we can write a rule to determine if one pumping station is directly upstream of another pumping station:

```
upstream(X,Y):- feeds_into(X,Y).
```

In English, this statement says "X is upstream of Y if X feeds into Y". Here, we use capital letters X and Y to represent variables, and we read the **:-** in Prolog is read as "if".

In Prolog, both facts and rules are clauses. However, there is an important difference between facts and rules. A fact is always unconditionally true, and a rule is true only if some set of conditions is satisfied. Because rules are conditional, we view them as having two parts:

- the *condition* section, i.e., the right-hand-side, and
- the *conclusion* section, i.e., the left-hand-side.

In the above rule, Prolog is saying that the conclusion is true (X is upstream of Y) if the condition is true (X feeds into Y). We call the conclusion section of a clause the *head* of the clause, and the condition section the *body* of the clause. We can also view rules as "then-if" statements. In a Prolog rule (if you will forgive the poor grammar), THEN the conclusion is declared true, IF the condition can be proven true. Figure 1.2 summarizes the essence of rules.

```
upstream(X,Y) :- feeds_into(X,Y).
(conclusion) :-  (condition).
(then) :-        (if).
head :-          body.
```

Figure 1.2. Format of Prolog rules.

Thus, a rule has a head and a body, and is only *conditionally* true, while a fact has a head only, and is *unconditionally* true. Figure 1.3 contrasts both facts and rules.

FACT	RULE	
feeds_into(a,c).	**upstream(X,Y):-**	**feeds_into(X,Y).**
(conclusion).	(conclusion):-	(condition)
head	*head*:-	*body*

Figure 1.3. Facts and rules in Prolog.

Note the order that Prolog uses in trying to satisfy a rule: the conclusion is true if the condition can be proven true:

conclusion *(is true)* :- condition *(can be proven true)*

When a clause is entered, Prolog has no prior knowledge of whether the condition is true or not.

A close look at Figure 1.3 shows that a rule in Prolog is in the *reverse order* of the IF statement in a language such as FORTRAN. Most scientists and engineers are familiar with the conventional computer languages such as FORTRAN, Pascal, or C. In computer science, each of these languages is called a *procedural* language. To program in one of these languages, we tell the computer the step-by-step procedure required to solve a problem. Because we tell the computer the explicit procedure, these are called *procedural* languages.

Prolog is different. It is a *declarative* language. To program in

Prolog, we tell the computer the relationships between objects. *How* to solve the problem is left up to Prolog. The programmer's responsibility is to write facts and rules, thereby defining and *declaring* relationships between objects. Because we declare relationships, and do not have to tell Prolog what procedure to follow to solve a problem, Prolog is called a *declarative* language. Figure 1.4 contrasts a Prolog rule with the FORTRAN IF statement.

Prolog Rule	FORTRAN IF Statement
"THEN-IF" Rule	"IF-THEN" Rule
THEN ( ) :- IF ( )	IF ( ), THEN ( )
conclusion :- condition	IF (condition) THEN (procedure)

Figure 1.4. Comparison of a Prolog rule with a FORTRAN IF statement.

In procedural languages, IF the condition is true, THEN a procedural action is undertaken. In Prolog THEN a conclusion is declared true, IF the condition can be proven to be true.

Since Prolog is a declarative language, the following statement:

```
X :- X.
```

translated "X is true if X is true," is permissible. While this statement is *declaratively correct*, it is *procedurally useless*. Variable X will keep calling variable X forever. Declaratively, Prolog will (correctly) say "X is true if X is true." However, since variable X will keep calling variable X forever, procedurally the program makes no progress. We will discuss the procedural and declarative nature of Prolog in more detail in section 2.3.

Before we go on to using rules in Prolog, let us summarize:

- A Prolog program consists of *clauses*. A clause consists of a *predicate*, and any number of *arguments*. A clause must end with a period, which represents a full stop to the clause.

- *Facts* and *rules* are two types of Prolog clauses. Facts are always unconditionally true. Rules are conditional, with the conclusion (head) true only if its condition (body) is true.

- Prolog solves problems and gives answers when we ask *questions*.

B. Using Rules in Prolog

1. Simple Questions: One Goal and One Variable

Let us look again at the previously developed **upstream** rule coupled with the **feeds_into** database (*db1c*). Our entire database is now:

```
upstream(X,Y):- feeds_into(X,Y).
```

```
feeds_into(a,c).
feeds_into(a,d).
feeds_into(b,d).
feeds_into(b,e).                                  % dblc
feeds_into(c,f).
feeds_into(d,f).
feeds_into(f,g).
```

To recall, the **upstream** rule says in English, "X is upstream of Y if X feeds into Y."

We can now question the program. We wish to know what station is directly upstream of station f. We use the variable **X** to represent this station, and ask:

```
?- upstream(X,f).
```

Prolog first responds with:

```
X = c
```

To ask for another solution, we enter a semicolon, and will see:

```
X = c ;
X = d
```

If we ask for yet another solution, we enter ; again and see:

```
X = c ;
X = d ;
no
```

since all solutions have been exhausted.

2. Multiple Variables

Prolog is very flexible about the type of questions we can ask. It can accept questions with multiple variables. Let us ask a question with two variables, X and Y. Our database, again, is *db1c*:

```
upstream(X,Y):- feeds_into(X,Y).

feeds_into(a,c).
feeds_into(a,d).
feeds_into(b,d).
feeds_into(b,e).                                   % db1c
feeds_into(c,f).
feeds_into(d,f).
feeds_into(f,g).
```

Now we pose the question with two variables, X and Y:

```
?- upstream(X,Y).
```

Prolog first answers:

```
X = a     Y = c
```

Requests for more solutions results in the following dialogue:

```
X = a     Y = d ;
X = b     Y = d ;
X = b     Y = e ;
X = c     Y = f ;
X = d     Y = f ;
X = f     Y = g ;
no
```

3. Multiple Goals

In addition to having the flexibility of handling multiple variables in a question, Prolog can even handle two goals at once. We can write:

```
?- feeds_into(a,X),feeds_into(b,Y).
```

In English, this statement reads "What station X exists such that a feeds into X, and what station Y exists such that **b** feeds into Y ?" The comma between two clauses is treated in Prolog as an *and*. It is also called a *conjunction* of goals, i.e., both **feeds_into** goals must be satisfied for Prolog to successfully answer the question. Prolog responds with the following dialogue:

```
X = c
Y = d ;
X = c
Y = e ;
X = d
Y = d ;
X = d
Y = e ;
no
```

We can ask another question with multiple goals to further demonstrate the process. Database, *db1c*, again is:

```
upstream(X,Y):- feeds_into(X,Y).

feeds_into(a,c).
feeds_into(a,d).
feeds_into(b,d).
feeds_into(b,e).                                        % dblc
feeds_into(c,f).
feeds_into(d,f).
feeds_into(f,g).
```

We ask:

```
?- feeds_into(c,X),feeds_into(Y,a).
```

This question asks, "What station **X** exists such that **c** feeds into **X**, and what station **Y** exists such that **Y** feeds into **a** ?" The presence of the comma between clauses again indicates a conjunction. Both **feeds_into** goals must be satisfied. Looking at our database, we see that the fact **feeds_into(c,f)** satisfies the first goal. When executing, Prolog instantiates variable **X** to atom **c**. However, no fact exists that can match **feeds_into(Y,a)**. Note that the two facts in the database:

```
feeds_into(a,c).
feeds_into(a,d).
```

cannot match **feeds_into(Y,a)**, since atom a is in the wrong location.

Thus, although the first **feeds_into** goal has a match, the second does not. Since both goals cannot be satisfied, Prolog answers:

no

Because the comma, *and* (i.e., a conjunction of goals), requires Prolog to answer *both* clauses successfully for the entire question to be successful, if either goal fails, Prolog answers ***no***.

4. Rules with Multiple Conditions

We can write more complex rules in Prolog. Looking at the pumping network in Figure 1.1, we see that station **d** is a blending site. Why? Because it has feeds from two different stations, **a** and **b**, and blends them together. If we write a blending-site rule in Prolog, our first statement may be:

```
blending_site(X,Y,Z):-
            feeds_into(Y,X),
            feeds_into(Z,X).
```

In English, this rule is read as: "Station X is a blending site for stations Y and Z if Y feeds into X, and Z feeds into X". Again, the comma **,** on the right-hand side of the clause represents an *and*.

This clause differs from those discussed previously. Here, the right-hand side has *two* conditions that are required for the **blending_site** goal to be true. For Prolog to prove that the **blending_site** goal is true, it must establish that both of the **feeds_into** conditions are true. We call the two **feeds_into** conditions *sub-goals*.

The general format for a Prolog rule with multiple conditions is:

$$
\begin{aligned}
\mathtt{predicate}(a_1,a_2,a_3,\ \ldots,\ a_n)\ \texttt{:-}\ \ & \mathtt{sub\_goal1}(b_1,b_2,b_3,\ \ldots,\ b_i),\\
& \mathtt{sub\_goal2}(c_1,c_2,c_3,\ \ldots,\ c_j),\\
& \mathtt{sub\_goal3}(d_1,d_2,d_3,\ \ldots,\ d_k),\\
& .\ .\ .\ .\ .\ .\ .\ .\ .\ .\ .\ .\ .\ .\\
& \mathtt{sub\_goalp}(n_1,n_2,n_3,\ \ldots,\ n_l).
\end{aligned}
$$

where a_n, b_i, c_j, etc. are arguments. To prove that the **predicate** relationship is true, Prolog must establish that all **sub_goal** relationships are true.

We place the **blending_site** rule into the Prolog database, along with the **upstream** rule and the **feeds_into** facts. Our Prolog program now is represented by database *dbld*:

```
blending_site(X,Y,Z):-
            feeds_into(Y,X),
            feeds_into(Z,X).

upstream(X,Y):- feeds_into(X,Y).
                                                      % dbld
feeds_into(a,c).
feeds_into(a,d).
feeds_into(b,d).
feeds_into(b,e).
feeds_into(c,f).
feeds_into(d,f).
feeds_into(f,g).
```

To determine if station **c** is a blending site, and if so, from what stations it receives products, we ask Prolog the question:

```
?- blending_site(c,S1,S2).
```

where S1 and S2 are unknown variables. This question says in English "If station **c** is a blending site, from what stations S1 and S2 does station **c** receive products ?" Looking at the pumping network, we see that station **c** has only one feed stream, that being from station a. Therefore, station **c** is *not* a blending site, and we expect Prolog to answer ***no*** to the question. But surprisingly, the response is:

S1 = a

S2 = a

What happened? We have problems here!

To understand why Prolog answered this way, we must follow the step-by-step path that Prolog takes to find the answer to our question. Following this step-by-step path is called *tracing* the program. Not surprisingly, a *trace* the set of steps that results from following the step-by-step execution. The trace of **?- blending_site(c,S1,S2).** is shown in Figure 1.5 and is discussed in detail below.

Step 1: We query the system with **blending_site(c,S1,S2)**. Therefore, **blending_site(c,S1,S2)** becomes the Prolog goal.

Step 2: Prolog solves problems by trying to *match* the goal with information in the database. Pattern matching is fundamental to Prolog, and to fully understand the language, we must understand pattern matching. To match, predicate needs to match predicate, and constant needs to match constant. If a variable exists, Prolog will force matching whenever possible by binding the variable to a constant. Prolog will give a variable any value to make the two terms *identical*. The essence of pattern matching is just that -- to make two terms identical by assigning values to free variables.

Once Prolog has a goal, it starts at the top of the program and moves down in search of a match. It hits the blending-site rule, and Prolog tries to match the head of each clause. The two Prolog statements are:

blending_site(c,S1,S2) ↔ **blending_site(X,Y,Z)**

To match these, Prolog instantiates the general object (i.e., variable) **X** to the fixed object (i.e., non-numerical constant, or atom) **c**. Variable **X** is now bound. Variables **Y** and **Z** are still free variables.

Step 3: The head of the goal has matched with a clause in the database. For the blending-site rule to be true, the condition section of the clause must be proven true. Prolog now enters the body of the clause. It remembers that the variable **X** is bound to the atom **c**. Prolog attempts to prove that the rule is true. To accomplish this, it now has two sub-goals:

feeds_into(Y,c), and
feeds_into(Z,c).

Note that to prove one goal [i.e., **blending_site(c,S1,S2)**], we now have to prove two sub-goals [i.e., **feeds_into(Y,c)**, and **feeds_into(Z,c)**].

Step 4: Again Prolog tries to match the new goals. Starting at the top of the program and working down, it tries to match **feeds_into(Y,c)**. The first fact it hits is **feeds_into(a,c)**. Predicate **feeds_into** matches predicate **feeds_into**. Atom **c** matches atom **c**. Prolog instantiates variable **Y** to the atom **a**. We have a match.

```
PROGRAM:  feeds_into(a,c).
          feeds_into(a,d).
          feeds_into(b,d).
          feeds_into(b,e).
          feeds_into(c,f).
          feeds_into(d,f).
          feeds_into(f,g).
          blending_site(X,Y,Z):-
                feeds_into(Y,X),
                feeds_into(Z,X).
```

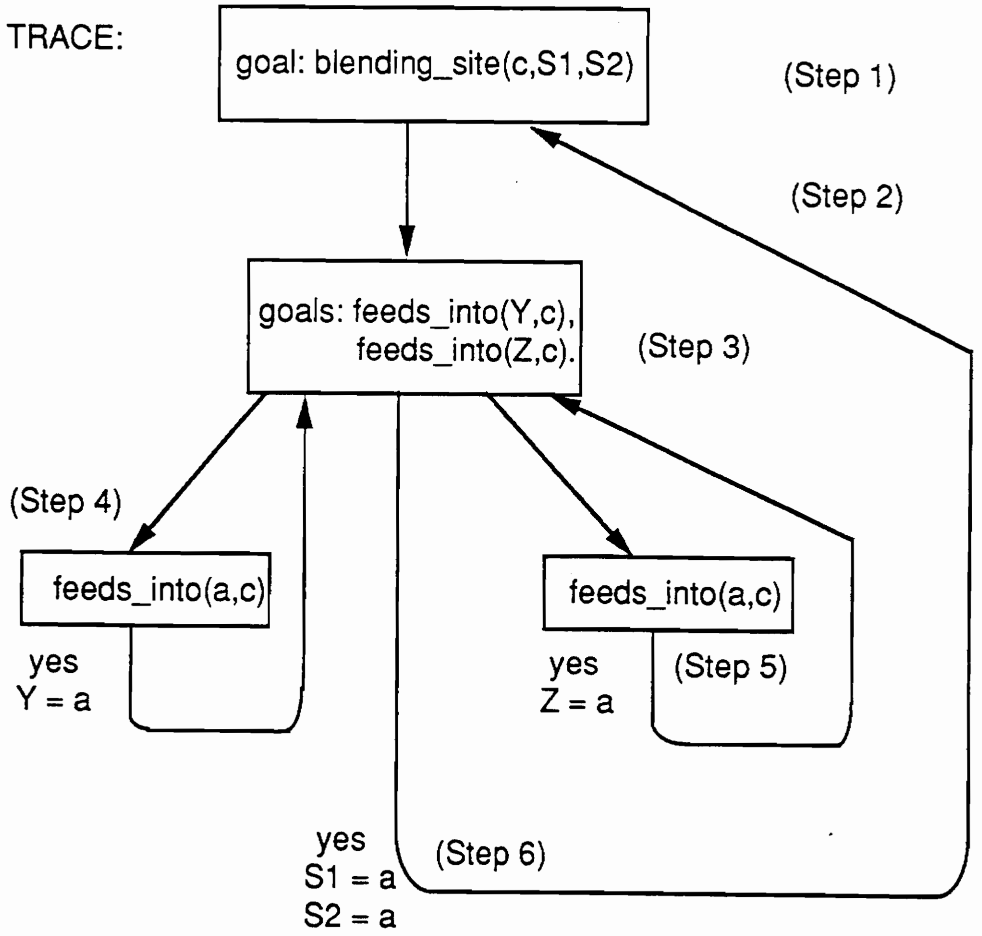


Figure 1.5. Trace of blending_site(c,S1,S2).

Step 5: To prove that the original **blending_site** rule is true, Prolog still has one more sub-goal to fulfill: **feeds_into(Z,c)**. As in step 4, Prolog starts at the top, moves down to match, and finds one with **feeds_into(a,c)**. As a result, variable **Z** is instantiated to atom **a**.

Step 6: The **blending_site** rule has been proven true. Variables S1 and S2 are both instantiated to the atom **a**, and Prolog reports:

```
S1 = a
S2 = a
```

We now understand Prolog's logic, but the program does not do what we originally intended. When we defined our blending-site rule, we certainly did not intend for station **c** to be a blending site with feed from station a twice! But, based on the rule we wrote, Prolog's response was perfectly logical. *Our rule says nothing about Y and Z not being the same pumping station.* Prolog assumes that variables Y and Z are independent of each other, and can be instantiated to *any* object to make the rule work. They can even be instantiated to the same object, in this case, the atom a.

To correct the problem, we introduce the relationship of **inequality**, denoted in Prolog by **\=**. We write the inequality **Y ≠ Z** in Prolog as:

```
     Y \= Z
```

This relationship is true is if **Y** *does not* match **Z**. If **Y** does match **Z**, the relation is false and Prolog fails. Thus, our **blending_site** rule becomes:

```
     blending_site(X,Y,Z):-
          feeds_into(Y,X),
          feeds_into(Z,X),
          Y \= Z.
```

This says in English, "Station X is a blending site for stations Y and Z if Y feeds into X, Z feeds into X, and Y and Z are different stations." Our database, *dble*, is now:

```
     blending_site(X,Y,Z):-
               feeds_into(Y,X),
               feeds_into(Z,X),
               Y \= Z.

     upstream(X,Y):- feeds_into(X,Y).
                                                       % dble
     feeds_into(a,c).
     feeds_into(a,d).
     feeds_into(b,d).
```

```
feeds_into(b,e).
feeds_into(c,f).
feeds_into(d,f).
feeds_into(f,g).
```

If we now query the system:

```
?- blending_site(c,S1,S2).
```

Prolog responds with:

```
no
```

But if we query with:

```
?- blending_site(f,S1,S2).
```

Prolog responds with:

```
S1 = c
S2 = d
```

If we query with:

```
?- blending_site(S,S1,S2).
```

Prolog first responds:

```
S = d
S1 = a
S2 = b;
```

When we ask for another solution, Prolog responds:

```
S = f
S1 = c
S2 = d;
```

Finally, when we ask again for another solution, Prolog responds:

```
no
```

since all solutions have been exhausted.

To summarize this section, we have seen that:

- Prolog programs are made up of *facts*, *rules*, and *questions*.

- A *fact* is a Prolog clause that is always *unconditionally* true.

- Facts are clauses with no body, while questions have a body but no head.

- A *rule* is a Prolog clause that is true if a certain set of *conditions* are met.

- Rules are clauses consisting of a *head* and a *body*. The body is a collection of sub-goals that must be proven true before the head is declared true. The sub-goals are separated by a comma, which is understood to mean *and*.

- In a procedural language such as FORTRAN, an IF statement is used to execute a procedure if conditions are met. In a declarative language like Prolog, the IF ":-" used in rules is in reverse order. A statement is declared true if the conditions can be proven true. This is shown below:

FORTRAN	→	**"IF-THEN" rule**
Prolog	→	**"THEN-IF" rule**

C. Exercises

1.3.1 Translate the following statements into Prolog rules:

a. X is downstream of Y if Y feeds into X (introduce the relation **downstream**).

b. X is upstream of Y if X feeds into Z, and Z feeds into Y.

c. X is downstream of Y if Y feeds into Z, and Z feeds into X.

1.3.2 Translate the following Prolog rules into English.

a.
```
corrosive(X):- strong_acid(X).
```

b.
```
organic(X):- contains(X,carbon).
```

c.
```
two_feeds(X):- feeds_into(Z,X),
               feeds_into(Y,X),
               Y \= Z.
```

d.
```
explosion_hazard(X,Y):- explosive(X),
                        strong_oxidizer(Y).
```

1.3.3 Develop the Prolog rule for an intermediate station in the pumping network shown in Figure 1.1. Introduce the relation **intermediate_station**; note that an intermediate station must receive product from one station, and pump it to another.

1.3.4 Develop three separate Prolog rules that say the following:

a. The ionic nature of X is cationic if X is positively charged.

b. The ionic nature of X is anionic if X is negatively charged.

c. The ionic nature of X is nonionic if X is uncharged.

Introduce the relation **ionic_nature**, and use this predicate for all three rules. For supporting relations in the body of the clause, use the predicate **charge**.

1.4 RECURSIVE RULES IN PROLOG

A. Introduction to Recursion

We have previously developed the rule "Station X is upstream of station Y if station X feeds into station Y":

```
      upstream(X,Y):- feeds_into(X,Y).
```

This **upstream** rule is part of our original pumping network database, *db1c*:

```
      upstream(X,Y):- feeds_into(X,Y).

      feeds_into(a,c).
      feeds_into(a,d).
      feeds_into(b,d).
      feeds_into(b,e).                                     % db1c
      feeds_into(c,f).
      feeds_into(d,f).
      feeds_into(f,g).
```

The **upstream** rule works if X is the next station upstream of Y. But looking at the database, station **a** is upstream of **f**, and station **b** is upstream of **g**. The **upstream** rule as written, however, cannot handle those cases. It can only deal with the case where station **X** is *directly* upstream of station Y.

We wish to generalize the upstream rule, so that no matter how many stations are between **X** and Y, Prolog will still be able to handle the situation. A more accurate set of rules for **upstream** is:

```
upstream(X,Y):-
       feeds_into(X,Y).

upstream(X,Y):-
      feeds_into(X,Z),
      feeds_into(Z,Y).

upstream(X,Y):-
      feeds_into(X,Z1),
      feeds_into(Z1,Z2),
      feeds_into(Z2,Y).
. . . . . . . . . . . . .
```

This set of rules takes into account the chain of pumping stations to determine if one is upstream of another.

The second rule says "X is upstream of Y if X feeds into Z, and Z feeds into Y". It handles the situation if **X** is two stations upstream of Y. The third rule handles the situation if **X** is three stations upstream of

Y.

We are making the **upstream** rule more general. We have written three rules that enable Prolog to handle three separate conditions. Although our approach will work, it is getting cumbersome. It is lengthy, and is limited by the number of **feeds_into** sub-goals we have. If we have a chain of thirty pumping stations, we would need to write thirty **feeds_into** rules. Prolog must have a better way to handle this situation.

There is a remedy to the problem. Let us introduce the concept of a *recursive* **upstream** rule. For example, we say:

```
X is upstream of Y if:
      1) X feeds into Y,                    (rule 1)
            or
      2)  X feeds into Z and               (rule 2)
          Z is upstream of Y.
```

In Prolog, we write:

```
upstream(X,Y):-                                % rule 1
      feeds_into(X,Y).
upstream(X,Y):-                                % rule 2
      feeds_into(X,Z),
      upstream(Z,Y).
```

Rule 2 is a *recursive* rule. This rule defines **upstream** in terms of itself, i.e., **upstream** calls **upstream** in search of a solution.

Defining a rule in terms of itself is called a *recursive* rule. Rule 2 shown above is recursive. Logically, the statements above are correct. We can legitimately say, "X is upstream of Y if X feeds into Z, and Z is upstream of Y."

Although the recursive rule is logically correct, can Prolog handle a rule defined in terms of itself? In FORTRAN, if a subroutine calls itself, an error results. Prolog, however, can indeed handle recursive rules. In fact, recursion is an integral part of efficient Prolog programming.

Our database, *db1f*, has become:

```
upstream(X,Y):- feeds_into(X,Y).
upstream(X,Y):- feeds_into(X,Z),
                upstream(Z,Y).

feeds_into(a,c).
feeds_into(a,d).
feeds_into(b,d).
feeds_into(b,e).                          % db1f
feeds_into(c,f).
feeds_into(d,f).
feeds_into(f,g).
```

Now we query the system:

```
      ?- upstream(a,g).
```

Prolog responds:

```
      yes
```

We ask:

```
      ?- upstream(e,g).
```

and Prolog responds:

```
      no
```

We ask:

```
      ?- upstream(X,g).
```

and Prolog first responds:

```
      X = f
```

Prolog reached this answer using the first **upstream** rule, i.e., **upstream(X,Y):- feeds_into(X,Y).** This rule is not recursive, and says X is upstream of Y only if X is *directly* upstream. Upon repeated queries for more solutions, Prolog responds with the following dialogue:

```
X = c ;
X = d ;
X = a ;
X = b ;
no
```

Prolog reaches these results using the second, recursive **upstream** rule, i.e.,

```
upstream(X,Y):- feeds_into(X,Z),
                upstream(Z,Y).
```

B. Processing Recursive Rules in Prolog

To understand how Prolog handles recursive rules, we trace through an example problem. Suppose we have the following database in our Prolog program:

```
upstream(X,Y):-
```

```
          feeds_into(X,Y).
     upstream(X,Y):-
          feeds_into(X,Z),
          upstream(Z,Y).

     feeds_into(a,c).
     feeds_into(a,d).                   % db1f
     feeds_into(b,d).
     feeds_into(b,e).
     feeds_into(c,f).
     feeds_into(d,f).
     feeds_into(f,g).
```

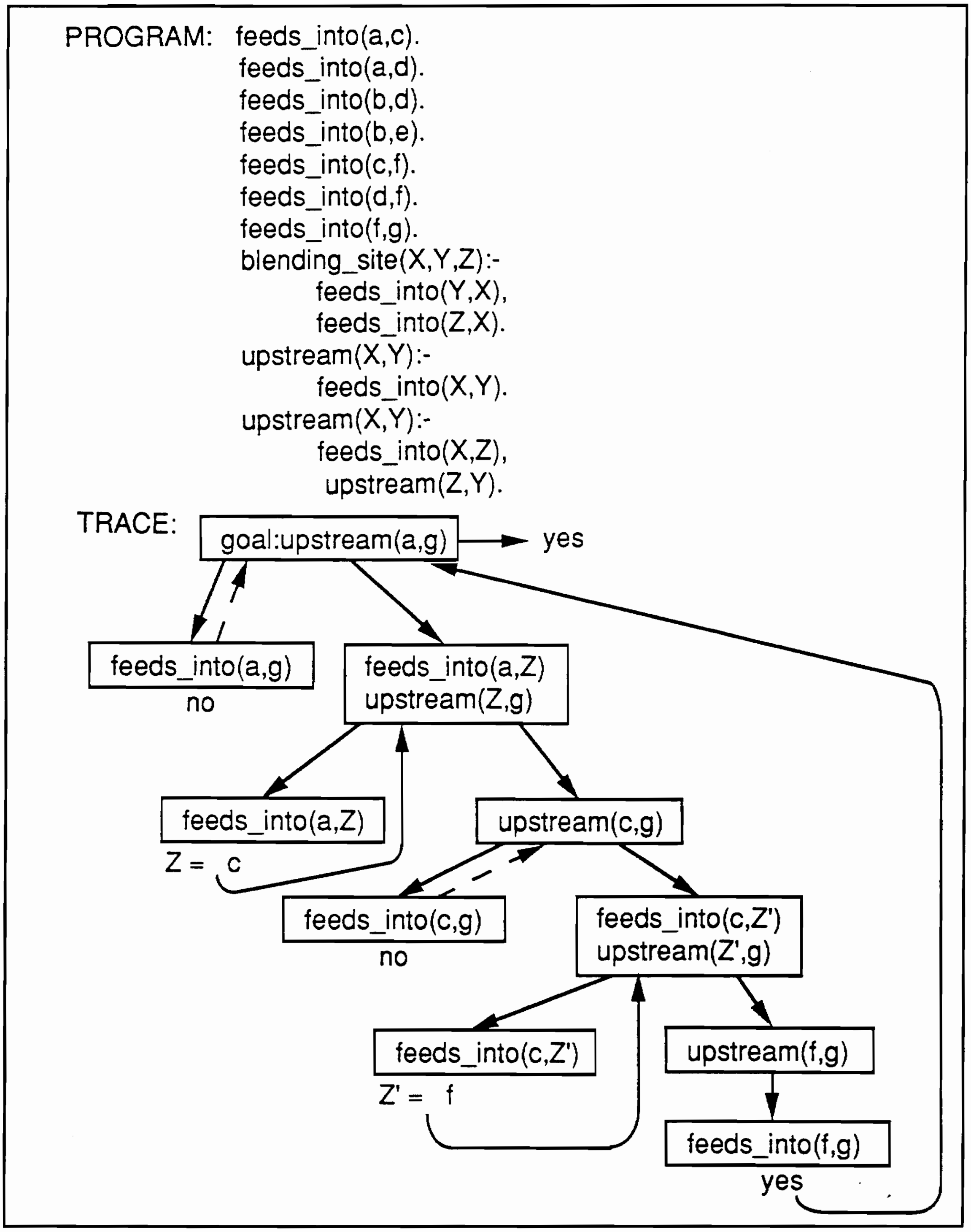
Now we pose the question:

```
     ?- upstream(a,g).
```

Prolog responds:

```
     yes
```

To arrive at this answer, Prolog takes the steps shown in Figure 1.6. A description of the trace is below.



```
PROGRAM:  feeds_into(a,c).
          feeds_into(a,d).
          feeds_into(b,d).
          feeds_into(b,e).
          feeds_into(c,f).
          feeds_into(d,f).
          feeds_into(f,g).
          blending_site(X,Y,Z):-
                 feeds_into(Y,X),
                 feeds_into(Z,X).
          upstream(X,Y):-
                 feeds_into(X,Y).
          upstream(X,Y):-
                 feeds_into(X,Z),
                  upstream(Z,Y).
```

Figure 1.6. The upstream recursive rule.

Step 1: Posing the initial question causes **upstream(a,g)** to become the goal.

Step 2: Prolog searches for a match, beginning at the top of the program. It scans down the program until it hits the first **upstream(X,Y)** rule. The head of this rule matches the current goal. A potential match exists if Prolog can now match the body of the rule. Prolog instantiates the variable **X** to atom **a** and the variable **Y** to **g**. The new goal becomes **feeds_into(a,g)**.

Step 3: Searching for a match, Prolog again does a top-down scan. It looks at all of the **feeds_into** facts, and finds that none of them matches the current goal **feeds_into(a,g)**. Therefore, the current goal fails.

Step 4: All is not lost, however. There are two **upstream** rules in the database. Only the first one failed. Prolog realizes this, and *reinstates* **upstream(a,g)** as the goal. It then jumps down to the second **upstream** rule in search of a match. This process of changing paths to find a solution when a goal fails is called *backtracking*. Prolog backtracks to determine an alternate way to solve the original goal. When Prolog backtracks, it *releases* the binding of instantiated variables **X** and **Y** from atoms a and **g** in step 2. Variables **X** and **Y** are now free variables.

Step 5: To backtrack, Prolog moves to rule 2, that is:

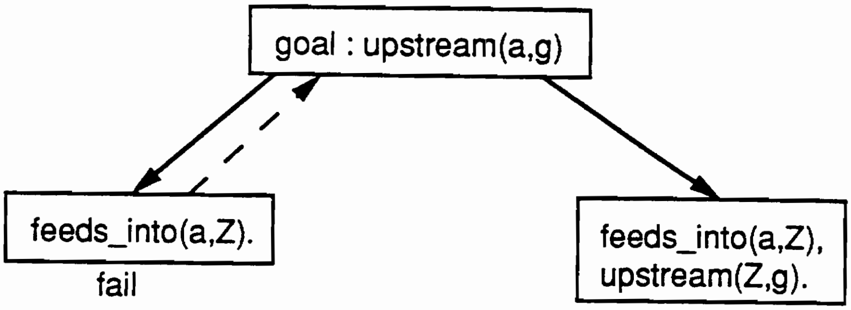
```
upstream(X,Y):-
        feeds_into(X,Z),
        upstream(Z,Y).
```

Here, the goal **upstream(a,g)** first matches the head of this clause. The variable **X** is instantiated to a and **Y** is instantiated to **g**.

Step 6: Prolog moves into the body of the clause. The new goals are now:

```
feeds_into(a,Z),
upstream(Z,g).
```

Note that Prolog has substituted the single goal **upstream(a,g)** for the two goals **feeds_into(a,Z)**, and **upstream(Z,g).** The backtracking done in steps 3 and 4 is shown below:



Step 7: The next goal Prolog needs to achieve is **feeds_into(a,Z)**. Scanning from the top, Prolog matches with the first fact in the program, i.e., **feeds_into(a,c)**. Prolog instantiates variable **Z** to atom **c**, and moves on to the next goal. The next goal is **upstream(Z,g)**. However, since variable **Z** is instantiated to atom **c**, the goal is **upstream(c,g)**.

Step 8: Prolog again does its top-down scan, and matches the **upstream(c,g)** with the head of the first **upstream** clause. This second application of the **upstream** rule is *completely* independent of the first application. We use **X′** and **Y′** to show that they are variables different from **X** and **Y**:

```
    upstream(X',Y'):-
         feeds_into(X',Y'),
```

Note that variable **X′** is instantiated to **c** and **Y′** to **g**.

Step 9: The next goal is **feeds_into(c,g)**. As in step 3, this goal fails, since no matching of the **feeds_into** facts exists. Prolog backtracks to the next upstream rule, freeing (uninstantiating) variables **X′** and **Y′** in the process.

Step 10: Going to our second upstream rule, we remember that variables are local to that clause only. Therefore we can write this rule as:

```
          upstream(X',Y'):-
                feeds_into(X',Z'),
                upstream(Z',Y').
```

Variables **X'** and **Y'** become bound to atoms **c** and **g** by matching the head of the **upstream** clause with the goal **upstream(c,g)**. The two new goals now become:

```
          feeds_into(c,Z'),
          upstream(Z',g).
```

Step 11: Scanning from the top of the database, the first goal **feeds_into(c,Z')** succeeds, matching **feeds_into(c,f)**. Variable **Z'** is instantiated to atom **f**. The second goal now becomes **upstream(f,g)**. This new goal initiates another recursive call.

Step 12: Prolog tries to match **upstream(f,g)**. Starting from the top-down again, Prolog matches the head of the first upstream rule. Again because of the local nature of clause variables, we write this rule as:

```
     upstream(X'',Y''):-
          feeds_into(X'',Y'').
```

since variables **X''** and **Y''** are totally separate from **X**, **Y**, **X'**, and **Y'**.

Prolog instantiates **X''** to atom **f**, and **Y''** to atom **g**. The new goal now becomes **feeds_into(f,g)**.

Step 13: Scanning from the top, Prolog matches the goal **feeds_into(f,g)** with the **feeds_into(f,g)** fact in the database. All goals have been achieved, and Prolog responds ***yes***.

The complete trace of the program shown in Figure 1.6 can be viewed as a tree. The progression of Prolog through the tree toward the solution is called the *search*. All artificial intelligence programs require some type of search. When Prolog enters a branch of a tree and fails to find the solution within that branch, it *backtracks* to another section of the tree in the search for the solution.

It is important to realize that the entire Prolog search, including scanning for matches, recursion, and backtracking, is done automatically and is not seen by the user.

C. Exercises

1.4.1. Given the database below, what will be Prolog's answers to the following questions?

```
upstream(X,Y):-
      feeds_into(X,Y).
```

```
upstream(X,Y):-
        feeds_into(X,Z),
        upstream(Z,Y).

feeds_into(a,c).
feeds_into(a,d).
feeds_into(b,d).
feeds_into(b,e).
feeds_into(c,f).
feeds_into(d,f).
feeds_into(f,g).

a. ?- upstream(b,g).
b. ?- upstream(g,_).
c. ?- upstream(b,X).
d. ?- upstream(X,f).
```

1.4.2 Define a recursive **downstream** relation as the reverse of the **upstream** relation in exercise 1.4.1. Use two rules, one of which is recursive.

1.5 CHAPTER SUMMARY

The following summarizes the key concepts of this chapter:

- Prolog is a *declarative* language, consisting of three basic actions:

  - *Declaring facts*- about objects and their relationships,
  - *Defining rules*- about objects and their relationships, and
  - *Asking questions*- about these same objects and their relationships to solve complex problems.

- Relationships in Prolog are written in the form of *clauses*. There are three type of clauses: *facts*, *rules* and *questions*.

- Clauses in Prolog must have a predicate. They can have any number of arguments, including zero. The following are acceptable Prolog clauses:

```
proton.
neutron.
electron.
```

Each of these are facts with no arguments.

• Facts in Prolog are *unconditionally* true. Facts are clauses that have a head with no body. Their general format is:

$$\mathtt{predicate}(a_1,a_2,a_3,\ \ldots,\ a_n).$$

where a_1, a_2, a_3, etc. are all arguments.

• Rules are clauses with a *head* and a *body*. The head is the *conclusion* to be proven, while the body consists of *conditions* required for the clause to be true. Their general format is:

$$\begin{aligned}\mathtt{predicate}(a_1,a_2,a_3,\ \ldots,\ a_n)\ \mathtt{:-}\ \ & \mathtt{sub\_goal1}(b_1,b_2,b_3,\ \ldots,\ b_i),\\ & \mathtt{sub\_goal2}(c_1,c_2,c_3,\ \ldots,\ c_j),\\ & \mathtt{sub\_goal3}(d_1,d_2,d_3,\ \ldots,\ d_k),\\ & .\ .\ .\ .\ ,\\ & \mathtt{sub\_goalp}(n_1,n_2,n_3,\ \ldots,\ n_l)\end{aligned}$$

where a_1, a_2, a_3, ..., a_n, etc. are arguments.

• Rules tie together other rules and facts, and allow Prolog to infer one conclusion from another.

• When trying to satisfy a goal, Prolog always begins at the top of the program and scans down looking for a match. A match occurs when the predicate and all the arguments of the goal are exactly the same as the predicate and arguments of a statement in the database.

- Following Prolog through its execution in a step-by-step fashion is called a program *trace*.

- A goal in Prolog *succeeds* if a match is found. If no match is found, the goal *fails*.

- A *recursive* rule is a clause that is defined in terms of itself. Recursion is an essential part of Prolog programming.

- As Prolog looks for a solution, its pathway can be viewed as a *tree*. If it fails in a branch of the tree, Prolog automatically *backtracks* in search of a solution.

- The progression of Prolog through the tree looking for a solution is called the *search*. All artificial intelligence programs require some type of search.

A LOOK AHEAD

In the next chapter, we shall get into more detail concerning Prolog. We shall discuss the types of data allowed, their syntax, and how Prolog matches objects. We shall also get into the declarative and procedural nature of Prolog, and contrast it in more depth with conventional languages such as FORTRAN.

REFERENCES

Two highly recommended general references for Prolog programming are by Ivan Bratko (1990), and Clocksin and Mellish (1987).

Bratko, Ivan, *Prolog Programming for Artificial Intelligence*, second edition, pp. 3-27, Addison-Wesley, Reading, MA (1990).

Burnham, W.D. and A.R. Hall, *Prolog Programming and Applications*, pp. 1-16, Halsted Press, a division of John Wiley & Sons, New York, NY (1985).

Clocksin, W.F. and C.S. Mellish, *Programming in Prolog*, third edition, pp. 1-19 Springer-Verlag, New York, NY (1987).

Ford, N., *Prolog Programming*, pp. 9-22, John Wiley & Sons, New York, NY (1989).

Garavaglia, Susan, *Prolog Programming Techniques and Applications*, pp. 12-23, Harper & Row Inc., New York, NY (1987).

Kluzniak, F. and S. Szpakowicz, *Prolog for Programmers*, pp. 1-24, Academic Press, Orlando, FL (1985).

Rowe, Neil C., *Artificial Intelligence Through Prolog*, pp. 4-25, Prentice-Hall, Englewood Cliffs, NJ (1988).

Schnupp, P. and L. Bernhard, *Productive Prolog Programming*, pp.1-28, Prentice-Hall, Englewood Cliffs, NJ (1986).

Sterling, L. and Shapiro E., *The Art of Prolog*, pp. 1-27, MIT Press, Cambridge, MA (1986).

Walker, Adrian, Editor, *Knowledge Systems and Prolog*, pp. 25-35, Addison-Wesley, Reading, MA (1987).

2

A MORE IN-DEPTH VIEW

This chapter presents a more in-depth view of Prolog. First, we describe data representation and syntax. Then, we move on to a critical aspect of Prolog programming: matching. To understand matching, we look "inside" Prolog to see exactly what steps it takes. This examination, in turn, leads to an understanding of both the declarative and procedural nature of Prolog.

2.1 DATA REPRESENTATION AND SYNTAX

Figure 2.1 shows all forms of data objects allowed in Prolog.

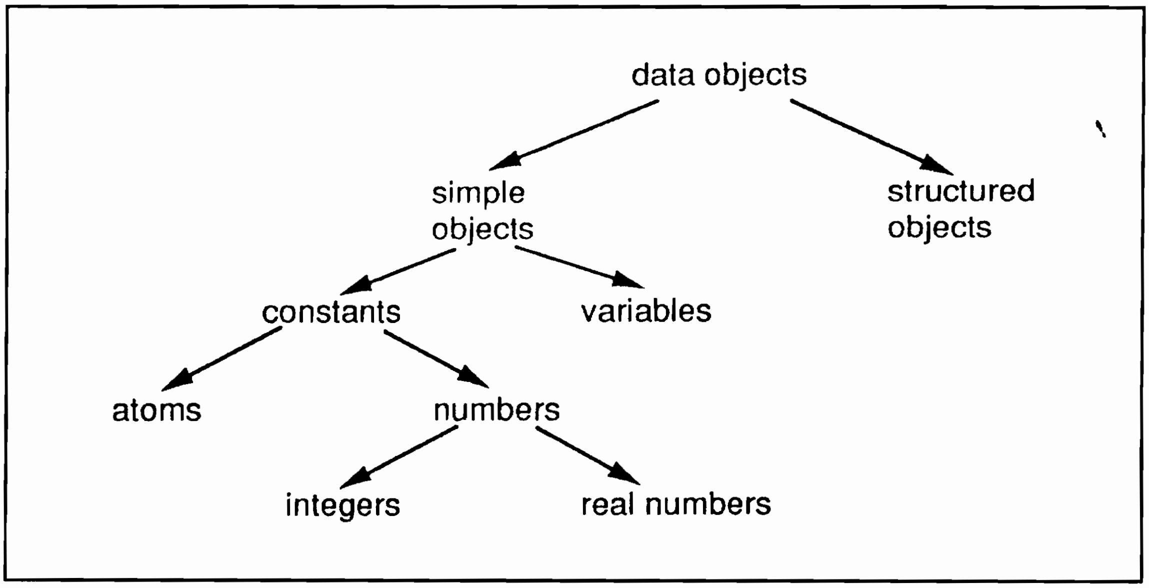


Figure 2.1. Data objects in Prolog.

Prolog has two general types of data objects: simple and structured. Simple objects have only one component. For instance, the objects **propane, 12.5, n_pentane** are all single-component statements and are therefore simple objects. But the following objects:

hydrocarbon(propane)

water_soluble_polymer(polyvinyl_alcohol, polyacrylate)

have more than one component and are not simple objects. An object with multiple components is called a *structured object.*

Simple objects can be either ***constants*** or ***variables***. Constants are fixed, concrete Prolog objects that never change as the program progresses. ***Numbers***, which can be either real numbers or integers, are *numerical* constants. ***Atoms*** are *non-numerical* constants.

In chemistry, atoms and numbers are the building blocks for more complex chemical structures, i.e., molecules. Thus, the atoms carbon (C), hydrogen (H), and oxygen (O), together with numbers 1, 2, and 5, can be used to build the ethanol molecule, C_2H_5OH. The same principle applies in Prolog. We use non-numerical atoms and numerical numbers to form "molecule-like" fixed, concrete Prolog objects, called constants. Constants are the building blocks for more complex structures such as clauses.

Variables are the second type of simple object. In contrast to constants which are fixed, variables are general objects that can take on different values as the Prolog program executes.

A. Constants

Atoms, integers and real numbers are the simplest type of data objects in Prolog. They are always constants.

1. Atoms

Prolog has a high degree of flexibility in constructing atoms. Atoms can consist of strings of:

- letters (a,A,b,B,...,Z)
- digits (0,1,2,...,9)
- other signs or characters such as _ & = + - # ! < > / @

To recall, some atoms used in chapter one were:

decomposes

acrylonitrile

ethyl_acetate

normal_boiling_point

All of these atoms begin with a lower-case letter. In addition, some use the underscore character _ to improve legibility. We could write a string like this:

testfortechnicalfeasibility

but inserting the underscore character between words makes the atom more readable:

test_for_technical_feasibility

However, atoms do not have to take this form. In practice, we have three possible methods for constructing atoms in Prolog:

Method 1: a string of letters, digits, and other characters (such as & # $ and _), starting with a lower-case letter. For example:

```
decomposes
ethyl_acetate
pump_number_1
proposalA
proposal_A
plan_c_item_3_column_2
x2A1
m13&21
c&21$22
```

Method 2: a string of characters enclosed in single quotes. This method is helpful if we want a data object to begin with a capital letter, but do not want it to be a variable. This happens, for instance, when referring to proper nouns. For example:

```
'Chemical X'
'Mr. Engineer'
'Company Y'
```

```
'Cumene'

'Ethyl_Chloride'
```

Method 3: Strings of special characters. For example:

```
=====>

<=:=>

...

<-----

------>
```

This method enables us to enhance the symbolic meaning of the program. For instance, we can use ----> in the statement

```
A ----> B
```

to say symbolically, "A implies B."

Using method 3 requires some care, however, because some character strings have predefined meanings in Prolog. For instance, the term :- is an atom in Prolog, but it has a special, *built-in* meaning. It means "if" in the Prolog rule format

```
conclusion :- condition.
```

The :- character, if used incorrectly, may cause Prolog to take unintended actions. Other characters with built-in meaning include = + - : *, all of which could cause unexpected outcomes if used incorrectly. We discuss these in more detail in section 3.1.

Of the three methods, we generally will prefer methods 1 and 2 for engineering analyses. Method 3, while useful in some applications, is less valuable to the engineer. In addition, method 3 is more error-prone. For these reasons, most engineering applications use methods 1 and 2 almost exclusively.

2. Numbers

Numbers in Prolog come quite easily for most engineers. Numbers in Prolog can be either integers or real numbers. They are summarized below:

Integers- integers are designated by a string of digits without a decimal point, such as:

```
    0
  -45
15000
   -1
```

Most versions of Prolog limit the range of integers we can use. Virtually all allow integers between - 16383 and + 16383 (this corresponds to ± 2^{14} integers, if zero is included). However, versions of Prolog geared for scientific and engineering applications allow a much wider range.

Real Numbers- real numbers are designated by a decimal point, such as:

```
  0.
-45.0
  3.14159
 -1.0
```

The syntax for real numbers depends on the version of Prolog we are using. While all versions require real numbers to have a decimal point, a few allow exponents, such as 1.2E09.

3. Methods of Evaluation and Computation

Prolog is primarily a language of symbolic, rather than numerical, computation. For instance, assume we want a computer to evaluate the function

$$f\,(x) = \int_0^x m^2\,dm$$

at various values of x. We can instruct the computer to evaluate the function symbolically or numerically.

Most engineers are familiar with numerical evaluation. To evaluate the above function on a computer, we can perform a numerical integration. We determine the area underneath the curve using, for instance, the trapezoidal rule or Simpson's rule. The area under the curve is equal to the numerical evaluation of the integral.

The function can also be evaluated symbolically. From calculus, we know that

$$f\,(x) = \int_0^x m^2\;dm = \frac{m^3}{3}\,\Big|_{m=0}^{m=x} = \frac{x^3}{3}$$

To evaluate the function symbolically, we need a set of rules. We define three rules:

- Rule number one symbolically integrates the expression using the rules of calculus, and converts $\int m^2$ dm as follows:

$$\int m^2\;dm = \frac{m^3}{3}$$

- Rule number two realizes that the expression is a definite integral, and evaluates the upper and lower bounds, 0 and x, respectively, and then gives a final algebraic form. The steps it involves are:

$$\frac{m^3}{3} \Big|_0^x = \frac{1}{3} \bullet x^3 - \frac{1}{3} \bullet 0 = \frac{x^3}{3}$$

- Rule number three takes the final algebraic form, places in numerical values for x, and gives the ultimate numerical result for each value of x.

For example, a symbolic integration of the mathematical statement

$$\int x^b \, dx = \frac{x^{b+1}}{b+1}$$

is written in Prolog as:

```
integral(X,B,Result):-
                                Result = ( X ** (B+1)) / (B+1).
```

In English, this says "The integral X to the exponential power B is equal to Result if Result equals X to the exponential power B plus one divided by the sum B plus one."

Because Prolog is primarily a language for symbolic computing, we generally do not need to do anything beyond simple arithmetic. Most other more complex operations (such as integration) are done symbolically. For this reason, Prolog's floating-point arithmetic and real-number processing capabilities have been historically weak. Some versions of Prolog do *not even allow* real numbers.

However, as scientific and engineering applications of Prolog have increased, the demand for floating-point arithmetic has also increased. Many new versions of Prolog, therefore, support floating-point arithmetic.

Prolog is limited in its ability to do large-scale "number crunching" problems such as matrix algebra or finite-element analyses. A procedural language, such as FORTRAN, is more suited for these applications. Prolog is best for relational and symbolic analyses like those of artificial intelligence.

B. Variables

1. Nature of Variables

Constants are data objects that cannot change as Prolog searches for a solution. In contrast, variables can change as the program progresses and the variables become instantiated to different objects. Prolog assumes that a free variable such as X can take on *any* data object.

Variables begin with capital letter or the underscore character. For example:

```
X
S1
Solution
Problem_Result
_X1
```

_12

We need to consider two other aspects of variables to ensure proper Prolog programming: the *scope* of variables, and the use of *anonymous* variables.

2. Scope of Variables

Variables in Prolog are local to individual clauses only. By "local," we mean that if X occurs in *separate* clauses, it represents a *different* variable in each clause. If X occurs more than once in the *same* clause, it represents the *same* variable each time. For example:

```
Clause 1:  upstream(X,Y):-
                  feeds_into(X,Y).
Clause 2:  upstream(X,Y):-
                  feeds_into(X,Z),
                  feeds_into(Z,Y).
```

Variables are *local* to individual clauses. In clause 1, the X in **upstream(X,Y)** represents the *same* variable as the X in **feeds_into(X,Y).** Comparing clauses 1 and 2, we find, however, the X in the **upstream(X,Y)** of clause 1 is a *completely different* variable from the X in **upstream(X,Y)** of clause 2. We could just as easily write:

```
Clause 1:  upstream(X,Y):-
                feeds_into(X,Y).
Clause 2:  upstream(L,M):-
                feeds_into(L,N),
                feeds_into(N,M).
```

and the program would have exactly the same meaning.

3. Use of Anonymous Variables

When a variable is used only once in a clause, it does not need to be named and can remain anonymous. The underscore denotes an anonymous variable. As an example, consider again our pumping network introduced in chapter one, shown again in Figure 2.2.

We can develop a rule that defines pumping station X as an *intermediate station* if: 1) some station feeds into X, and 2) X also feeds into some other station. We write this rule in Prolog as:

```
    intermediate_station(X):-
        feeds_into(Y,X),
        feeds_into(X,Z).
```

Here, variables Y and Z are only used once. An equivalent Prolog statement using anonymous variables is:

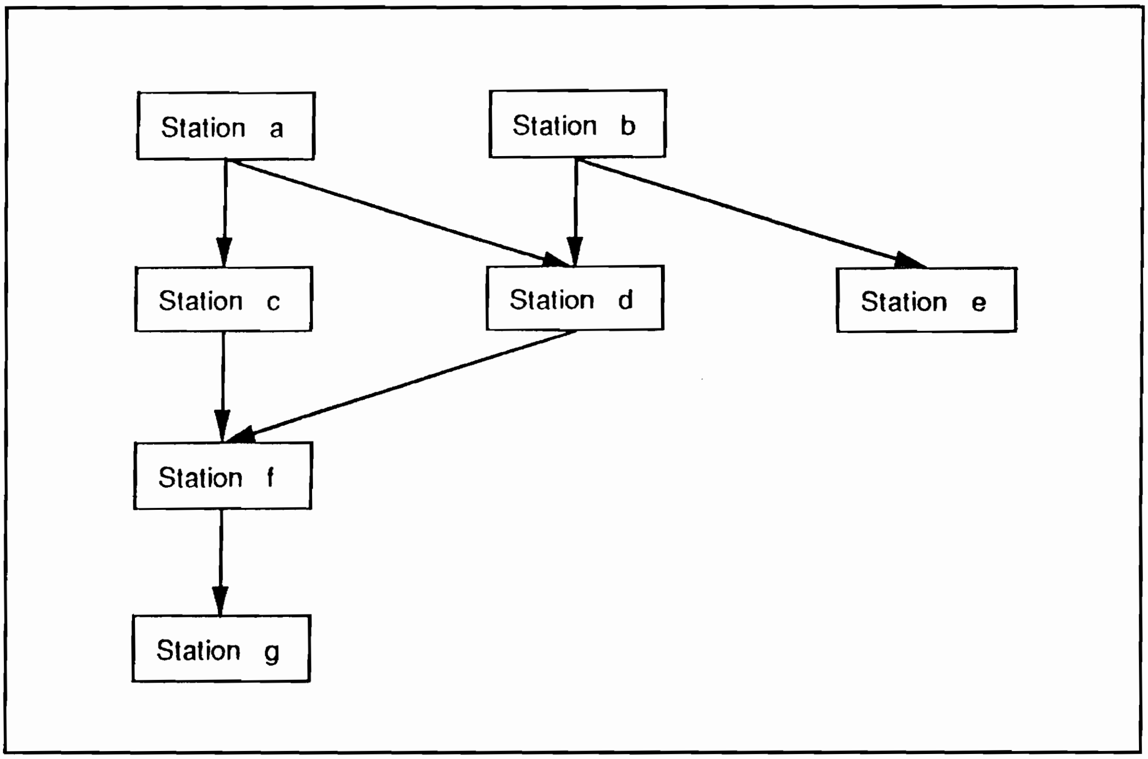


Figure 2.2. A network of pumping stations.

```
feeds_into(X,_).
```

Each time an underscore is used in Prolog, it denotes a completely *new* variable. For instance, we write:

```
a_pumping_station_exists:- feeds_into(_,_).
```

Here, the two underscores represent two different anonymous variables. An equivalent statement is:

```
a_pumping_station_exists:- feeds_into(X,Y).
```

```
      a_pumping_station_exists:- feeds_into(X,Y).
```

Why do we use anonymous variables? They make the program easier to read and understand. In a complex Prolog clause with many variables, using anonymous variables wherever possible makes the program more understandable. The second **intermediate_station** rule above uses anonymous variables and is more readable than the first rule that does not use anonymous variables.

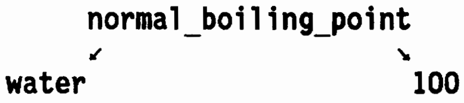
C. Simple and Structured Objects

Simple objects are objects containing only one component. Variables, atoms, and numbers are examples of simple objects:

```
      X
      feeds_into
      _x1
      23.2
      10
      'Benzene'
```

Structured objects (also known as *structures*) are data objects containing more than one component or simple object. As an example, we consider the normal boiling point of a material. The normal boiling point of water is

100 °C. This statement can be represented as a structured object:



Here, **normal_boiling_point** relates the atom **water** with the number **100**. We generate a structured object by combining simple objects. Specifically, we combine **normal_boiling_point** with **water** and **100** to give the structure:

```
normal_boiling_point(water,100).
```

which is a Prolog clause (a fact) with the predicate **normal_boiling_point** and arguments **water** and **100**.

In the above clause, **normal_boiling_point** is called the *functor*, and **water** and **100** are its *components*. All Prolog structures must have a functor. The functor names the Prolog structure. Functors in Prolog can be viewed much like functions in mathematics. Consider the function f(x) defined below:

$$f(x) = \sin(x) + \cos(x)$$

The function f(x) relates, or *maps*, the term x into the right-hand side of the equation. Similarly, in the clause

```
normal_boiling_point(water,100)
```

the functor **normal_boiling_point** *maps* the objects **water** and **100**.

The components of the structure are enclosed in parenthesis, and can themselves be structures. For instance, the following structure is acceptable in Prolog:

```
cost(project,operating(100),capital(2000)).
```

and is translated as: "The project cost is 100 operating [cost] and 2000 capital [cost]."

In the above examples, the functor maps constants (i.e., atoms and numbers such as **water**, **100**, or **project**) or other structures (e.g., **operating(100)**). Functors can also map variables. For instance,

```
normal_boiling_point(water,X).
normal_boiling_point(X,100).
normal_boiling_point(X,Y).
```

are all acceptable Prolog structures.

Now let us consider the functor **feeds_into**. If pumping station a feeds into pumping station **c**, we write:

```
feeds_into(a,c).
```

Suppose we know the pumping station **a** feeds into **c**, which also feeds into station f. This information is captured in a single structure as:

```
feeds_into(a,c,f).
```

In the statements **feeds_into(a,c)** and **feeds_into(a,c,f)**, Prolog views each **feeds_into** as a different functor. They have the same *name* (i.e., **feeds_into**). However, one has two arguments and the other has three, so Prolog views them as two separate functors.

The *arity* of a functor refers to how many arguments the functor has. The **feeds_into(a,c)** has an arity of two. The **feeds_into(a,c,f)** has an arity of three. Functors with different arities are treated as different, even if they have the same name. Thus, to fully specify a functor, we must have:

1) the *name*, which must be an atom, and
2) the *arity*, i.e., the number of arguments.

As mentioned, the components inside a Prolog structure can themselves be structures. Consider the pumping system shown in Figure 2.3. We can describe this system with the following facts using the **feeds_into** functor:

```
feeds_into(a,b).
```

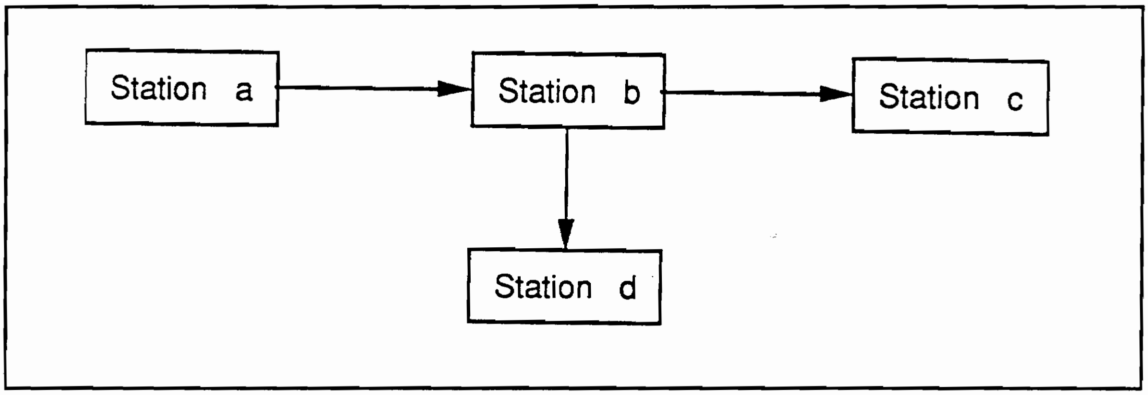


Figure 2.3. A sequential pumping system.

We can also combine these facts into a single functor, called **network**:

```
network(feeds_into(a,b),feeds_into(b,c),feeds_into(b,d)).
```

Here, **network** is a functor with an arity of three. The arguments within the **network** structure are themselves structures, defined by the **feeds_into** functor. The outer functor, in this case **network**, is called the *principal functor*. The principle functor is always outside the parenthesis, and names the overall structure. Previously, we saw the Prolog fact:

```
cost(project,operating(100),capital(2000)).
```

Here, the atom **cost** is the principal functor.

D. Summary

Let us summarize what we have learned in this section.

- Data objects in Prolog can be either *simple* or *structured* objects.

    - *Simple objects* have only one component. They can be either *constants* or *variables*.

        - *Constants* are fixed, concrete objects whose values do not change as the Prolog program executes. They can be either *atoms* or *numbers*.

            - *Atoms* are non-numerical constants that can be identified one of three different ways:

              (1) a string of letters, digits, and other characters, beginning with a lower-case letter, e.g., **ethyl_chloride**, **x2A#3**, **c2&d3$23**.

              (2) a string of characters enclosed in single quotation marks, e.g., **'Company Y'**, **'Mr. Engineer'**.

              (3) a string of special characters, such as **--->** or **===>**.

- *Numbers* can be either *integers* or *real numbers*. *Integers* are strings of digits without a decimal point, e.g., **125**, **1356**, or **- 10**. *Real numbers* contain a decimal point, e.g., **125.1**, **- 3.14159**.

- *Variables* are flexible, general objects that can change as a Prolog program executes. To be a variable, an object must start with a capital letter or the underscore character, e.g., **X**, **S12**, **_xl**, **Result**. Variables are *local* to individual clauses only. If the variable **X** occurs in separate clauses, it represents *different* variables.

- *Structured objects* contain more than one component, i.e., more than one simple object.

- All Prolog structures must have a *functor*. The functor is outside of the parenthesis of the structure, and gives the structure its name. To fully specify a functor, we must have its *name*, which must be an atom, and its *arity*, i.e., the number of arguments in the structure.

- Arguments within Prolog structures can themselves be structures.

E. Exercises

2.1.1 Is the following Prolog syntax correct? If not, state why not. If they are correct, identify the type of data object for each syntax (i.e., atom, integer, real number, variable, structure).

a. **benzene**

b. **Benzene**

c. **_benzene**

d. **1_butene**

e. **'Benzene'**

f. **18**

g. **14.7**

h. **_14**

i. **composition(carbon,hydrogen(deuterium),oxygen)**

j. **Composition(carbon,hydrogen,oxygen)**

k. **+ (3,2)**

2.1.2 Identify principal functor and the arity for each of the following structures:

a. **feeds_into(a,c).**

b. feeds_into(a,c,f).

c. network(feeds_into(a,c),network(feeds_into(b,d),feeds_into(d,f))).

2.1.3 Given the following Prolog database, predict how Prolog will respond to questions a-f.

```
feeds_into(a,b).
feeds_into(b,c).
feeds_into(c,d).
feeds_into(c,e).

upstream(X,Y):- feeds_into(X,Y).
upstream(X,Y):- feeds_into(X,Z),
                feeds_into(Z,Y).

downstream(X,Y):- feeds_into(Y,X).

intermediate_station(X):- feeds_into(_,X),
                          feeds_into(X,_).
```

a. ?- feeds_into(c,X).

b. ?- upstream(a,c).

c. ?- upstream(X,Y).

d. ?- intermediate_station(X).

```
e. ?- downstream(X,_).

f. ?- upstream(X,Y),downstream(Y,X).

g. ?- upstream(X,Y),downstream(X,Y).
```

2.2 MATCHING

A. Instantiation of Variables

A variable in Prolog is different from a variable in a procedural language such as FORTRAN. *A Prolog variable represents a specific object, while a FORTRAN variable corresponds to a location in memory.* In a procedural language such as FORTRAN, we represent the assignment statement of J to one as:

```
J = 1
```

This statement instructs the computer to procedurally replace any previous value of J (in some location of memory) with the value 1. A variable in FORTRAN stands for a store location in memory.

In Prolog, however, the statement:

```
J = 1
```

must be viewed in terms of an important concept called *matching* or *unification*. When Prolog encounters this statement, three things can happen:

(1) If J has already been instantiated to 1, the match *succeeds*.

(2) If J has already been instantiated to some other value, atom, or structure (e.g., **station_a**, 3.1416, **feeds_into(a,b)**, etc.), the match *fails*.

(3) If J is currently *uninstantiated* (i.e., a free variable), the match *succeeds* and J becomes instantiated to the number 1 to make the statements J and 1 identical.

Unification is the process of making two previously different data objects (J and 1) identical. Prolog uses matching to effect unification.

A *free variable* in Prolog represents some currently unknown object. A *bound variable* is a variable that has been instantiated, through Prolog's matching mechanism, to a specific object. Thus in cases 1 and 2, J is bound prior to the statement **J = 1**. In case 3, **J** is initially free, but becomes bound once the statement is executed.

B. Requirements for a Match

The equal sign = instructs Prolog to match the two sides of the statement. When we write the statement **J = 1**, we invoke Prolog's matching mechanism and are instructing Prolog to match the variable **J** with the constant 1.

Prolog goes through a specific procedure when attempting to match. With the statement **J = 1** above, we saw three different routes Prolog could take depending on the status of the variable **J**.

Now let us get more general. Assume we have two objects A and B. Prolog follows three "rules of matching" to determine if A and B match:

Matching Rule 1: *if A and B are constants (or bound variables), the match of A and B succeeds only if they are the same simple object.*

For example, if A is instantiated to the atom **distillate** and B is likewise instantiated to **distillate**, then the match succeeds. If B is instantiated to something else, the match fails. Consider the Prolog clause:

```
equal(X,Y):- X = Y.
```

This says "X is equal to Y if X and Y match." We can then ask the question:

```
?- equal(distillate,distillate).
```

Prolog will respond

yes

When we ask Prolog the question, variables **X** and **Y** are both bound to the atom **distillate**. Entering the **equal** clause, Prolog encounters the statement:

```
X = Y
```

Since both variables are bound to the atom **distillate**, the statement effectively reads:

```
distillate = distillate
```

The match succeeds, and Prolog reports the result, i.e., ***yes***. If we ask the question

```
?- equal(distillate,bottoms).
```

Prolog responds ***no***. The question fails on the **X = Y** statement, which effectively reads:

```
distillate = bottoms
```

These two atoms are obviously not identical and the match fails.

Matching Rule 2: *if A is a free variable and B is any object (including a free variable, atom, number, structure, etc.), the match succeeds and A is instantiated to B.*

For example, if A is a free variable and B is instantiated to the integer 0, then the match succeeds and A is instantiated to the integer 0.

If A and B are *both* free variables when matched, the match again succeeds; A and B are bound to each other. At this point, they are both still free variables, but they must share the same data object. If one is later instantiated to a constant, the other will take the same value.

Consider the following sequence of goals, where **A** and **B** are initially free variables:

```
B = 0,
A = B,
. . .
```

In the first statement, Matching Rule 2 applies, since **B** is a free variable and **0** is any object (in this case, an integer). This match also succeeds, and **B** is instantiated to **0**. In the second statement, Matching Rule 2 again applies, since **A** is a free variable and **B** is any object (in this case, bound to the integer 0). Prolog views the attempted match as:

```
A = 0.
```

The match succeeds, and Prolog instantiates free variable A to the integer 0 also. At the end of the sequence, both **A** and **B** are bound to the same object, the integer **0**.

Now let us demonstrate a more subtle aspect of Matching Rule 2. We change the sequence of goals:

```
A = B,
B = 0,
. . .
```

To fulfill these goals, Prolog encounters the first statement:

```
A = B
```

Here, we attempt to match variables **A** and **B**, both of which are still free variables. Matching Rule 2 applies, and the match succeeds. Prolog binds the two variables to each other, *even though they are both still free variables*. Variables **A** and **B** are now said to *share* the same, currently unknown object. Prolog always maintains maximum flexibility when instantiating variables. Since there is no need to bind variables **A** and **B** to a constant at this point, Prolog does not do it. They remain free variables, but they share the same data object, i.e., they are instantiated to each other.

Now Prolog encounters the next statement:

```
B = 0
```

Variable **B** is free. The match succeeds, and **B** is instantiated to the integer 0. From the previous step, Prolog remembers that **A** and **B** share the same data object. Therefore, Prolog immediately instantiates **A** to the integer 0 also. The matching is complete, and at the end of the sequence, variables **A** and **B** are both instantiated to the integer 0.

Matching Rule 3: *if A and B are structured objects, the match succeeds if: 1) A and B both have the same principal functor, and 2) all arguments within the functor match. Any free variables are instantiated to make the terms identical.*

Matching Rule 3 can be subtle. Consider the following statement:

```
feeds_into(a,b) = feeds_into(a,b,c).
```

This match fails. One **feeds_into** functor has an arity of two (i.e., has two arguments), while the other has three. Thus the components within each structure cannot match.

Consider the following:

```
feeds_into(a,b,c) = feeds_into(X,Y,Z).
```

The match of these two structures succeeds (assuming that X, Y and Z are all free variables), resulting in the following instantiations:

```
X = a
Y = b
Z = c
```

Now we consider the statement:

```
feeds_into(X1,b,c) = feeds_into(X,Y,Z).
```

This match also succeeds, with the following instantiations:

```
X1 = X
Y  = b
Z  = c
```

Here, the two variables, **X1** and **X**, are instantiated to each other, but they are still free variables. Prolog maintains maximum flexibility, and does not instantiate them to a constant until necessary. Variables **X1** and **X** share the same data object.

Table 2.1 summarizes the "rules of matching".

Table 2.1. The three rules of matching applied to the statement A = B.

Rule	Status of Objects		Criteria for a Match
	A	B	
1 A=B	bound simple object	bound simple object	A and B must be the same data object.
2 A=B	free variable	any object, bound or free	Match succeeds. A is instantiated to B.
3 A=B	structured object	structured object	A and B must have the same principal functor, and all arguments within the functor must match. Instantiation is performed on any free variables.

To further elaborate on the process involved in matching, we ask the question:

```
?- feeds_into(X1,b,c) = feeds_into(X,Y,Z),
   feeds_into(X1,b,c) = feeds_into(a,b,c).
```

Prolog responds:

```
X1= a
X = a
Y = b
Z = c
```

Let us go through the step-by-step matching process to see how Prolog arrives at this result:

Step 1:

Prolog looks at the first question, **feeds_into(X1,b,c) = feeds_into(X,Y,Z)**. It is comparing two structures, so it uses Rule 3. Both structures have the same principal functor (i.e., **feeds_into**) and the same arity (i.e., three). A potential match exists.

Step 2:

Looking within each structure, Prolog matches the components. The following instantiations are performed:

X1 = X
Y = b
Z = c

Note that **X** is instantiated to **X1**, and both **X** and **X1** are still free variables.

Step 3:

Prolog now moves to the second question [i.e., **feeds_into(X1,b,c) = feeds_into(a,b,c)**]. Again, both functors match and have the same arity. Prolog looks inside each structure, and a match does exist. Constants **b** and **c** match constants **b** and **c**. In addition, variable **X1** matches constant **a**. Prolog instantiates **X1** to **a**.

Step 4:

At this point, **X** and **X1** are instantiated to each other, while **X1** is instantiated to a. Prolog now enforces consistency. Since **X** and **X1** share the same data object, **X** is also instantiated to **a**. The matching is complete, and Prolog reports:

```
X1= a
X = a
Y = b
Z = c
```

When matching structures, Prolog always begins at the principal functor. The functor name and arity must be the same. If they are, Prolog then attempts to match the arguments inside parenthesis. It instantiates variables as necessary to maintain a consistent match.

Free variables, when matched to each other, share the same, currently unknown data object. Prolog keeps the *most general* (i.e., nonspecific) instantiation possible, while still maintaining consistency. It does not bind free variables to constants until it is absolutely forced to do so. Free variables remain free but share the same object. As the program progresses and more matching occurs, variables normally become instantiated to increasingly more specific terms.

C. Equality

As seen in sections 2.2A and 2.2B, the statement **J = 1** is treated much differently in Prolog compared to a procedural language such as FORTRAN. In actuality, the atom **=** is a ***built-in predicate***, i.e., it is reserved by Prolog and is recognized as having special meaning. A different, and perhaps more consistent way to write the equality **J = 1** is:

```
= (J, 1).
```

Here, **=** is the functor, and **J** and **1** are the arguments.

As mentioned, the built-in predicate **=** has special meaning in Prolog. The statement **= (J, 1)** is translated to mean "J matches 1." It tells Prolog to perform the three rules of matching on the two components in the structure, i.e., **J** and **1**.

The complete statement, **= (J, 1)**, is an accurate Prolog statement, but it is also a cumbersome, inconvenient way to see if two objects match. People feel more at home with the easy and familiar format **J = 1**.

For convenience, instead of requiring the cumbersome format **= (J, 1)**, Prolog allows us to write the **=** predicate in the form of:

```
J = 1
```

We must remember, however, that = is a built-in predicate with special meaning. It instructs Prolog to perform the three rules of matching on the components within the structure.

D. Exercises

2.2.1 Considering the statements below, with all variables being free and uninstantiated. Will the matches succeed? If not, why not? If they do succeed, what will be the instantiations of variables after Prolog executes the statements?

a. **feeds_into(a,b) = feeds_into(X,Y)**

b. **feeds_into(a,b,c) = feeds_into(X,Y)**

c. **network(X,Y) = network(feeds_into(a,b), feeds_into(A,B))**

d. **plus(1, 1) = 2**

e. **feeds_into(X,Y) = feeds_into(Z,_)**

2.2.2 Define the relation **three_equal** that succeeds if all three components of the structure match, i.e., the statement **three_equal(a,a,a)** succeeds, while **three_equal(a,a,b)** fails.

2.2.3 Define the relation **two_of_three_equal** that succeeds is any two of the three components of the structure are equal. For example, the following statements succeed:

```
    two_of_three_equal(a,a,a).

    two_of_three_equal(a,a,b).

    two_of_three_equal(a,b,a).

    two_of_three_equal(b,a,a).
```

and the following statement fails:

```
    two_of_three_equal(a,b,c).
```

2.3 DECLARATIVE AND PROCEDURAL NATURE OF PROLOG

A. Understanding Declarative and Procedural Languages

1. <u>Declarative Nature</u>

As mentioned, Prolog is a *declarative* language. The term "declarative" implies that we declare logical relations in terms of variables, and Prolog determines what relations are true for each variable. We focus on the relationship itself. Thus, when we write:

```
    W if X and Y and Z.
```

which in Prolog is written:

```
    W:- X, Y, Z.
```

what matters to us is the relationship among **W**, **X**, **Y**, and **Z**.

The declarative translation of this clause is: "W is true if X and Y and Z are all true". It is declarative because only the logical relations concern us. *How* Prolog solves it is not important.

2. <u>Procedural Nature</u>

Prolog, however, also has a *procedural* side. "Procedural" focuses on *how* Prolog solves the problem. Consider again the clause:

```
W:- X,Y,Z.
```

The procedural translation of this clause is: "to solve W, first solve X, then solve Y, and finally solve Z". As another example, consider the built-in predicate =. When we write the statement

```
X = Y.
```

we are instructing Prolog to execute a specific *unification* or *matching* procedure dictated by the rules of the = predicate. Although Prolog is primarily a declarative language, it does have an essential procedural nature.

3. Prolog versus Conventional Programming Languages

In contrast with a conventional programming language such as FORTRAN, Prolog is a "higher level" language. We write relations, and let Prolog do the work. *How* it solves problems does not matter as long as the relations are correct. Ideally, to solve problems using Prolog, we need to:

- understand and declare proper relations, and

- allow Prolog's built-in matching and search procedure to do the work.

FORTRAN, by contrast, is a procedural language. To solve FORTRAN problems, we need a step-by-step solution procedure called an ***algorithm.*** Specifically, we need to:

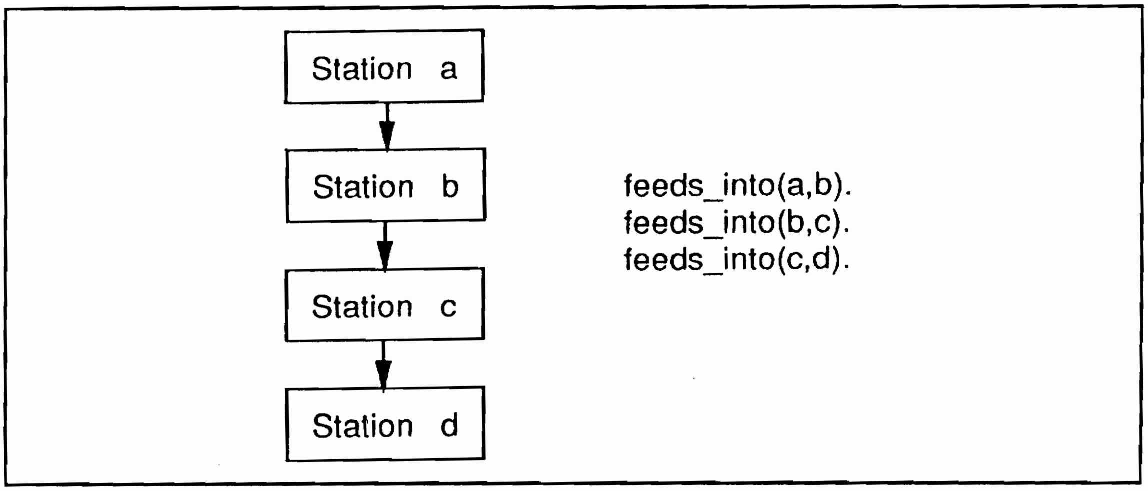
- understand and encode proper relations, and
- specify the *exact* order of computation via an algorithm.

B. Ordering of Clauses: The Danger of Infinite Loops

Although Prolog is primarily declarative, it does have a procedural side. To build efficient, productive Prolog programs, we must pay attention to Prolog's procedural nature. Consider the simple pumping network shown in Figure 2.4, which includes the **feeds_into** facts representing the structure of the network.

Previously, we defined the **upstream** relation using two rules, where one was recursive. Our previous definition was:

```
upstream(X,Y):-
               feeds_into(X,Y).
upstream(X,Y):-
               feeds_into(X,Z),
               upstream(Z,Y).
```



```
feeds_into(a,b).
feeds_into(b,c).
feeds_into(c,d).
```

Figure 2.4. Another pumping network.

If we ask the following question:

```
?- upstream(d,a)
```

Prolog correctly responds ***no***, since station **d** is not upstream of station a. When Prolog encounters the first rule with the goal **upstream(d,a)**, the sub-gaol becomes **feeds_into(d,a)**. This goal fails, since there is no matching **feeds_into** fact in the database. When Prolog tries to match the second rule, the sub-goal becomes:

```
feeds_into(Z,a)
```

This goal also fails, since no fact exists with atom **a** as the second argument. Since no matches exist, Prolog responds ***no***.

But now, let us change the order of the sub-goals in the second clause. Our new definition for **upstream** becomes:

```
upstream(X,Y):-
               feeds_into(X,Y).
upstream(X,Y):-
               upstream(Z,Y),
               feeds_into(X,Z).
```

These statements are declaratively correct. The first rule says "X is upstream of Y if X feeds into Y". The second rule says "X is upstream of Y if Z is upstream of Y, and X feeds into Z". We again ask the question:

```
?- upstream(d,a).
```

Figure 2.5 traces the procedural steps Prolog takes in solving this problem.

Step 1: The initial goal is **upstream(d,a)**. Prolog scans top-down, and enters the first **upstream** clause. Variables **X** and **Y** are instantiated, and the new goal becomes **feeds_into(d,a)**.

Step 2: The **feeds_into(d,a)** goal fails, since no match is found. Prolog backtracks, and enters the second **upstream** clause. The head of the clause matches, and variables **X** and **Y** are instantiated to atoms **a** and **d**, respectively.

Prolog hits the statement **upstream(Z,Y)**. Variable Y is instantiated to atom a, and Z is still free. Therefore, Prolog executes the recursive call, and the new goal becomes **upstream(Z,a)**.

Step 3: Prolog again scans top-down. The goal matches the head of the first upstream clause. Entering the first **upstream** clause, the new goal becomes **feeds_into(Z,a)**.

Step 4: The **feeds_into(Z,a)** goal fails as in step 2, since no matching fact can be found. No fact exists in the database with atom a in the second position. Prolog backtracks and enters the second **upstream** clause. Another recursive call yields the new goal: **upstream(Z′,a)**.

Step 5: We are right where we were at the end of step 2. In step 2, the goal was **upstream(Z,a)**; here, the goal is **upstream(Z′,a)**. As in steps 3 and 4, the first **upstream** rule fails and the second **upstream** rule executes a recursive call. The new goal becomes: **upstream(Z′′,a)**.

Note that in steps 4 and 5, a prime is placed on each variable, i.e., **Z′** and **Z′′**. These primes indicate that **Z′** and **Z′′** are completely different variables.

The result? An infinite loop! Prolog will keep searching futilely, going deeper and deeper into recursive calls. An infinite set of goals results:

```
upstream(Z,a)

upstream(Z',a)

upstream(Z'',a)

upstream(Z''',a)

. . .
```

and no solution is found. These goals only differ in their first variables (Remember, **Z** is different from **Z'**, which is different from **Z''**, etc.) Prolog cannot make any progress and will not realize the futility of deeper and deeper recursive calls. Eventually, the computer will run out of memory and the program will "crash."

```
PROGRAM:  feeds_into(a,b).
          feeds_into(b,c).
          feeds_into(c,d).

          upstream(X,Y):-
              feeds_into(X,Y).

          upstream(X,Y):-
              upstream(Z,Y),
              feeds_into(X,Z).
```

TRACE:

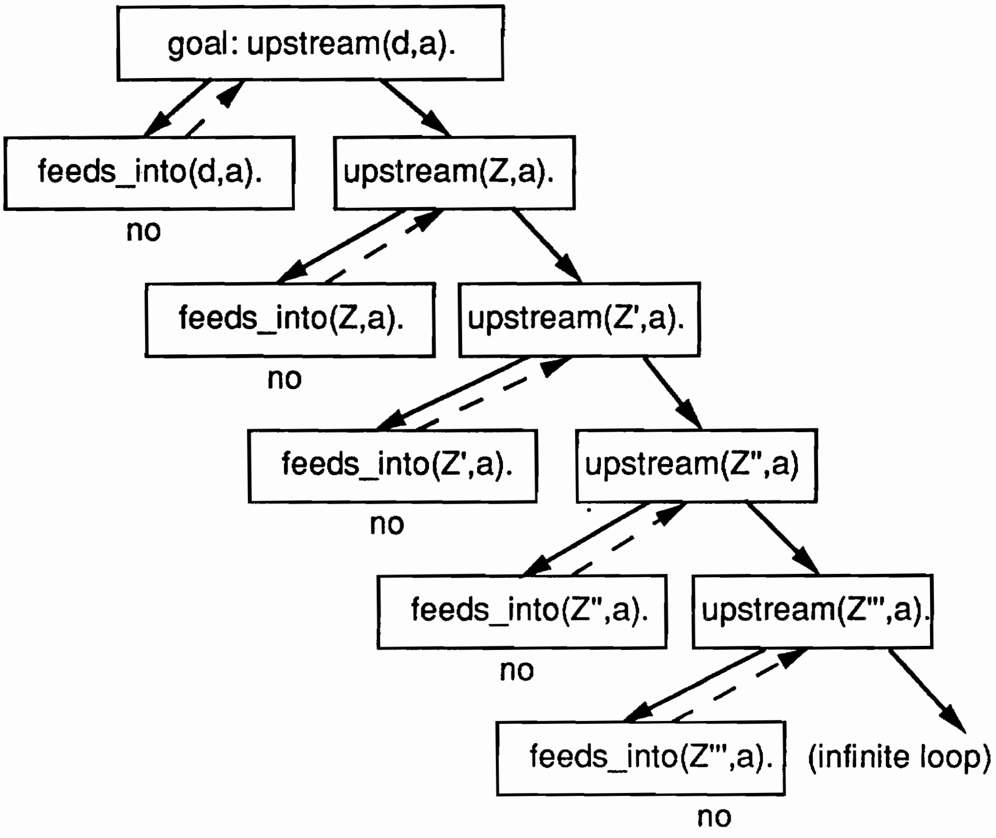


Figure 2.5. Trace of upstream(d,a).

As this example demonstrates, we cannot ignore the procedural nature of Prolog. If we do, we may run into such serious problems as infinite loops. For example, the simple statement:

```
X :- X.
```

is declaratively correct, but procedurally useless. Variable X will keep calling variable X forever.

In the **upstream** clause above, a simple change of the rule order, although declaratively correct, resulted in procedural disaster. In this case, a solution does exist, but Prolog will never find it.

Infinite loops are not unique to Prolog. They can also occur in procedural languages such as FORTRAN. Extra caution does need to be exercised in using recursion, however. A unique aspect of Prolog is that a program can be declaratively correct, but procedurally incorrect. Although the relations may be specified correctly, Prolog may not be able to satisfy a goal because it repeatedly goes down the wrong pathway.

C. Techniques in Ordering Clauses

We need to be careful when utilizing recursion, but we need not view recursion as an infinite loop waiting to happen. Following certain techniques will help "toughen up" the program and prevent infinite loops. The simplest techniques are to:

- delay recursive calls to as late as possible, and
- call the "simpler" rules first, such as the matching of facts.

As an example, let us consider again the **upstream** relation used above. This relation consists of two clauses. The most robust form of ordering in these clauses is:

Set One:

```
upstream(X,Y):-
    feeds_into(X,Y).

upstream(X,Y):-
    feeds_into(X,Z),
    upstream(Z,Y).
```

This ordering is both procedurally and declaratively correct, and will give the proper answer to *any* query. To see how well this set-up performs, let us pose some questions and trace the results.

If we first ask **upstream(a,d)**, Prolog solves this question quickly and efficiently, as Figure 2.6 shows. It created ten search "blocks" before reaching the answer.

Figure 2.7 shows the trace for the goal **upstream(d,a)**. Again Prolog solves this question quickly, creating only four search blocks. In the last block, the goal **feeds_into(d,Z)** fails with **Z** as a free variable, since no matching fact exists.

To see how ordering affects recursion efficiency, let us consider the ordering shown in Set Two.

```
feeds_into(a,b).
feeds_into(b,c).
feeds_into(c,d).

upstream(X,Y):- feeds_into(X,Y).
upstream(X,Y):- feeds_into(X,Z),
                upstream(Z,Y).
```

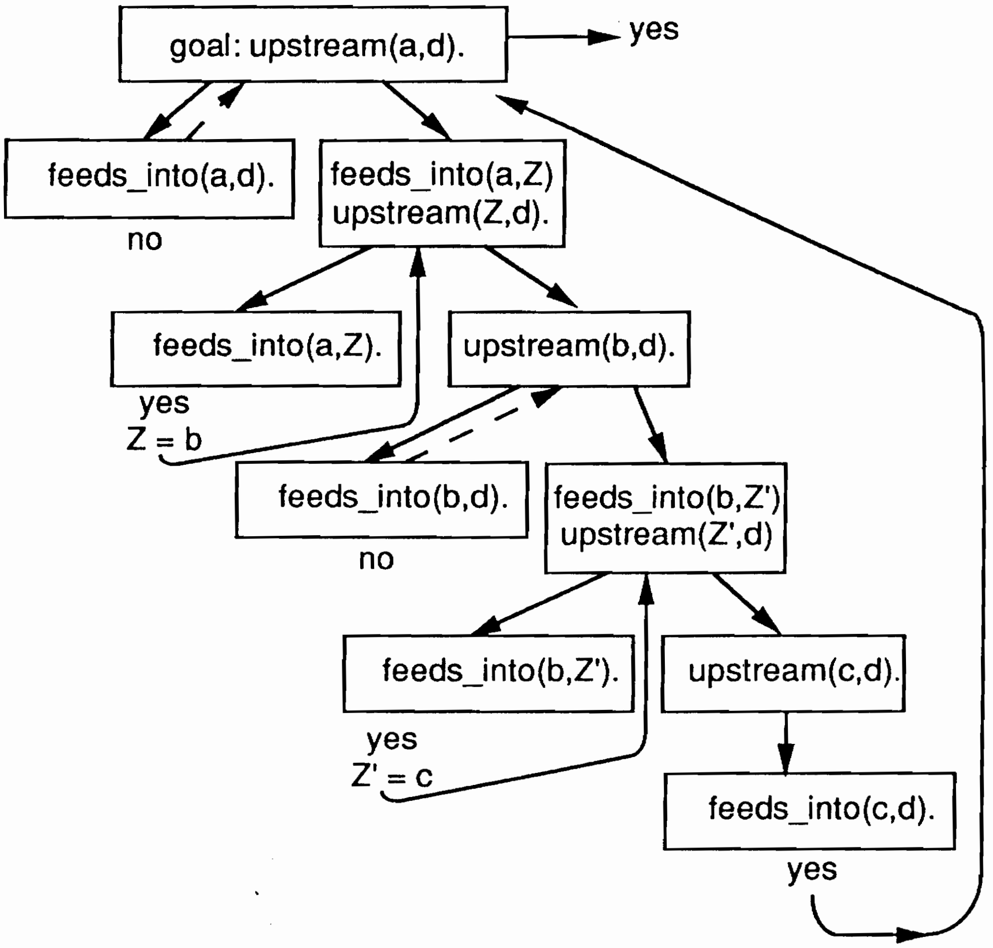


Figure 2.6. The goal upstream(a,d) posed to set one ordering.

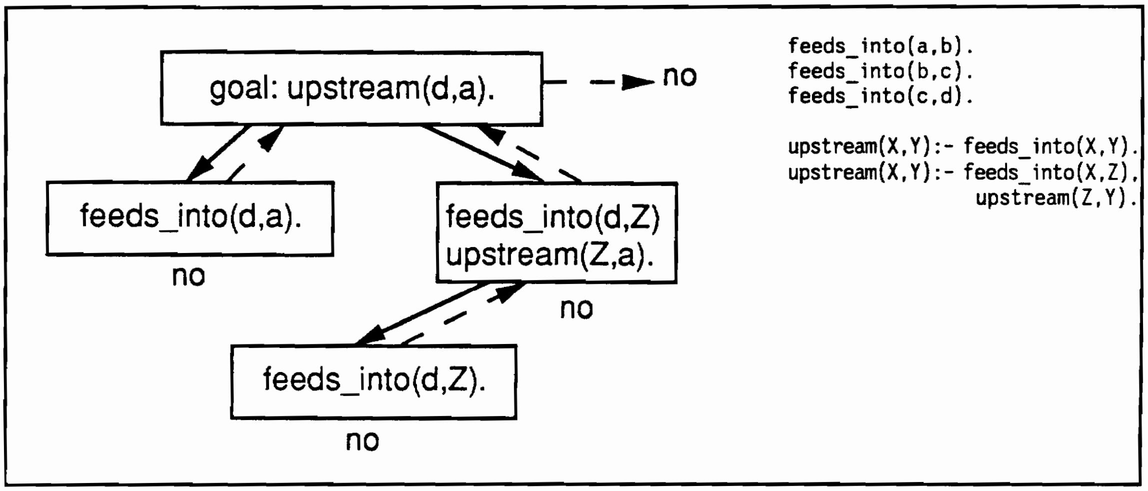


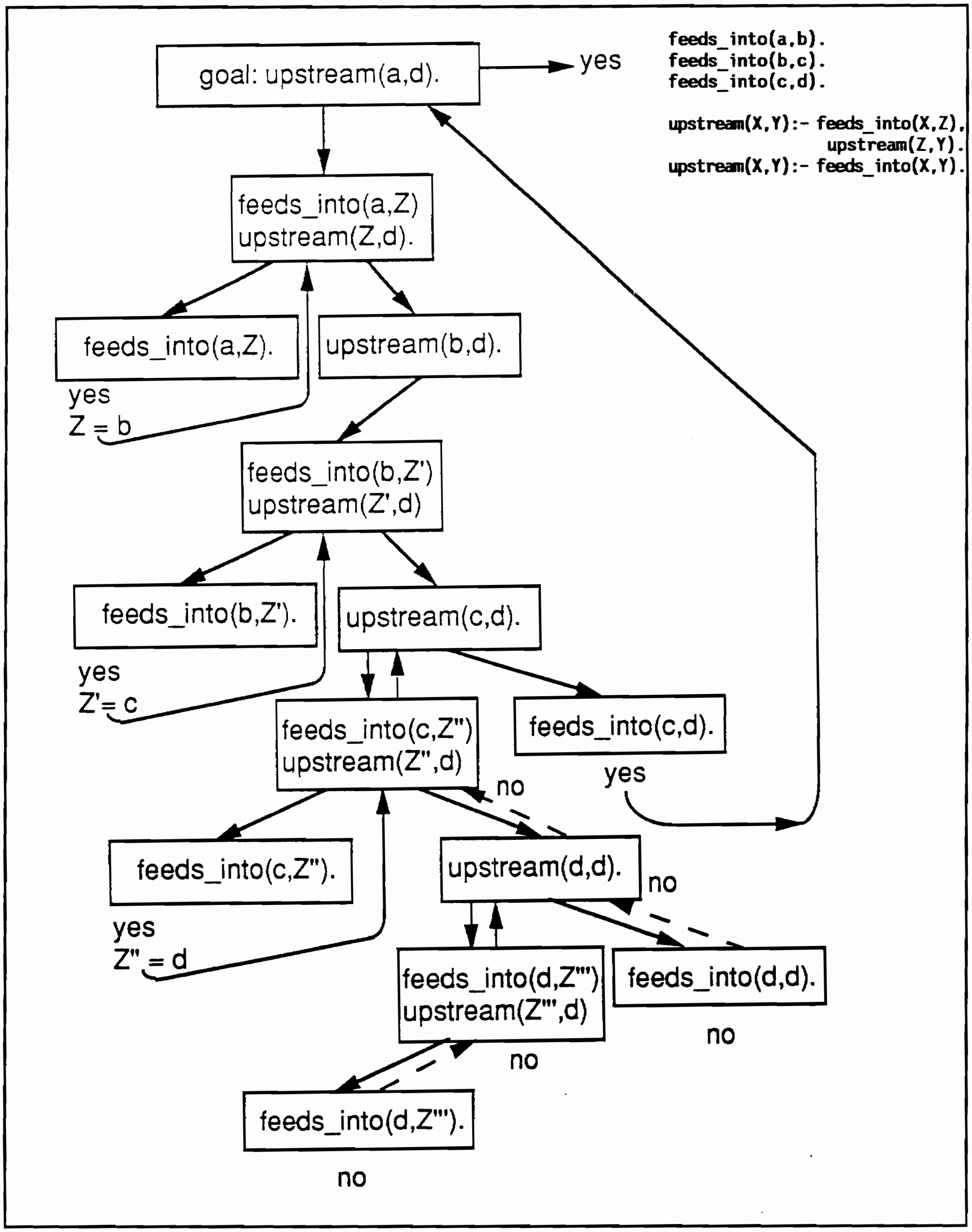
Figure 2.7. The goal upstream(d,a) posed to set one ordering.

Set Two:

```
    upstream(X,Y):-
        feeds_into(X,Z),
        upstream(Z,Y).

    upstream(X,Y):-
        feeds_into(X,Y).
```

Here we switched the two rules. The simpler clause is now second. Note that within each rule, the sub-goal order is maintained. This set is less efficient than the first set, as Figures 2.8 and 2.9 demonstrate. For the question **upstream(a,d)**, Prolog traces out fourteen search blocks (compared to ten in Set One). For the goal **upstream(d,a)**, Prolog traces out four search blocks.

The "bottom line" here is that when more complex rules are called



```
feeds_into(a,b).
feeds_into(b,c).
feeds_into(c,d).

upstream(X,Y):- feeds_into(X,Z),
                upstream(Z,Y).
upstream(X,Y):- feeds_into(X,Y).
```

Figure 2.8. The goal upstream(a,d) posed to set two ordering.

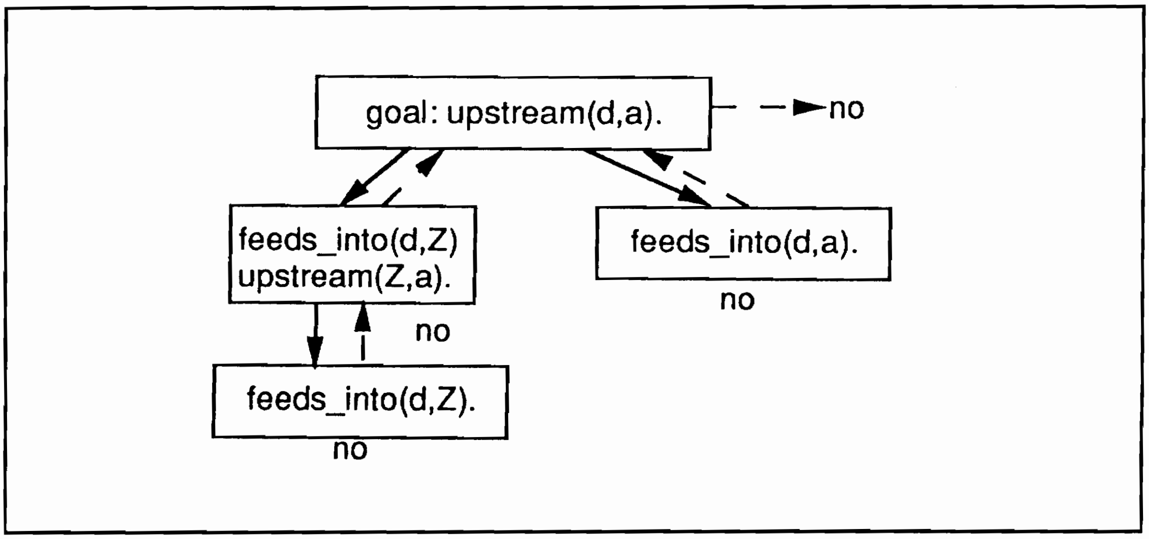


Figure 2.9. The goal upstream(d,a) posed to set two ordering.

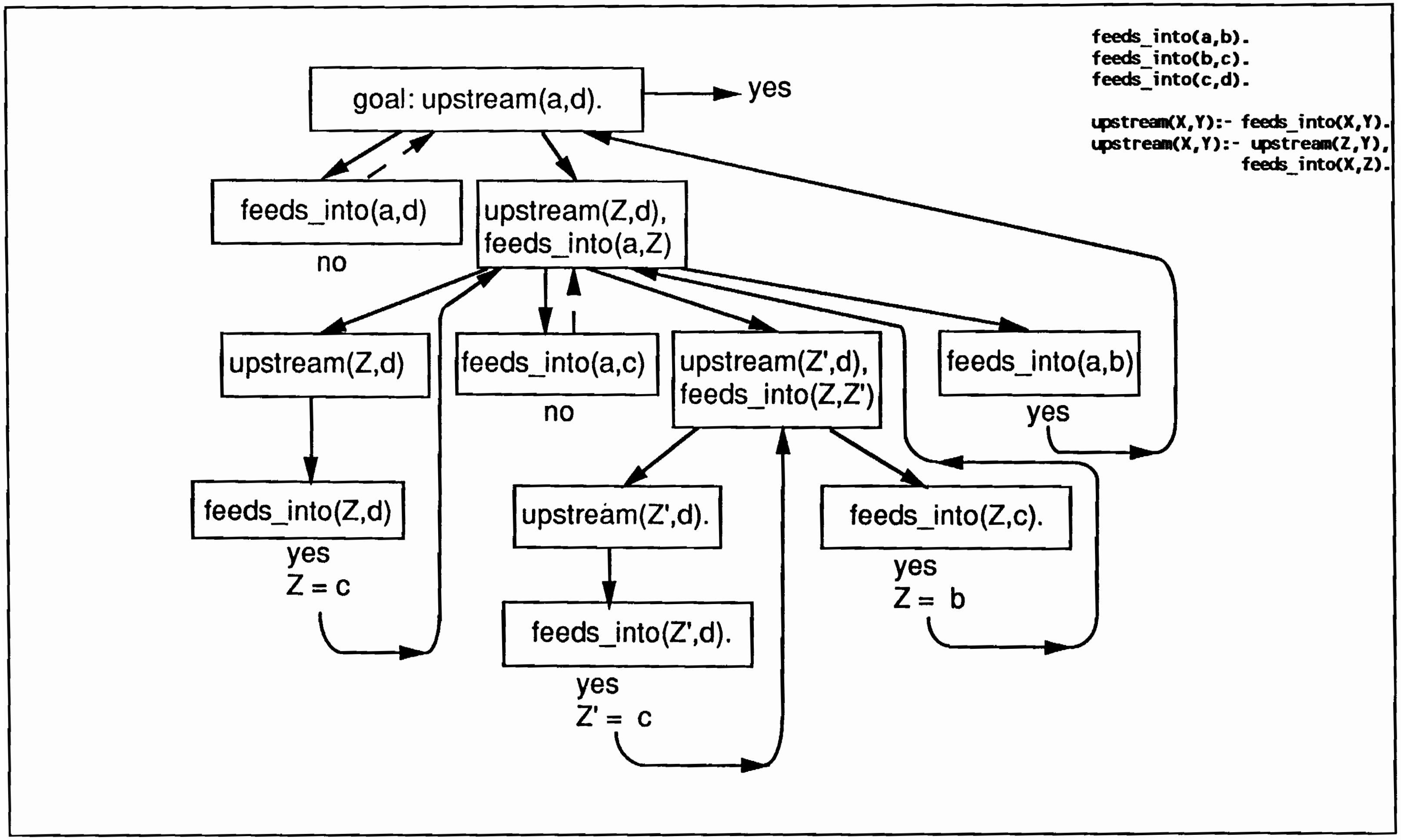
first, more searching is required. Set Two is not as efficient, but still produces the answer.

Now we are going to make the order of the goals more precarious. Consider Set Three below:

Set Three:

```
upstream(X,Y):-
    feeds_into(X,Y).

upstream(X,Y):-
    upstream(Z,Y),
    feeds_into(X,Z).
```

Compared to Set One, Set Three keeps the rule order the same, but switches the sub-goal order in the second clause. Now we get into trouble. As shown in Figures 2.10 and 2.11, the goal **upstream(a,d)** works, but **upstream(d,a)** send the program into infinite loop. This loop results from



```
feeds_into(a,b).
feeds_into(b,c).
feeds_into(c,d).

upstream(X,Y):- feeds_into(X,Y).
upstream(X,Y):- upstream(Z,Y),
                feeds_into(X,Z).
```

Figure 2.10. The goal upstream(a,d) posed to set three ordering.

making the recursive call too early.

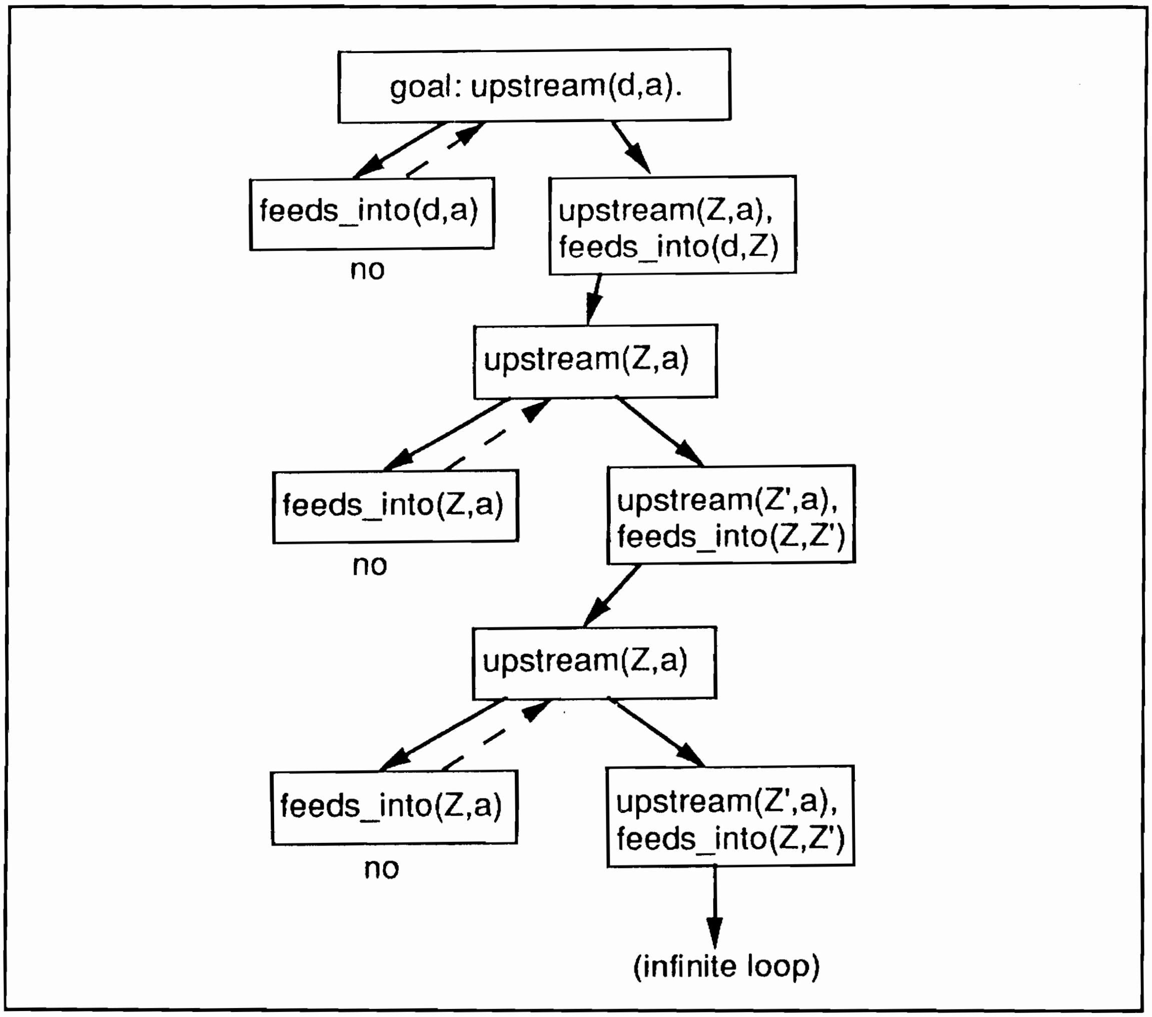


Figure 2.11. The goal upstream(d,a) posed to set three ordering.

Finally, we consider:

Set Four:

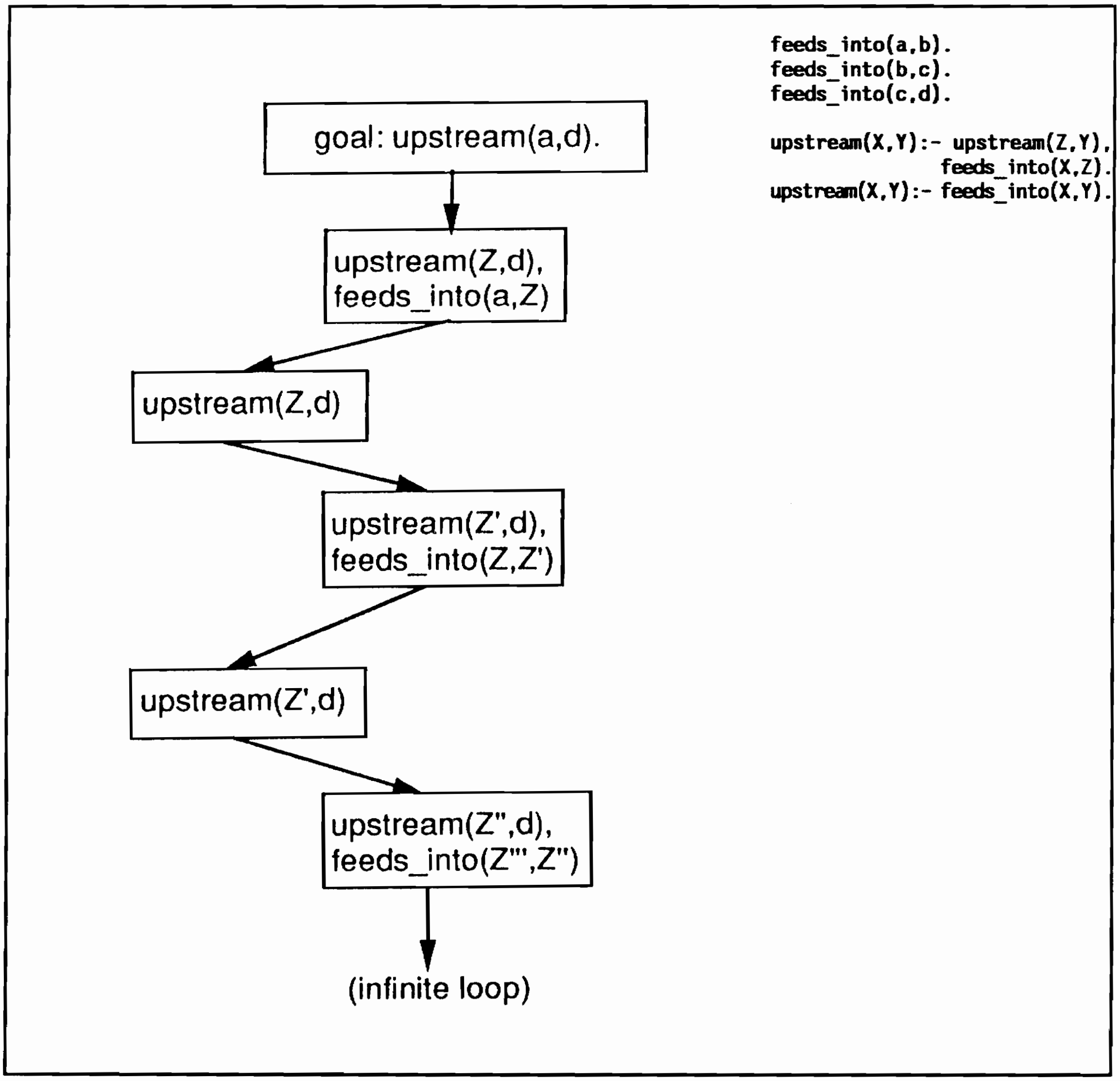
```
    upstream(X,Y):-
        upstream(Z,Y),
        feeds_into(X,Z).

    upstream(X,Y):-
        feeds_into(X,Y).
```

We switch both the rule and sub-goal order in Set Four ordering. The program not only uses the more complex rule first, but within that rule, it has recursion as the first sub-goal. The result? Disaster! *Any* goal, even **upstream(a,d)**, will go into an infinite loop. Figure 2.12 shows the trace for this process.

Since this example about ordering clauses has been long. Let us summarize what we have learned:

- Prolog is a declarative language. However, to write effective Prolog programs, we must pay attention to Prolog's procedural nature. An effective Prolog program must be both:
    - *declaratively accurate*- all relations are properly identified.
    - *procedurally proper*- program enables Prolog to systematically come to problem resolution.

- We can run into serious trouble if we ignore Prolog's procedural


goal: upstream(a,d).
upstream(Z,d), feeds_into(a,Z)
upstream(Z,d)
upstream(Z',d), feeds_into(Z,Z')
upstream(Z',d)
upstream(Z",d), feeds_into(Z'",Z")
(infinite loop)

```
feeds_into(a,b).
feeds_into(b,c).
feeds_into(c,d).

upstream(X,Y):- upstream(Z,Y),
                feeds_into(X,Z).
upstream(X,Y):- feeds_into(X,Y).
```

Figure 2.12. The goal upstream(a,d) posed to set four ordering.

nature when using recursive rules. The most robust recursive rules follow two techniques of rule ordering, namely:

- delay recursive calls to as late as possible; and
- call the "simpler" rules first, such as the matching of facts.

We have introduced four rule orderings, sets one through four, to assess the effects of following these two techniques. Table 2.2 summarize the four orderings.

Table 2.2. Summary of the effects of rule ordering using recursion.

		Obedience to Technique		
Set	Rule Structure	One	Two	Comments
One	upstream(X,Y):- feeds_into(X,Y). upstream(X,Y):- feeds_into(X,Z), upstream(Z,Y).	yes	yes	Most robust; handles all questions quickly and efficiently.
Two	upstream(X,Y):- feeds_into(X,Z), upstream(Z,Y). upstream(X,Y):- feeds_into(X,Y).	yes	no	Not as efficient, but will not go into an infinite loop.
Three	upstream(X,Y):- feeds_into(X,Y). upstream(X,Y):- upstream(Z,Y), feeds_into(X,Z)	no	yes	Will go into an infinite loop if the first rule repeatedly fails.
Four	upstream(X,Y):- upstream(Z,Y), feeds_into(X,Z). upstream(X,Y):- feeds_into(X,Y).	no	no	Disastrous. Will go into an infinite loop every time.

Set One in the **upstream** example follows both techniques of ordering. Consequently, it will properly handle any question and is the most efficient at problem resolution. Set Two delays the recursive call until as late as possible, but fails to call the simpler rule first. It is less efficient, since recursion is unnecessarily forced on the program. However, Set Two will not go into an infinite loop. Set Three violates technique one, i.e., the recursive **upstream** rule immediately calls **upstream**, rather than testing for **feeds_into** first. Consequently, if repeatedly called, this recursive rules will go into an infinite loop. Finally, Set Four is the worst ordering. It violates both techniques. Consequently, it will *always* go into an infinite loop when called.

Ordering of clauses in recursive loops is important if we are to avoid infinite loops. There is a way to classify recursion, and based on that classification, identify how robust the recursion is. We use the classifications of *left*, *center*, and *tail* recursion, and discuss this topic in-depth in section 8.3D.

There are additional ways to ensure a program will search efficiently and avoid infinite loops. We shall discuss these in more depth later in chapter 10 under the topic of *search*.

D. Exercises

2.3.1 Give both the declarative and procedural English translation of the following clause:

```
    P :- Q,R,S,T.
```

2.3.2 Give both the declarative and procedural English translation of the following two clauses:

```
     P :- Q,R,S,T.
     P:- U,V.
```

2.3.3 Consider the following program:

```
     feeds_into(a,b).
     feeds_into(b,c).
     feeds_into(c,d).
     feeds_into(d,e).

     downstream(X,Y):- feeds_into(Y,X).

     downstream(X,Y):- downstream(X,Z),
                       feeds_into(Y,Z).
```

Give the result and the trace of the following questions:

```
     a. ?- downstream(a,b).
     b. ?- downstream(d,a).
     c. ?- downstream(a,c).
     d. ?- downstream(a,X).
     e. ?- feeds_into(b,X),downstream(X,e).
```

Which type of rule ordering (i.e., Set One, Two, Three, or Four) does the program use?

2.3.4 Improve the program in exercise 2.3.3 such that it obeys both techniques of rule ordering. Trace questions a through e using your improved rule ordering. Comment on the efficiency of the new rule ordering versus the ordering in exercise 2.3.3. Which takes more steps to reach the answer?

2.4 CHAPTER SUMMARY

- Data objects in Prolog can be *atoms*, *numbers* (i.e., integers and real numbers), *variables*, or *structured objects* (also known as *structures*).

- Atoms and numbers are *constants*.

- Constants and variables are *simple objects*.

- *Structures* in Prolog are constructed by use of a *functor* with corresponding components. The components can be any Prolog data objects, including atoms, numbers, variables, or other structures.

- *Functors* are related to mathematical functions in that they map objects in a specific way.

- The *name* of the functor must be an atom. The *arity* of a functor is the number of components within it.

- *Unification* is the process of making two terms identical. Prolog implements unification via *matching*, a process to determine if two terms are identical and to *instantiate* any free variables.

- Variables are *instantiated* through the matching process. When two free variables are instantiated to each other they *share* the same data object.

- The term = in Prolog is a *built-in predicate* that follows specific rules in trying to match terms.

- Prolog has a *declarative* and *procedural* nature to it.

- The declarative nature of Prolog means we can develop relations without regard as to *how* Prolog solves the problem. The procedural nature does take into account how and *in what order* Prolog solves the problem.

- In practice, the procedural nature of Prolog must be addressed to prevent infinite loops and ensure a workable, efficient program.

- The careful ordering of sub-goals *within* a clause is one of the simplest and most powerful ways to prevent infinite loops.

A LOOK AHEAD

In the next chapter, we shall look at Prolog's arithmetic operations in more depth. We shall then introduce a very important data structure: the *list*. Lists are an essential part of artificial intelligence programming. Finally, we shall take a more in-depth look at recursion, and assess whether clauses are *deterministic*.

REFERENCES

Bratko, Ivan, *Prolog Programming for Artificial Intelligence*, second edition, pp. 29-66, Addison-Wesley, Reading, MA (1990).

Clocksin, W.F. and C.S. Mellish, *Programming in Prolog,* third edition, pp. 21-34, 49-59, Springer-Verlag, New York, NY (1987).

Ford, N., *Prolog Programming*, pp.3-29 and 59-110, John Wiley & Sons, New York, NY (1989).

Garavaglia, Susan, *Prolog Programming Techniques and Applications*, pp. 70-79, Harper & Row Inc., New York, NY (1987).

Rowe, Neil C., *Artificial Intelligence Through Prolog*, pp. 43-60, Prentice-Hall, Englewood Cliffs, NJ (1988).

Schnupp, P. and L. Bernhard, *Productive Prolog Programming*, pp.31-35, Prentice-Hall, Englewood Cliffs, NJ (1986).

Sterling, L. and Shapiro E., *The Art of Prolog*, pp. 28-30, 51-57, MIT Press, Cambridge, MA (1986).

3

ARITHMETIC, LISTS, AND RECURSION

This chapter introduces more tools for effective Prolog programming. We operators and mathematical predicates. In section 3.2, we investigate lists, which are an important part of Prolog programming. Sections 3.3 and 3.4 expand upon lists, focusing on coupling recursion and lists, and then introducing some common, useful list processing techniques. Following a summary of the chapter in section 3.5, we conclude with several practice problems for applying different list processing techniques to chemical engineering.

3.1 ARITHMETIC IN PROLOG

As mentioned in chapter two, Prolog is a symbolic computing language. Extensive arithmetic abilities are not needed, and therefore arithmetic in Prolog is simple. However, as scientific and engineering applications of Prolog appear, so does the need for numerical computation.

This section summarizes arithmetic in Prolog. The Prolog version under discussion has the capability to process both integers and real numbers. Depending on the version of Prolog that we use, though, some aspects of the discussion below may differ slightly.

A. Arithmetic Operators

1. Operator Notation

In chapter two, we saw that the statement **J = 1** could be expressed in Prolog in the form:

```
= (J, 1).
```

The same is true for other mathematical expressions. We may write

```
X + Y
```

in the following form:

```
+ (X, Y)
```

This pattern is called *prefix notation*, since the + operator is placed before the arguments. The expression

```
(X, Y) +
```

is called *postfix notation*, because the + operator comes after the arguments. Not surprisingly, the original statement X + Y is called *infix notation*, since the operator comes between the arguments.

Although we may not realize it, many of us have used the postfix notation on hand-held calculators. Calculators such as the Hewlett-Packard types using RPN employ the postfix notation. To perform 1 + 2 on an RPN calculator, we type on the keyboard:

```
1
ENTER
2
+
```

and the calculator gives us the result, 3.

Operators, and in particular, infix notation, are used to improve the readability of programs. Consider the statement written in infix notation:

```
2-(3+4)/(2+1)
```

In prefix notation, we write:

```
-(2,/(+(3,4),+(2,1)))
```

The infix notation is obviously easier to understand.

Operators perform according to their priority, also known as *precedence*. For instance,

```
2 + X * Y
```

is assumed to mean 2 + (X * Y). In Prolog, as in most calculators, we can use parenthesis to control priority. To change the priority of the above statement, we may write:

```
(2 + X) * Y
```

which asks us to perform addition, rather than multiplication, first.

Usually, operators in themselves do *not* take any action; i.e., they do not cause arithmetic to be carried out. They simply offer a way of rewriting statements to make them more readable. Thus the statement 1 + 2 written with the + infix operator is simply the Prolog statement:

```
    + (1,2).
```

where + is the functor, and 1 and 2 are its arguments. In fact, in Prolog, the statement 1 + 2 *does not* mean the same as 3. Writing 1 + 2 is no different than writing:

```
    is_a(benzene,chemical)
```

as

```
    benzene is_a chemical
```

Here, the functor **is_a** is an infix operator, similar to the functor +. Likewise, both **benzene** and **chemical** are arguments similar to **1** and **2**.

We can tell Prolog to sum the numbers 1 and 2 to give the result, 3, by using a Prolog command. This will be covered in Section 3.1A3. Before we get there, however, we need to discuss what mathematical operators are available in Prolog. These operators are the topic of Section 3.2. For now, it is important to realize that:

- operators have *notation*, either *prefix*, *infix*, or *postfix*.

- operators act on a priority level.
- the priority level can be adjusted by the use of parenthesis.
- operators in themselves generally take no action. They simply provide a means of rewriting terms to make them more understandable.

2. General Arithmetic Operators

General operators are those found in virtually all versions of Prolog, including versions that do not support real numbers. Table 3.1 summarizes the general arithmetic operators.

Table 3.1. General arithmetic operators.

Operation	**Operator**	**Syntax**
addition	+	X + Y
subtraction	-	X - Y
multiplication	*	X * Y
division	/ or **div**	X / Y or X **div** Y
integer remainder	**mod**	X **mod** Y

In Table 3.1, the **mod** operator stands for *modulo*. When numerically evaluated, this operator gives the integer remainder of the division between two integers. For instance, 9 divided by 5 equals 1, with a remainder of 4. Therefore, the statement

```
9 mod 5
```

when numerically evaluated, gives a value of **4.**

Some versions of Prolog use the **div** operator. The statement **X div Y** represents the division of two integers, while **X / Y** represents the division of real numbers. For example, the statement

```
9 div 4
```

when numerically evaluated, gives the integer **2**, while the statement

```
9 / 4
```

when numerically evaluated, gives the real number **2.25.**

As in procedural languages, in Prolog, confusion can result when a single statement uses both integers and real numbers. Table 3.2 summarizes the numerical evaluation results that most versions of Prolog yield when integers and real numbers appear together.

Table 3.2. Arithmetic computation in Prolog.

OPERATION	SYMBOL	SYNTAX	X	Y	ARITHMETIC RESULT
add	+	X + Y	integer	integer	integer
			integer	real	real
			real	integer	real
			real	real	real
subtract	-	X - Y	integer	integer	integer
			integer	real	real
			real	integer	real
			real	real	real
multiply	*	X * Y	integer	integer	integer
			integer	real	real
			real	integer	real
			real	real	real
divide	/	X / Y	either	either	real
modulo	mod	X mod Y	integer	integer	integer
divide	div	X div Y	integer	integer	integer

Of course, for versions of Prolog that do not support real numbers, all arithmetic computations result in integers.

3. The = and is Predicates

Suppose we ask the following question in an attempt to execute a mathematical calculation:

```
?- X = 1 + 1.
```

Most versions of Prolog respond:

```
X = 1 + 1
```

We would expect Prolog to answer **X = 2**. What happened? The built-in predicate = instantiated the variable **X** to the term **1 + 1**. The = predicate does *not* instruct Prolog to do numerical computation. The statement **1 + 1** is simply a Prolog structure where **+** is a functor, and **1** and **1** are arguments. It is simply an alternate way of writing the statement:

```
+ (1, 1)
```

We utilize infix notation to make the statement more readable.

We can instruct Prolog to carry out numerical computation. To do so, we use the **is** predicate. Thus, if we ask the question:

```
?- X is 1 + 1.
```

Prolog responds as we expect with:

```
X = 2
```

The **is** statement is a built-in predicate that instructs Prolog to numerically evaluate the terms. The query

```
?- X is 5/2.
```

yields ***X* = *2.5***. The query

```
?- X is 5 div 2
```

gives ***X* = *2***; the query

```
?- X = 5/2, Y = 5 div 2
```

leads to the Prolog response:

X* = *5/2*   *Y* = *5 div 2

If we ask the question

```
?- X is 20 mod 7.
```

Prolog responds

X* = *6

4. Mathematical Predicates

With the growth of engineering and physical science applications for Prolog, many versions now include built-in ***mathematical predicates***. These predicates are specific, reserved strings of characters that instruct Prolog to carry out the desired mathematical operation, such as square root or logarithm. We write the square root of X as:

```
sqrt(X)
```

The string **sqrt** is a mathematical predicate. Whenever Prolog sees **sqrt**, Prolog views it as a mathematical square root. When using **sqrt(X)**, we usually require the **is** operator for numerical evaluation. With the question:

```
?- X = sqrt(4).
```

Prolog responds:

```
X = sqrt(4)
```

Prolog views **sqrt** as a functor and **4** as its argument. But with the question:

```
?- X is sqrt(4).
```

Prolog responds:

```
X = 2
```

Prolog has many mathematical predicates. Table 3.3 summarizes the typical built-in mathematical predicates supported by most advanced versions of Prolog.

Table 3.3. Built-in mathematical predicates.

PREDICATE	FUNCTION
abs(X)	Absolute value of X
arccos(X)	Arccosine of X (in radians)
arcsin(X)	Arcsine of X (in radians)
arctan(X)	Arctangent of X (in radians)
cos(X)	Cosine of X (in radians)
exp(x)	The value *e* raised to the power of X
ln(X)	Natural logarithm of X
log(X)	Base 10 logarithm of X
sin(X)	Sine of X (in radians)
sqrt(X)	Square Root of X
tan(X)	Tangent of X (in radians)

Finally, some versions of Prolog also support more general exponential expressions. The format of these is usually **X**Y**, where **X** and **Y** are real numbers. When coupled with the **is** operator, i.e., **Z is X**Y**, Prolog performs the numerical computation and the result (instantiated to Z) is a real number.

B. Arithmetic Comparison Operators

Arithmetic comparison operators are *relational* operators. They compare two Prolog objects that are represented by numbers. These Prolog objects can be constants, e.g., integers and real numbers such as 11 and 3.141596. In addition, the objects can be variables instantiated to numbers, e.g., X and Y. For example, the statement

```
?- X > 1.5.
```

uses the arithmetic comparison > with the variable **X** and the constant (real number) **1.5**. It asks, "Is the number represented by the variable X greater than 1.5 ?"

Previously, we used the **=** operator as a comparison operator. For instance, the statement

```
?- X = 1.5.
```

succeeds if the two terms **X** and **1.5** match (i.e., **X** is instantiated to

1.5). If the terms do not match, the statement fails. It asks, "Does the number represented by variable X match the number 1.5 ?"

Prolog includes additional mathematical comparison operators. They are listed in Table 3.4.

Table 3.4. Mathematical comparison operators.

Operation	Operator	Syntax
greater than	>	X > Y
less than	<	X < Y
greater than or equal to	>=	X >= Y
less than or equal to	=<	X =< Y
values are equal	=:=	X =:= Y
values are not equal	=\=	X =\= Y

These comparison operators will *not* perform instantiation. Both **X** and **Y** *must* be either constants or variables instantiated to numerical values prior to using these operators.

Note also that "less than or equal to" is *not* written as <=, as seen in some conventional programming languages. Instead, we write it as =<. The notation <= looks like an arrow, which is a common notation in logic; the statement **X** <= **Y** is read in logic as "Y implies X." Logic programmers like to use the notation <= as an atom, and Prolog accommodates that representation by denoting "less than or equal to" as =<.

A subtle aspect of Prolog is the difference among the =, is, and

=:= operators. In the statement

```
X = Y
```

the = predicate instructs Prolog to match **X** and **Y**, instantiating any variables if necessary. Neither **X** nor **Y** needs to be instantiated at the time of execution. Importantly, as discussed previously, the = predicate does not perform numerical evaluation.

In the statement

```
X is Y
```

the **is** predicate instructs Prolog to perform numerical evaluation *and* matching. The variable **Y** on the right-hand side *must* be instantiated to an expression that can be evaluated numerically. When the evaluation is done, the numerical result is matched with **X**. Instantiation of **X** will be performed if needed to complete the match. For example, we write the question:

```
?- Y = sqrt(4), X is Y.
```

This says in English, "match Y with the square root of four, numerically evaluate the square root of four, and instantiate X to the numerical result." Prolog responds:

X = 2 Y = 2

Questions that succeed with numerical constants will not succeed with nonnumerical constants when we use the is predicate. We ask the question, "match Y with the structure **is_a(benzene,chemical)**, attempt to numerically evaluate the structure, and instantiate the numerical result to X," we would write in Prolog:

```
?- Y = is_a(benzene,chemical), X is Y.
```

But depending on the version of Prolog we use, this statement results in an error message or the answer ***no***. The reason? The is predicate is specifically designed for numerical computation. The first statement, Y = **is_a(benzene,chemical)** instantiates the variable Y to the fact **is_a(benzene,chemical)**. The next statement, **X** is Y, effectively reads

```
X is is_a(benzene,chemical)
```

Since the is_a fact cannot be numerically evaluated by the is predicate, Prolog responds with either ***no*** or an error message.

Finally, in the statement

```
X =:= Y
```

the =:= predicate instructs Prolog to perform numerical evaluation and comparison only. No instantiation of variables is done. Therefore, X and Y must *both* be instantiated at the time of execution.

Table 3.5 contrasts the =, is, and =:= operators.

Table 3.5. The =, is, and =:= operators.

DESIRED ACTION	VARIABLE STATUS	ALLOWED IN PROLOG WITH OPERATOR		
		X = Y	X is Y	X =:= Y
comparison	X bound Y bound	yes	yes	yes
instantiation	X free Y bound	yes	yes	no
	X bound Y free	yes	yes	no
	X free Y free	yes	no	no
numerical evaluation	X any Y any	no	yes	yes

Using the prohibited operations, indicated by "no" in the table, can either cause a *fail* in the built-in predicate, or an error message, depending on version of Prolog we are using.

As examples, let us consider the following questions and answers:

```
?- 3 + 4 = 4 + 3

no

?- 3 + 4 is 4 + 3

yes

?- 3 + 4 =:= 4 + 3
```

```
yes
?- X = 3 + 4
X = 3 + 4
?- X is 3 + 4
X = 7
?- 3 + X = Y + 4
X = 4
Y = 3
```

We need to be careful when comparing real numbers with **is** or **=:=**. If one number is 1.00001, while the other is 1.00000, the test may fail. In these situations, it is best to test if the variables are within a certain range of each other. For instance, instead of using the statement

```
X is 1.00001
```

we may use the statement

```
X < 1.001, X > 0.999
```

to avoid a spurious fail due to round-off error.

To summarize this section, we see that:

- *Operators* can be used to improve the readability of programs. They generally take no action.
- There are *prefix*, *infix*, and *postfix* operators, named according to their location.
- The arithmetic operators are +, -, *, /, and **mod**. Some versions of Prolog also use the **div** operator for integer division.
- Prolog uses built-in predicates to numerically evaluate arithmetic expressions. Most versions **is** and **=:=** are used.
- The Prolog equality operator **=** does *not* do arithmetic evaluation. To do arithmetic evaluation, we use the **is** operator.

C. Exercises

3.1.1 How will Prolog respond to the following questions?

```
a. ?- X is 2, X = 1 + 1.
b. ?- X is 2, X =:= 1 + 1.
c. ?- X is 10 mod 3, X =\= 2.
d. ?- X + 3 = 4 + Y.
e. ?- X + 3 is 4 + Y.
```

3.1.2 Define the relation **gcd** (greatest common divisor) such that **gcd(X,Y,Z)** is true if, given X and Y, Z is the greatest common divisor between them. For example, the following dialogue results from the proper

gcd relation:

```
?- gcd(10,15,X).
X = 5

?- gcd(1080,1220,X)
X = 20
```

3.1.3 Define the relation **lcm** (least common multiple) such that **lcm(X,Y,Z)** is true if, given X and Y, Z is the least common multiple between them. For example, the following dialogue results from the proper **lcm** relation:

```
?- lcm(5,3,X).
X = 15

?- lcm(24,36,X).
X = 72
```

3.1.4 Define the relations **max** and **min**. The predicate **max(X,Y,Z)** is true if Z is the greater of the two numbers, X and Y. Likewise, **min(X,Y,Z)** is true if Z is the lesser of the two numbers X and Y.

3.2 LISTS

A. Description and Syntax

Lists are another important type of Prolog data structure. Lists have been used extensively in artificial intelligence, in languages such as *LISP*.

A list is an ordered sequence of elements of any length. The following is an example of a list of hydrocarbons:

```
[methane, ethane, propane, butane]
```

Square brackets: [and] denote a list, and commas separate each element within the list. The above list has four elements. If a list has no elements, we call it an *empty* or *nil* list, and represent it in Prolog as simply the opening bracket followed by the closing bracket with no element in-between:

```
[ ]
```

The elements in a list can be atoms, numbers (integers and real numbers), variables, or structures. We may even have a list of lists. The following are some examples of acceptable Prolog lists:

```
[a, 12, X]                          % [ atom, integer, variable ]
```

```
[X, Y]                                          % [ variable, variable]
[pump, feeds_into, [X, Y]]                      % [ atom, atom, list ]
[[station a 20], [station b 15], [station c 35]] % [ list, list,
                                                 %   list]
```

Prolog lists are similar in some ways to *arrays* in a conventional language such as FORTRAN. We can view an array as an ordered data structure. The array S(N) with N = 3 has the following terms: S(1), S(2), and S(3). In a sense, this array represents a "list" of memory cells with the same name where each cell stores information.

There are, however, substantial differences between lists and arrays. First, the array S(N) represents a subscripted variable, i.e., S_N. A list in Prolog is a collection of data objects (that are usually related). In addition, an array in FORTRAN is a *static* data structure. Once the array receives a dimension, it can *never* change. For instance, the statement

```
DIMENSION S(3)
```

in FORTRAN "locks" the variable S into possessing three subscripted variables: S(1), S(2), and S(3). If we want to use the variable S(4) somewhere in the program, we are out of luck. The maximum "length" is three.

A list, however, is a *dynamic* data structure in Prolog. A list *can* change its length during the program execution. Prolog's ability to use lists with variable lengths (and elements) is one of its advantages over FORTRAN. We discuss how to dynamically change lists in the next section.

B. Head and tail of a list

To effectively process data structures using lists, we need to be able to "tear down" and "build up" a list. To do this, we consider a list in Prolog as containing two elements:

(1) the *head* of a list, i.e., the first element, and

(2) the *tail* of a list, i.e., the remaining part, which *must* be another list.

Consider again the list of hydrocarbons:

[methane, ethane, propane, butane]

Here, the head of the list is **methane**, while the tail is another list: **[ethane, propane, butane]**. Now we consider:

[[station a 20], [station b 15], [station c 35]]

The head is the list **[station a 20]**, while the tail is the list of lists

[[station b 15], [station c 35]].

It is important to realize that the head of a list can be any Prolog object (atom, number, variable, structure, or list). The tail, though, *must* be another list. For example, let us consider the list:

```
[ 1, 2 ]
```

The tail is the list **[2]**, and *not* the atom **2**. If we forget this point and later try to match the tail with the integer **2**, the match will fail since the number **2** and the list **[2]** are not matching objects.

Prolog provides a way to write lists to make them easier to process. Most versions of Prolog use a vertical bar **|**, to separate the head and tail. We write the list of hydrocarbons as:

```
[methane | [ethane, propane, butane]]
```

We can use the vertical bar in a more general sense too. For example, the following lists:

```
[methane | [ethane, propane, butane]]
[methane, ethane | [propane, butane]]
[methane, ethane, propane | [butane]]
[methane, ethane, propane, butane | []]
```

all represent the same data structure. The head and tail of each, however, are *different* data structures.

Let us look at a list with a single element:

```
[ methane ]
```

The head of the list is the first element, in this case the atom **methane**. But what is the tail? We know the tail must be another list. When a list has only a single element, the tail is the *empty* list, [ ]. We write the list [**methane**] using vertical bar notation as:

```
[ methane|[ ] ]
```

Use of the head and tail enables us to dynamically change lists while Prolog is executing. Consider the following question:

```
?- X = a, Y = [b,c,d], Z = [X,X,X | Y].
```

Prolog responds:

```
X = a
Y = [b,c,d]
Z = [a,a,a,b,c,d]
```

We have just "built up" a list, in this case, Z, to six elements. Now to "tear down" a list, we pose the following question:

```
?- K = [a,b,c,d], K = [L,M | N].
```

Prolog responds:

```
K = [a,b,c,d]
L = a
M = b
N = [c,d]
```

Here, we have partially "torn down" a list: K has four elements, while N only has two. And finally, we consider:

```
?- K = [a,b,c,d], K = [L,L | M]
```

Prolog answers ***no***. Why? In the instantiation process, Prolog instantiates M to the list [c,d]. It attempts to instantiate L to the atom a *and* atom to the b. Since these atoms do not match, the goal fails, and Prolog answers ***no***.

C. Lists as binary trees

It can be helpful to represent lists as binary trees using the *dot operator*, . . This operator is a functor whose components are the head and the tail of a list. The following lists are all equivalent:

```
[methane, ethane, propane, butane]
.(methane, [ethane, propane, butane])
.(methane, .(ethane, [propane, butane]))
.(methane, .(ethane, .(propane, [butane])))
.(methane, .(ethane, .(propane, .(butane,[]))))
```

Inside the language, Prolog stores lists in the complete dot-operator form. Thus, the above list **[methane, ethane, propane, butane]** is stored in Prolog as **.(methane, .(ethane, .(propane, .(butane, nil))))**, where **nil** is the empty list **[ ]**.

As the above examples indicate, lists written with square brackets are much easier to read than the dot-operator notation. Consequently, although Prolog stores the list inside in the dot-operator form, the square-bracket notation is used when we do actual Prolog programming.

Using the dot operator does give us some advantages. It allows us to graphically display the list as a binary tree, as shown in Figure 3.1.

The list **[[methane,ethane],[propane,butane]]** is shown in Figure 3.2. In complete dot-operator form, the list is:

```
.(.(methane, .(ethane, nil)),.(propane, .(butane, nil)))
```

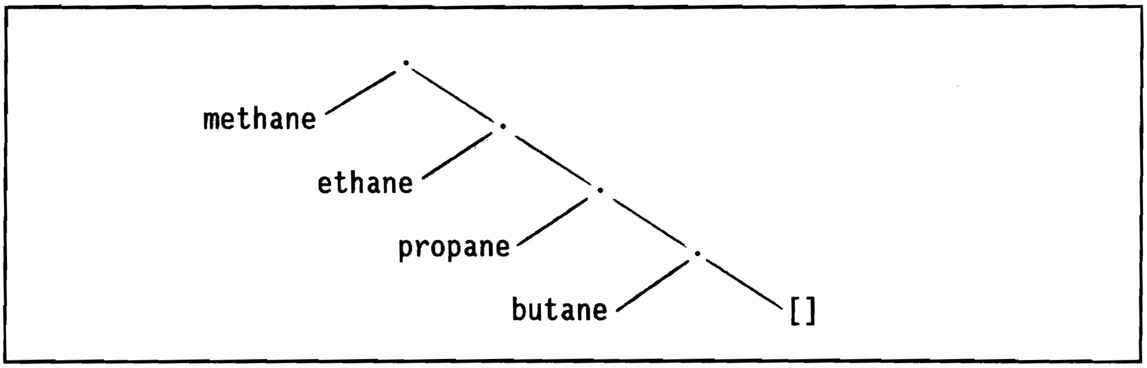


Figure 3.1. Binary-tree form of [methane, ethane, propane, butane].

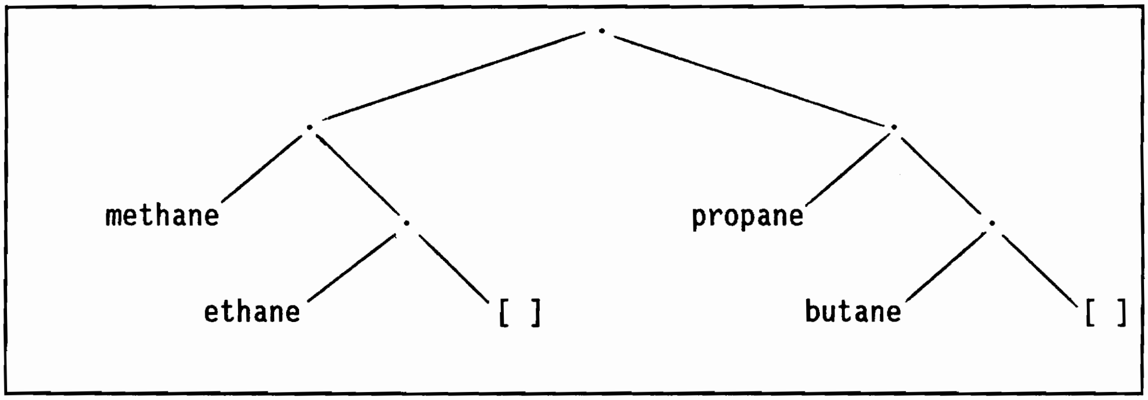


Figure 3.2. Binary-tree form of [[methane, ethane], [propane, butane]].

We can use the dot operator to represent pumping networks, process flowsheets, cause-and-effect analyses, and many other scientific and engineering problems as binary trees. Generally, we use the trees to analyze the relations and develop Prolog rules. For writing the actual Prolog program using lists, however, we prefer the square-bracket notation.

D. Exercises

3.2.1 Consider the following Prolog program:

```
material(hydrocarbon, [ethane, ethylene, propane, propylene]).
material(olefin, [ethylene, propylene, vinyl_acetate]).
material(aldehyde, [formaldehyde, acetaldehyde]).
material(ketone, [acetone, methyl_ethyl_ketone]).
```

How will Prolog respond to the following questions?

a. `?- material(hydrocarbon,[H|T]).`

b. `?- material(X,[E1,E2,E3|T].`

c. `?- material(_,[X,Y]).`

d. `?- material(_,[X|Y]).`

3.2.2 Write the following lists in dot operator form and draw the corresponding binary tree.

a. `[methane, ethane, propane, isobutane, n_butane, neopentane]`

b. `[ [methane, ethane, propane], isobutane, n_butane, neopentane]`

c. `[ [methane, ethane], propane, [isobutane, n_butane], neopentane]`

d. `[ [methane, ethane], [propane, isobutane], [n_butane,neopentane]]`

3.3 RECURSION AND LIST PROCESSING IN PROLOG

A. Recursion Using Lists

Recursion and list processing go hand-in-hand in Prolog. Combining these two aspects gives us a great deal of programming power and flexibility. Consider an "environmental" Prolog program tied into a water analyzer monitoring chemicals in a process stream. The Prolog program keeps an updated list of currently detectable chemicals. Is benzene in the process stream? To determine the answer using Prolog, we need to see if benzene is a member of the list of chemicals. We can use the predicate **member** to determine if a certain data object is a list element.

We first need to write the rules for the **member(X,L)** relation, where **X** is an object and **L** is a list. The **member(X,L)** statement succeeds when **X** is a member of **L** and fails otherwise. One difficulty is that our list of detectable chemicals can have any number of elements in it. In addition, we do not know which element of the list is benzene. One possible way to define **member(X,L)** in Prolog is:

```
member(X,[E1]):-                % rule set 1- handles a list with
          X = E1.               % one element, E1

member(X,[E1,E2]):-
          X = E1.               % rule set 2- handles a list with
```

```
member(X,[E1,E2]):-              % two elements, E1 and E2
            X = E2.


member(X,[E1,E2,E3]):-
            X = E1.
member(X,[E1,E2,E3]):-           % rule set 3- handles a list with
            X = E2.              % three elements, E1, E2, and E3
member(X,[E1,E2,E3]):-
            X = E3.


member(X,[E1,E2,E3,E4]):-
            X = E1.
member(X,[E1,E2,E3,E4]):-        % rule set 4- handles a list with
            X = E2.              % four elements, E1, E2, E3, and E4
member(X,[E1,E2,E3,E4]):-
            X = E3.
member(X,[E1,E2,E3,E4]):
            X = E4.
```

This set of rules works if benzene is in a list with no more than four elements. However, the program is lengthy. Also, what if we had 40 chemicals detected in the process stream? We would be in trouble-- we would need forty rule sets. Writing the program this way is obviously inconvenient.

There is a more effective way to implement the **member(X,L)** relation in Prolog. We use recursion on the list. We define the **member** relation as:

```
member(X,[X|_]).
member(X,[_|Tail]):-
            member(X,Tail).
```

The first clause is a fact. In English, it says: "X is a member of a list if X is the head of the list". The second clause is a rule. It says "X is a member of a list if X is a member of the tail of the list". Also note the use of the underscore, which is the anonymous variable in the relations. In the first clause, there is no point in naming the tail of the list with a variable, because the tail is not used in the clause. In the second clause, the head does not need to be named, since it is not used.

Now assume that our Prolog water-monitor currently has the list **L** instantiated to **[ethanol,toluene,acetone,benzene]**, and we ask:

```
?- member(benzene,L).
```

Prolog responds ***yes***. The trace of this recursive search is shown in Figure 3.3. We discuss the trace step-by-step below.

Step 1: With **L** instantiated to **[ethanol,toluene,acetone,benzene]**, posing

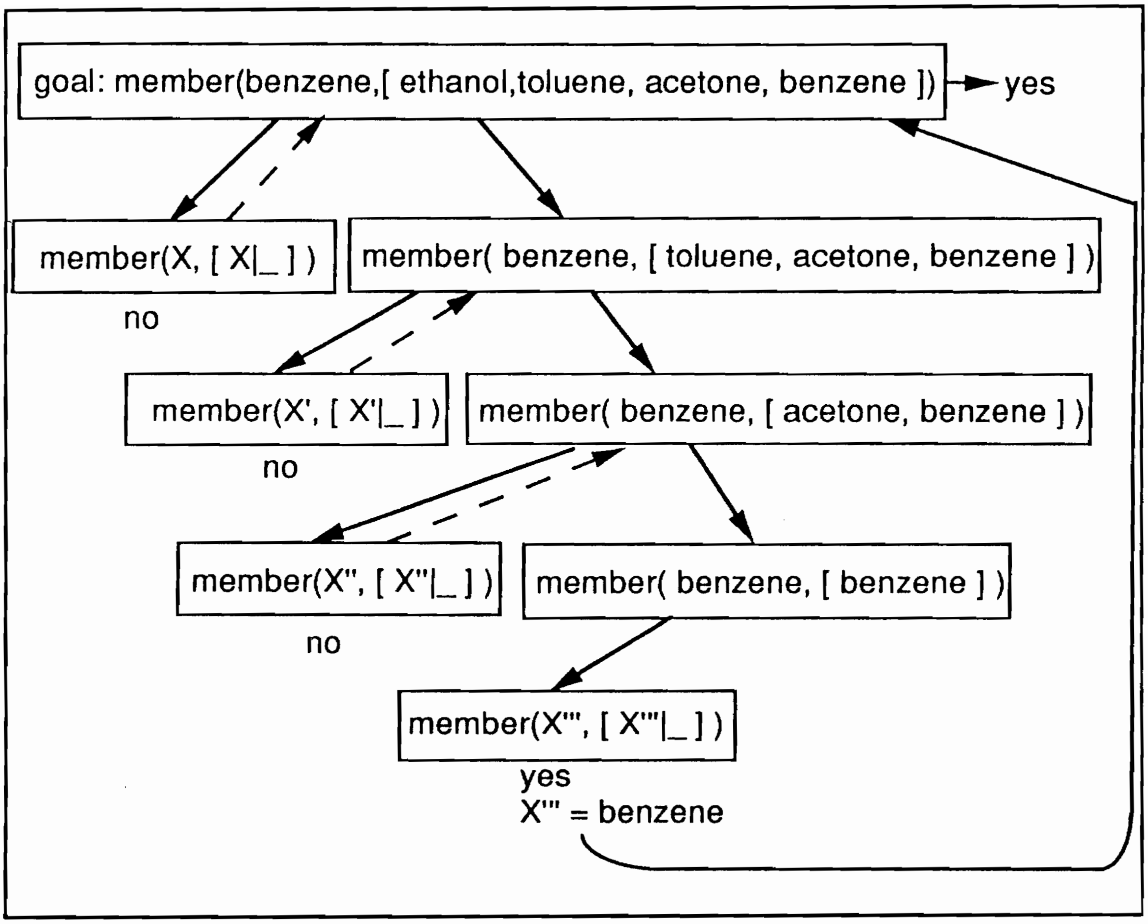

goal: member(benzene,[ethanol,toluene, acetone, benzene])
yes
member(X, [X|_])
no
member(benzene, [toluene, acetone, benzene])
member(X', [X'|_])
no
member(benzene, [acetone, benzene])
member(X", [X"|_])
no
member(benzene, [benzene])
member(X"', [X"'|_])
yes
X"' = benzene

Figure 3.3. Trace of member(benzene,[ethanol, toluene, acetone, benzene]).

the question creates the initial goal, **member(benzene, [ethanol, toluene, acetone, benzene]).**

Step 2: Prolog attempts to match this goal with the first **member** clause in the database, **member(X,[X|_]).** The match fails, since **benzene** does not match **ethanol**. Prolog backtracks to the next **member** clause in the database in an attempt to satisfy the goal.

Step 3: The next **member** clause is **member(X,[_|Tail]):- member(X,Tail).** Prolog matches the head of this clause with the goal, and instantiates **X** to **benzene** and **Tail** to **[toluene,acetone,benzene]**. Prolog then makes the recursive call **member(benzene,[toluene,acetone,benzene]).**

Step 4: We now have a new goal, **member(benzene,[toluene,acetone,benzene])**, so Prolog starts at the top of the database and searches downward for a match. We first hit the **member** fact again, i.e., **member(X',[X'|_])**. We place the prime on the **X** to remember that this variable is different from the previous variable **X**. The match fails, since **benzene** and **toluene** do not match. Prolog backtracks to the next **member** clause.

Step 5: Entering the **member** rule, Prolog matches **X'** with **benzene** and **Tail'** with **[acetone,benzene]**. Prolog then makes the recursive call, **member(benzene, [acetone, benzene])**.

We see recursion at work. Prolog is making progress towards its goal by systematically shortening list **L**. List **L** is down to two elements, and Prolog has not yet found a match. We continue the search.

Step 6: Again we have an entirely new goal, **member(benzene, [acetone, benzene])**. Prolog searches form the top of the database, and the first **member** relation, **member(X'',[X''|_])**, again fails. The atom **benzene** does not match the atom **acetone**. Prolog backtracks to the next **member** rule.

Step 8: Entering the **member** rule, Prolog matches **X''** with **benzene** and **Tail''** with **[benzene]**. Prolog then makes the recursive call, **member(benzene, [benzene])**.

Step 9: We have a new goal, **member(benzene,[benzene])**. Prolog starts at the top of the database, and first attempts to match with the fact **member(X''',[X'''|])**. We have a match! Atom **benzene** matches atom **benzene**. Prolog instantiates **X'''** to **benzene**, and reports the result, ***yes***.

Classifying clauses is useful when using recursive rules. Two types of clauses exist when using recursion: the *base case*, and the *recursive case* (sometimes called the *general case*). The base case is a matching fact that *does not* involve recursion. The recursive case is a rule that *does* involve recursion:

BASE CASE	→	FACT	→	DOES NOT INVOLVE RECURSION
RECURSIVE CASE	→	RULE	→	DOES INVOLVE RECURSION

Let us look at the member relation again:

```
member(X,[X|_]).          % base case: X is a member of a list if
                          % X is the head of the list
member(X,[_|T]):-         % recursive case: X is a member of a list
        member(X,T).      % if X is a member of the tail of the list
```

The base case is essential to program success. It breaks the recursive loop. If our program lacks the base case, the **member** predicate will continue to recursively call the **member** predicate, and an infinite loop will result. The presence of the base case, then, avoids an infinite loop. When a recursive **member** call matches the base case, an answer is successfully located. No more sub-goals need to be fulfilled; none are called for since the base case is a fact. The match of the base case closes the search through the relation, and the loop is broken.

The recursive case is also essential for program success. It is used when the base case fails. The recursive case does not finalize an answer, but it does move us one step closer to the answer. The recursive case in the **member** relation says, in effect, *we know X is not the head of the list because we just tested it in the previous fact (i.e., the base case). Therefore, we chop off the head of the list and search for X in the tail of the list.*

This action of "chopping off the head" moves us one step closer to the answer. We look at the recursive calls in Figure 3.3 with the goal

```
member(benzene,[ethanol,toluene,acetone,benzene])
```

and we see the following result from chopping off the head of the list:

```
member(benzene,[toluene,acetone,benzene])
member(benzene,[acetone,benzene])
member(benzene,[benzene])
```

Each time, the recursive case shortens the list until the final call,

```
member(benzene,[benzene])
```

matches the base case. Matching the base case breaks the loop.

Suppose we ask the question

```
?- member(propane,[ethanol,toluene,acetone,benzene]).
```

By inspection, we know this goal will fail, since **propane** is not a member of the list. In an attempt to satisfy the goal, however, Prolog:

(1) tests each sub-goal to see if it matches the base case, and finds no matches; and

(2) continues to "chop off the head of the list" and make recursive calls, hoping to move us one step closer to the answer.

The following goals arise from the recursive case:

```
member(propane,[toluene,acetone,benzene])

member(propane,[acetone,benzene])

member(propane,[benzene])

member(propane,[])
```

The final goal **member(propane,[])** contains the empty list as the second argument. Because the empty list cannot be split into a head and a tail, this goal does not match either **member** clause, and Prolog answers

no

To summarize, all recursion needs:

(1) A *base case*, which is usually a Prolog fact, to break the recursive loop.

(2) A *recursive case* to apply when the base case fails. The recursive case propagates a loop by calling a shortened alternative to the relation from where it originated (e.g., the goal **member(p,[e,t,a,b])** calls **member(p,[t,a,b])** as a sub-goal). Hopefully, the recursive call leads us one step closer to the answer.

If both the base and recursive cases fail, then the entire rule fails.

B. Comparison with Iteration

Most scientists and engineers are familiar with an *iterative* procedural language such as FORTRAN that uses the DO loop. Consider the following DO loop in FORTRAN:

```
          DO 100 I= 1, 5
          N = I + 1
          PRINT I, N
      100 CONTINUE
```

The variables I and N represent store locations in memory. The first time through the loop, the store location for I has a value of 1, while N is 2. The next time through the loop, 2 *replaces* 1 in the store location for I. Likewise the value of 3 *replaces* N. The number of store locations remains at a constant value of two regardless of how many times we go through the loop.

Recursion is different. Each step Prolog takes in a recursive loop is *remembered*. Prolog adds each step onto the stack memory of the computer. The more we go through the loop, the more steps are stored on the stack. Therefore, compared to iteration, recursion utilizes *more memory space*. This extra space can become an obstacle to implementing large Prolog programs. The computer must have a lot of available memory space.

Recursion thus seems less advantageous than iteration. Why do we want to use up more memory space? Recursion seems costly, possibly wasteful. What does it buy us? The answer is simple. Because Prolog remembers each step, it has the ability to *backtrack*. This ability allows us to solve more complex problems (such as those in artificial intelligence) requiring search. If Prolog fails to find the answer in a certain area of a search, it can simply backtrack until it finds the answer. Iterative languages such as FORTRAN do *not* have this ability.

3.4 PRACTICAL APPLICATIONS OF LIST PROCESSING

We are now at the point where we can begin practical Prolog programming. Since lists are such an important part of Prolog, we are devoting this section to list processing. We have already seen one list operation, assessing whether an element is a member of a list. In this section, we introduce some additional, frequently used, list operations. These list operations are *not* built into Prolog. Instead, we must define them in the program ourselves. After introducing these list relations, we summarize them in Table 3.7.

A. Append

The **append** predicate is used to join two lists together to form a single list. We append **List1** and **List2** to give **List3**, and write:

```
append(List1,List2,List3)
```

For example,

```
append( [a,b,c], [3,1], [a,b,c,3,1])
```

is true; while

```
append([a,b,c],[d,e,f],[a,b,c,a,d,e,f])
```

is false. The *base case* for **append** is when the **List1** is an empty list, [ ]. When we append [ ] with **List2**, the result is **List2**. We write this fact as:

```
append([],L,L).
```

This says in English, "the list L appended with an empty list gives the same list L."

The *recursive case* for **append** occurs when **List1** is not empty. In that case, we "tear down" **List1** using its head and tail. In addition, we "build up" **List3** using the head *from* **List1**. We write the recursive case:

```
append([H1|T1],L2,[H1|T3]):-
     append(T1,L2,T3).
```

To append lists L1 and L2 to produce list, L3, we "take the head of L1, make it the head of the list L3, and then append the tail of list L1 with the entire list L2 to give the tail of the resulting list, L3." The complete append relation is defined by two clauses:

```
append([],L,L).                     % base case
append([H1|T1],L2,[H1|T3]):-        % recursive case
```

```
            append(T1,L2,T3).
```

For example, let us ask the question:

```
?- append( [a,b,c], [d,e,f], L ).
```

Prolog responds:

```
L = [a,b,c,d,e,f]
```

We ask another question:

```
?- append([ethanol,toluene],[acetone,benzene],L).
```

Prolog responds:

```
L = [ethanol,toluene,acetone,benzene]
```

In the above question, the first two arguments of the **append** functor are input lists. The third argument is the output list. We can view the information flow as:

append(*input*,*input*,*output***).**

But, we can also change the information-flow direction. Let us make arguments one and two the *output* lists, and argument three the *input* list. To do this, we ask the question:

```
?- append(L1,L2,[ethanol,toluene,acetone,benzene]).
```

This is a perfectly acceptable Prolog question. It asks, "What two lists **L1** and **L2**, when appended together, give the resulting list **[ethanol,toluene,acetone,benzene]** ?" The input and output terms are reversed, and we can view the information flow as:

append(*output*,*output*,*input*)

Prolog responds to the question with five acceptable answers:

```
L1 = []
L2 = [ethanol,toluene,acetone,benzene] ;

L1 = [ethanol]
L2 = [toluene,acetone,benzene] ;

L1 = [ethanol,toluene]
L2 = [acetone,benzene] ;
```

```
L1 = [ethanol,toluene,acetone]
L2 = [benzene] ;

L1 = [ethanol,toluene,acetone,benzene];
L2 = [] ;

no
```

In this case, we did not use **append** to construct a list. Instead, we used it to *tear down*, or *decompose* a list. There are five acceptable ways to decompose the list **[ethanol,toluene,acetone,benzene]** such that the **append** relation is true. Because **append** can give multiple results to a question, it is *nondeterministic*.

Note also that the information flow went from

append(*input*,*input*,*output***)**

to

append(*output*,*output*,*input***)**.

We can just as well do

append(*input*,*output*,*input***)**

with the question:

```
?- append([ethanol],L2,[ethanol,toluene,acetone,benzene]).
```

and Prolog responds

L2 = [toluene,acetone,benzene]

We now have three different ways to utilize the **append** relation:

append(*input*,*input*,*output*)

append(*output*,*output*,*input*)

append(*input*,*output*,*input*)

The **append** relation actually has a total of *seven* different information-flow variations, summarized in Table 3.6.

Information flows and list processing can sometimes be confusing to engineers and scientists more familiar with procedural languages such as FORTRAN. They tend to view Prolog predicates incorrectly as "mini-subroutines." In a procedural language, the information flow of terms passed to and from a subroutine usually remains the same. Terms are designated as input and output terms, and they rarely change their status, since procedural languages are algorithmic.

Table 3.6. Variations of the append relation.

ARGUMENT STATUS IN APPEND RELATION			
FIRST	SECOND	THIRD	RESULTS AND EXAMPLES
input	*input*	*input*	Prolog answers ***yes*** or ***no***, e.g.
bound	bound	bound	**?- append([a,b],[c,d],[a,b,c,d]).** ***yes***
input	*input*	*output*	Instantiation of third argument:
bound	bound	free	**?- append([a,b],[c,d],L).** ***L = [a,b,c,d]***
input	*output*	*input*	Instantiation of second argument:
bound	free	bound	**?- append([a,b],L,[a,b,c,d]).** ***L = [c,d]***
output	*input*	*input*	Instantiation of first argument:
free	bound	bound	**?- append(L,[c,d],[a,b,c,d]).** ***L = [a,b]***
output	*output*	*input*	Divides third argument:
free	free	bound	**?- append(L1,L2,[a,b,c]).** ***L1 = [] L2 = [a,b,c] ;*** ***L1 = [a] L2 = [b,c] ;*** ***L1 = [a,b] L2 = [c] ;*** ***L1 = [a,b,c] L2 = [ ] ;*** ***no***
output	*input*	*output*	Puts anonymous variables in at beginning of the second argument:
free	bound	free	**?- append(L1,[a,b,c],L2).** ***L1 = [] L2 = [a,b,c] ;*** ***L1 = [_] L2 = [_,a,b,c] ;*** ***L1 = [_,_] L2 = [_,_,a,b,c] ;*** etc.
input	*output*	*output*	Puts anonymous variables in at the end of the first argument:
bound	free	free	**?- append([a,b,c],L2,L3).** ***L2 = [] L3 = [a,b,c] ;*** ***L2 = [_] L3 = [a,b,c,_] ;*** ***L2 = [_,_] L3 = [a,b,c,_,_] ;*** etc.

In Prolog, however, the status of a term can be dynamic. Because Prolog is relational and goal-oriented, it attempts to match terms. Therefore, *any* term in a predicate can be an input or output term. Its status can change as the program executes. Prolog predicates are *not* subroutines.

B. Reverse

The **reverse** relation takes a list with any number of elements, i.e.,

$[E_1, E_2, E_3, \ldots, E_{n-1}, E_n]$

and arranges the elements in the reverse order as the result:

$[E_n, E_{n-1}, \ldots, E_3, E_2, E_1]$

The base case corresponds to the fact that the reverse of an empty list is itself an empty list. We write the fact:

```
reverse([],[]).                              % base case
```

We write the recursive case:

```
reverse([Head|Tail],ReverseList):-           % recursive case
```

```
          reverse(Tail,ReverseTail),
          append(ReverseTail,[Head],ReverseList).
```

Procedurally, this statement says to reverse a list, "first reverse the tail of the list, and then append the head on to the end of the reversed tail." Note that this **reverse** relation has an arity of two (i.e., has two arguments). We shall refer to it as the "normal" **reverse** relation below.

The **reverse** of a list is defined such that:

```
?- reverse([e,t,a,b],[b,a,t,e]).
```

is true; while

```
?- reverse([e,t,a,b],[b,a,e,t]).
```

is false.

We ask the following question:

```
?- reverse([e,t,a,b],X).
```

and Prolog responds:

X = [b,a,t,e]

We ask:

```
?- reverse([e,t,a,b],[X,a,t,Y]).
```

and Prolog answers:

```
X = b
Y = e
```

Try to trace through this yourself.

We can develop a second, more efficient way to write the reverse relation in Prolog. We write **reverse** in a way that incorporates the append relation directly into the procedure using an *accumulator* list. The accumulator is initially an empty list, **[ ]**. As Prolog progresses through the relation, it steadily builds up the reverse of the input list by adding elements into the accumulator. Once the reversal is complete, the accumulator contains the result, i.e., the reverse of the input list. The base case **reverse([],Result,Result)** passes the accumulated list to the result. We write the entire relation as:

```
reverse([],Result,Result).                    % base case
reverse([X|L],Accumulator,Result):-           % recursive case
            reverse(L,[X|Accumulator],Result).
```

To properly use this "enhanced" **reverse** relation, our information flow is:

reverse(*input,[ ],output*).

The accumulator is initialized to an empty list [ ].

Procedurally, the base case says, "The first argument, the input list, is now empty. Therefore, we know that the input list has been fully processed. We also know the accumulator contains the result, i.e., the reverse of the input list. Therefore, to complete the procedure, we pass the result from the accumulator to the third argument, the variable **Result.**"

The recursive case says, "The input list is not empty. To reverse the input list, we build up the accumulator by taking the head of the input list and adding it on to the accumulator. Then we reverse the tail of the input list with the recently built-up accumulated list to give the final result."

The question

```
?- reverse([e,t,a,b],[],Result).
```

yields

```
Result = [b,a,t,e]
```

This "enhanced" **reverse** relation has an arity of three while the "normal" **reverse** relation has an arity of two. The enhanced relation eliminates the need for appending lists at the end by using accumulator to incorporate the append relation into the procedure. Consequently, the enhanced version is more efficient than the normal version; the enhanced version takes fewer steps to reach the answer.

Using accumulators makes procedures more efficient; Prolog arrives at the answer quickly, and uses less memory space. Accumulators do have a disadvantage, however-- programs written with accumulators are usually more difficult to read and understand than those without accumulators. This difficulty is seen with the **reverse** relation, where the "normal" version with an arity of two is, for most people, more intuitive and easier to understand than the "enhanced" version with an arity of three.

C. Add and Delete

The **add** relation adds an object onto a list. A new list results, with the added object as the head and the old list as the tail. With X as an object and L as a list, we write **add** as a fact:

```
    add(X,L,[X|L]).
```

This says, "When X is added onto list L, the result is a list with X as its head and L as its tail."

The **Add** predicate is defined such that:

```
?- add(b,[a,t,e],[b,a,t,e]).
```

is true, while

```
?- add(b,[a,t,e],[a,a,t,e]).
```

is false. **Add** does not require recursion.

The **delete** relation deletes an object from a list. We separate **delete** into a base case and a recursive case. We first define **delete(X,L,R)** where X is an object to be deleted from list L, giving result R. The information flow for **delete** is:

delete(*input-object, input-list, output-list***)**

We then identify:

(1) *Base case*- if object X is the head of the input list, then the resulting output list is the tail of the input list:

```
delete(X,[X|Tail],Tail).
```

(2) *Recursive case*- we know object X is not the head of the input

list, since it was just tested in the base case. Therefore, we must retain the current head of the input list and build up the output list by adding the head of the input list onto the output list. We are not done yet-- we still need to analyze the tail of the input list and build up the tail of the output list. Object X may be in the tail of the input list. Therefore, we recursively call the **delete** relation again to determine if X is the tail of the input list, and then use the result from that to build the tail of the output list. We write this as:

```
delete(X,[Y|Tail1],[Y|Tail2]):-
      delete(X,Tail1,Tail2).
```

We define the entire **delete** relation using both clauses as:

```
delete(X,[X|Tail],Tail).
delete(X,[Y|Tail1],[Y|Tail2]):-
      delete(X,Tail1,Tail2).
```

The **delete** relation is defined such that:

```
delete(t,[b,a,t,e],[b,a,e]).
```

is true, while

```
delete(t,[b,a,t,e],[b,a,t]).
```

is false. In addition, the **delete** relation will fail if the element to be deleted is not in the list.

Like **append**, **delete** is *nondeterministic*. This means that it can give multiple answers to a query upon backtracking. If we pose the following question to Prolog:

```
?-delete(t,List,[b,a,e]).
```

Prolog responds:

```
List = [t,b,a,e] ;
List = [b,t,a,e] ;

List = [b,a,t,e] ;
List = [b,a,e,t] ;
no
```

You may wish to trace through the procedure to find out why Prolog answers this way.

Another aspect of nondeterminism in **delete** appears when an element appears more than once in a list the way **t** does in the following question:

```
?- delete(t,[b,t,a,t,e,t],R).
```

Prolog answers:

```
R = [b,a,t,e,t] ;
R = [b,t,a,e,t] ;
R = [b,t,a,t,e] ;
no
```

Perform a trace here to see if you get the same result.

D. Sublist

The **sublist** relation tests whether one list is a sublist of another. For **sublist(X,Y)** to be true, X and Y must both be lists, and X has to be present within Y as a sublist. The *order* of the elements in both lists is essential in the relation. For X to be a sublist of Y, all the elements in X must be present in Y. In addition, the elements in X *must be in the same order* as those in Y. Perhaps the simplest way to implement **sublist** is:

```
sublist(X,Y):-
      append(_,Y2,Y),
      append(X,_,Y2).
```

This says, "X is a sublist of Y if: 1) Y can be separated into two lists, one of which is Y2, and 2) Y2 can in turn be separated into two lists, one of which is X." The sublist relation is defined such that:

```
sublist([a,t],[b,a,t,e])
```

is true, while

```
sublist([a,e],[b,a,t,e])
```

is false and therefore fails. Note the use of anonymous variables _ in the definition.

The **sublist** relation is nondeterministic, and can have the following information flows in a Prolog program:

sublist(*input*,*input***)**

sublist(*input*,*output***)**

sublist(*output*,*input***)**

E. First and Last Elements

The predicate **first(X,L)** tests whether object X is the first element of the list L. We express this relation as the Prolog fact:

```
first(X,[X|_]).
```

This says "X is the first element in a list if X is the head of the list." The **first** relation is defined such that:

```
first(b,[b,a,t,e])
```

is true, while

```
first(a,[b,a,t,e])
```

is false. Recursion is unnecessary.

The **first** relation is deterministic, and can have the following information flows in a Prolog program:

first(*input*,*input***)**

first(*input*,*output***)**

first(*output*,*input***)**

The **last** relation tests whether object X is the last element in list L. We define the base case and the recursive case as:

(1) *Base case*- if the list has only one element X, X is the last element in the list. We write the fact:

```
last(X,[X]).
```

(2) *Recursive case*- X is the last element in a list if X is the last element in the tail. We write the recursive rule:

```
last(X,[_|Tail]):-
        last(X,Tail).
```

The complete last relation is:

```
last(X,[X]).
last(X,[_|Tail]):-
        last(X,Tail).
```

The **last** predicate is defined such that:

```
last(e,[b,a,t,e])
```

is true, while

```
last(a,[b,a,t,e])
```

is false. The following question:

```
?- last(X,[b,a,t,e]).
```

yields the Prolog response *X* = *e*. If we ask:

```
?- last(b,X).
```

Prolog responds (depending on the version of Prolog we are using):

```
X = [b] ;
X = [_,b] ;
X = [_,_,b] ;
X = [_,_,_,b] ;
X = [_,_,_,_,b] ;
. . .
```

We see that last(X,L) is also nondeterministic, and can have the following information flows:

last(*input*,*input*)

last(*input*,*output*)

last(*output*,*input*)

F. Length

The **length** relation calculates the number of elements (i.e., the length) in a list, giving an integer as the result. To implement **length**, we use a counter in a recursive loop. We define the base case and recursive case as:

(1) *Base case*- the length of an empty list is zero. We write the fact:

```
length([],0).
```

(2) *Recursive case*- the length of a non-empty list is the length of its tail plus one. We write the recursive rule:

```
length([_|Tail],N):-
            length(Tail,TailLength),
            N is TailLength + 1.
```

If L is a list and N is an integer, then **length(L,N)** is defined such that:

```
length([b,a,t,e],4)
```

is true, while

```
length([b,a,t,e],3)
```

is false. The question

```
?- length([b,a,t,e,s],N).
```

gives the answer ***N*** *= 5*.

G. Remove Duplicate

The **remove_duplicate** relation removes all duplicate elements in a list. It takes list L1, removes any duplicate elements, and outputs the new list that is free of duplicates to list L2. We define **remove_duplicate** using an accumulator. The information flow in the relation is:

remove_duplicate(*input*,*accumulator*,*output***)**.

When first called, **remove_duplicate** *must* have the accumulator initialized to an empty list. The information flow on the first call is:

remove_duplicate(*input*,*[ ]*,*output***)**

We define the relation:

```
remove_duplicate([],L2,L2).                    % rule 1
remove_duplicate([H|T],Accumulator,L2):-       % rule 2
```

```
          member(H,Accumulator),
          remove_duplicate(T,Accumulator,L2).
remove_duplicate([H|T],Accumulator,L2):-        % rule 3
          remove_duplicate(T,[H|Accumulator],L2).
```

The logic behind rules 1, 2, and 3 is as follows:

Rule 1- if input list L1 is the empty list, the processing is completed. The accumulator now contains the result, so we pass the result to the third argument.

Rule 2- if the head of the input list is a member of the accumulated list, we have a duplication of elements in both lists. We do not want to add that element onto the accumulator, since the accumulator contains a list free of duplicates. Therefore, we continue the **remove_duplicate** procedure, analyzing the tail of the input list. We do not change or add onto the accumulator, since the head of the input list is already a member of the accumulator.

Rule 3- if we are at this point, we know from rule 2 that the head of the input list is *not* a member of the accumulator. We add the head of the input list onto the accumulator, since it is a unique object that is currently not a member of the accumulator. Then, we continue the **remove_duplicate** procedure, analyzing the tail of the input list.

Prolog responds to this program with the following dialogue:

```
?- remove_duplicate([b,b,a,a,t,t,e,e],[ ],L).
L = [e,t,a,b]
?- remove_duplicate([1,2,3,4],[ ],L).
L = [4,3,2,1]
?- remove_duplicate([ ],[ ],L).
L = [ ]
```

This procedure gives incorrect results if we backtrack. For instance, we ask again:

```
?- remove_duplicate([b,b,a,a,t,t,e,e],[ ],L).
```

Prolog first responds:

```
L = [e,t,a,b]
```

Now if we look for alternate solutions by forcing backtracking with the semicolon, we see the following dialogue:

```
L = [e,t,a,b] ;
```

```
    L = [e,e,t,a,b] ;

    L = [e,t,t,a,b] ;

    L = [e,e,t,t,a,b] ;

    L = [e,t,a,a,b] ;

    L = [e,e,t,a,a,b] ;

    (etc.)
```

If we continue to enter the semicolon operator, we get sixteen answers.

H. Summary of List Processing Relations

Table 3.7 summarizes the list processing relations introduced in this section.

Table 3.7. List processing in Prolog.

RELATION	USAGE	
append	definition	append([],L,L). append([H1|T1],L2,[H1|T3]):- append(T1,L2,T3).
	examples	?- append([a,b],[c],X). *X = [a,b,c]* ?- append([a,b],X,[a,b,c]). *X = [c]*
	determinism	nondeterministic

reverse (*normal*)	definition	reverse([],[]). reverse([H|T],RL):- reverse(T,RT), append(RT,[Head],RL).
	example	**?- reverse([a,b,c,d],X).** ***X = [d,c,b,a]***
	determinism	nondeterministic
reverse *(enhanced)*	definition	reverse([],R,R). reverse([X|L],A,R):- reverse(L,[X|A],R).
	example	**?- reverse([a,b,c,d],[],X).** ***X = [d,c,b,a]***
	determinism	nondeterministic
add	definition	add(X,L,[X|L]).
	example	**?- add(a,[b,c,d],X).** ***X = [a,b,c,d]***
	determinism	deterministic
delete	definition	delete(X,[X|T],T). delete(X,[Y|T1],[Y|T2]):- delete(X,T1,T2).
	example	**?- delete(d,[a,b,c,d],X).** ***X = [a,b,c]***
	determinism	nondeterministic
member	definition	member(X,[X|_]). member(X,[_|Tail):- member(X,Tail).
	example	**?- member(d,[a,b,c,d]).** ***yes***
	determinism	nondeterministic
sublist	definition	sublist(X,Y):- append(_,Y2,Y), append(X,_,Y2).
	examples	**?-sublist([b,c],[a,b,c,d]).** ***yes*** **?-sublist([a,c],[a,b,c,d]).** ***no***
	determinism	nondeterministic

first	definition	first(X,[X|_]).
	example	**?- first(X,[a,b,c,d]).** ***X = a***
	determinism	deterministic
last	definition	last(X,[X]). last(X,[_|Tail]):- last(X,Tail).
	example	**?- last(X,[a,b,c,d]).** ***X = d***
	determinism	deterministic
length	definition	length([],0). length([_|T],N):- length(T,TL), N is TL + 1.
	examples	**?- length([a,b,c,d],X).** ***X = 4*** **?- length([ ],X).** ***X = 0***
	determinism	deterministic
remove duplicate	definition	remove_dup([],L2,L2). remove_dup([H|T],A,L2):- member(H,A), remove_dup(T,A,L2). remove_dup([H|T],A,L2):- remove_dup(T,[H|A],L2).
	example	**?- remove_dup([a,a,b,b,c,c],[],X).** ***X = [c,b,a]***
	determinism	nondeterministic (*yields incorrect results if backtracked into*)

I. Exercises

3.4.1 In Prolog, the fact **triangle** gives three lists (in right cartesian coordinates) of the locations of the corner points of a triangle. For instance, consider a triangle with corner points at (1,1), (3,1), and (2,2). In Prolog, we write:

```
triangle([ 1, 1], [3, 1], [2, 2] ).
```

Given the fact **triangle**, write:

a. The relation **area**, that calculates the area of the triangle.

b. The relation **perimeter**, that calculates the perimeter of the triangle.

c. The relation **right**, that determines if the triangle is a right-triangle.

d. The relation **equilateral**, that determines if the triangle is an equilateral triangle.

3.4.2 Define the relations **split_below** and **split_above**. The **split_below** relation splits a list at a particular element, placing that element into the second split list. For instance,

```
?- X = [methane,ethane,propane,isobutane,n_butane],
   split_below(X,ethane,L1,L2).
```

gives

L1 = [methane] L2 = [ethane, propane, isobutane, n_butane]

and

```
?- X = [methane,ethane,propane,isobutane,n_butane],
   split_below(X,propane,L1,L2).
```

gives

L1 = [methane,ethane] L2 = [propane, isobutane, n_butane]

The **split_above** relation also splits a list at a particular element. In contrast to **split_below**, however, **split_above** places that element in the first split list. For instance,

```
?- X = [methane,ethane,propane,isobutane,n_butane],
   split_above(X,ethane,L1,L2).
```

yields

L1 = [methane,ethane] L2 = [propane, isobutane, n_butane]

and

```
?- X = [methane,ethane,propane,isobutane,n_butane],
   split_above(X,propane,L1,L2).
```

yields

```
L1 = [methane,ethane,propane] L2 = [isobutane, n_butane]
```

Write Prolog rules for **split_below** and **split_above**.

3.4.3 Define the relation **permutation** that generates permutations of a list when backtracked into. For example, the question

```
?- permutation( [acid, base, neutral], P).
```

yields the following dialogue:

```
P = [acid, base, neutral] ;
P = [acid, neutral, base] ;
P = [base, acid, neutral] ;
P = [base, neutral, acid] ;
. . . . . . . .
```

3.4.4 Define the relation **positive_flow** that, given a list of components and their flow rates in a process stream, the result is an output list of all components with flow rates greater than zero. For instance, we ask the question:

```
?- X = [methanol, ethanol, propanol, water],
   Y = [ 0, 10, 12.5, 0],
   positive_flow(X,Y,Z).
```

and Prolog responds:

```
Z = [ethanol, methanol]
```

3.4.5 Define the two relations **even_length** and **odd_length**. The **even_length** relation is true if the length of the list is an even number, i.e., divisible by two. The **odd_length** relation is true if the length of the list is an odd number. For instance, the following dialogue results from the relations:

```
?- evenlength([methane, ethane, propane]).
no
?- oddlength([methane, ethane, propane]).
yes
?- evenlength([methane, ethane, propane, isobutane]).
yes
?- oddlength([methane, ethane, propane, isobutane]).
```

no

3.4.6 Define the relation **sumlist(X,Y)**, that takes the list of numbers, X, and sums all the elements within list X to give result Y. For instance, **?- sumlist ([0, 10, 12.5, 0],Y).** yields the result *Y = 22.5.*

3.4.7 Define the relation **maxlist(X,Y)**, that takes the list of numbers, X, and instantiates Y to the maximum number in the list. For instance, **?- maxlist ([0, 10, 12.5, 0], Y).** yields the result *Y = 12.5*

3.4.8 Define the relation **max_component_flow(C,F,R)** that takes in **C**, the list of components in a flow stream, **F**, the list of flow rates for each of those components, and outputs **R**, the resulting list that has both the component name and flow rate for the component with the maximum flow. For instance, the question:

```
?- C = [methane,ethane,propane,butane],
   F = [10,20,30,10],
   max_component_flow(C, F, R).
```

leads to the answer:

```
R = [propane, 30]
```

3.5 CHAPTER SUMMARY

- *Operators* can be used to improve the readability of programs. In themselves they take no action.

- There are *prefix*, *infix*, and *postfix* operators, named according to their location.

- The arithmetic operators are +, -, *, /, and **mod**. Some versions of Prolog also use the **div** operator for integer division.

- Prolog uses built-in predicates to numerically evaluate arithmetic expressions. Generally, **is** and **=:=** are used.

- The Prolog equality operator **=** does *not* do arithmetic evaluation. In some versions of Prolog that do not support is, however, the **=** *can* do numerical evaluation.

- When numerical evaluation is done, operators act on a priority level. The priority level can be adjusted by the use of parenthesis.

- A *list* is an ordered sequence of any length. It consists of a *head*, which is a Prolog object, and a *tail*, which is another list.

- Prolog programming becomes more powerful when we combine recursion with lists. To do this, we must identify a *base case* and a *recursive case*.

3.6 PRACTICE PROBLEMS

An engineer is designing a distillation unit that will separate a four-component stream. Table 3.8 shows the material-balance specifications. The feed stream has four components, A, B, C, and D. Four product streams, P1, P2, P3, and P4 are desired. We represent the entire material balance with the collection of Prolog facts shown in Figure 3.4. For example, by referring to Table 3.8, we see the statement **flow(p2,a,10)** in the database means that the flow rate of component a in product **p2** is 10 mol/hr. Based on this Prolog database, develop the following Prolog relations:

(Note: a single predicate will probably not suffice for each of the questions below. The questions asked are moderately complex, and will require you to develop some simple, "support" or "utility" predicates to make the relation work.)

1. Develop the relation **sum_component_flow(List,C,F)**, where **List** is a given list of products (e.g., [p1,p2,p3]), **C** is a given component (e.g., component a), and **F** is the cumulative total flow rate of component **C** in the list of products **List**. As an example, the goal **sum_component_flow([p1,p2,p3],b,F)** yields the result *F* = ***25***. The unit "mol/hr" is implied.

2. Develop the relation **sum_product_flow(CList,P,F)**, where **CList** is a

```
flow(p4,a,0).   flow(p4,b,0).    flow(p4,c,0).   flow(p4,d,15).
flow(p3,a,0).   flow(p3,b,0).    flow(p3,c,20).  flow(p3,d,10).
flow(p2,a,10).  flow(p2,b,12.5). flow(p2,c,0).   flow(p2,d,0).
flow(p1,a,15).  flow(p1,b,12.5). flow(p1,c,5).   flow(p1,d,0).
```

Figure 3.4. Prolog database for problem material balance.

given component list (e.g., [a,b,c]), **P** is a given product (e.g.,p2), and **F** is the cumulative total flow rate of components in list **CList** designated for product **P**. For example, the goal **sum_product_flow([a,b,c],p2,F)** yields the result *F* = *22.5*. Again, the unit "mol/hr" is implied.

Table 3.8. Feed and product specifications.

Desired product streams	Component flow rate (mol/hr) A	B	C	D	Product flow rate (mol/hr)
P4	0	0	0	15	15
P3	0	0	20	10	30
P2	10	12.5	0	0	22.5
P1	15	12.5	5	0	32.5
Component flow rate (mol/hr)	25	25	25	25	100

3. Develop the relation **mole_fraction(CList,FList,C,X)**, where **CList** is a given list of components in a process stream, **FList** is the corresponding flow rate of each component in that process stream, **C** is the given component, and **X** is the calculated mole fraction of component **C** in the

process stream. For example, the goal **mole_fraction([a,b,c,d],[25,25,25,10],d,X)** gives the result ***X* = *0.1176*.**

4. Develop the relation **positive_component_flow(List,CList,PCFList)**, where **List** is a given list of products in the material balance, **CList** is a given list of components, and **PCFList** is the positive component flow list, i.e., the list of components from **Clist** that have positive (i.e., non-zero) flow rates in the product list **List**. For example, the goal **positive_component_flow_list([p1,p2],[b,c,d],PCFList)** yields the result ***PCFList* = *[b,c]***, since component **d** has zero flow rate in products **p1** and **p2**.

5. Develop the relation, **most_plentiful_product(List,CList,P)** where **List** is the list of products from the material balance, **CList** is the list of components from the material balance, and **P** is the most plentiful product, i.e., the product from **List** consisting of component flows from **CList** such that the total flow of product **P** is greater than that of any other product in **List**.

For example, the goal:

most_plentiful_product([p1,p2,p3],[b,c],P),

produces the result ***P* = *p3***. Why? Product **p1** has a combined flow rate of 17.5 mol/hr for components **b** and **c**, while product **p2** has one of 12.5

mol/hr. Product **p3**, however, has a combined flow rate of 20 mol/hr for components b and c, and is therefore the most plentiful product.

A LOOK AHEAD

Having completed this chapter, we are now able to start doing practical Prolog programming, such as Practice Problems in Section 3.6. With the skills developed in arithmetic and list processing, we have the tools to tackle almost any problem. What we need now is the ability to *control* programs and solve problems *efficiently*. To develop this ability, we shall introduce a Prolog control facility called the *cut* in chapter four.

REFERENCES

Bratko, Ivan, *Prolog Programming for Artificial Intelligence*, second edition, pp. 67-96, Addison-Wesley, Reading, MA (1990).

Clocksin, W.F. and C.S. Mellish, *Programming in Prolog*, third edition, pp. 28-66, 49-59, Springer-Verlag, New York, NY (1987).

Garavaglia, Susan, *Prolog Programming Techniques and Applications*, pp. 80-109, Harper & Row Inc., New York, NY (1987).

Rowe, Neil C., *Artificial Intelligence Through Prolog*, pp. 74-89, Prentice-Hall, Englewood Cliffs, NJ (1988).

Schnupp, P. and L. Bernhard, *Productive Prolog Programming*, pp.15-46, Prentice-Hall, Englewood Cliffs, NJ (1986).

Sterling, L. and Shapiro E., *The Art of Prolog*, pp. 33-67, MIT Press, Cambridge, MA (1986).

4

BACKTRACKING AND PROGRAM CONTROL

This chapter introduces techniques of program control. The primary way to control program execution is through use of the *cut*, a built-in Prolog command that prevents backtracking. After introducing the cut, we investigate in more detail both how and why Prolog backtracks. We then examine the use of the cut in more detail, and close the chapter with an assessment of its benefits and consequences.

4.1 PREVENTING BACKTRACKING: THE CUT

Consider again our "environmental" Prolog program tied into a water analyzer monitoring the chemicals in a process stream. We previously used the **member** predicate to determine if a certain chemical is in a list. The **member** relation is:

```
member(X,[X|_]).                    % base case
member(X,[_|Tail]):-                % recursive case
            member(X,Tail).
```

The base case says, "X is a member of a list if X is the head of a list." The recursive case says, "X is a member of a list if X is a member of the tail of a list."

If we ask:

```
?- X = [benzene, p_xylene, o_xylene, toluene],
   member(toluene, X).
```

Prolog responds:

```
yes
```

We want Prolog to issue a warning if: 1) a detected chemical is in a pre-defined list of "bad actors," and 2) its concentration is over 100 parts per million (ppm) in the process stream. If X is the chemical and N is its concentration in ppm, we write the Prolog rule:

```
sound_alarm(X,N,BadActorList):-
            member(X,BadActorList),
            N > 100.
```

This rule says "Sound the alarm if X is a member of **BadActorList** and the concentration of X is over 100 ppm in the process stream." Now let us consider the following question:

```
?- sound_alarm(mercury,50,[mercury,dioxin,ddt,kepone]).
```

This question asks, "Do we sound the alarm if mercury is detected in the process stream at 50 ppm (realizing that mercury is a member of the list of bad actors) ?" Prolog answers ***no***, but let us trace the steps it takes. The complete trace is shown in Figure 4.1.

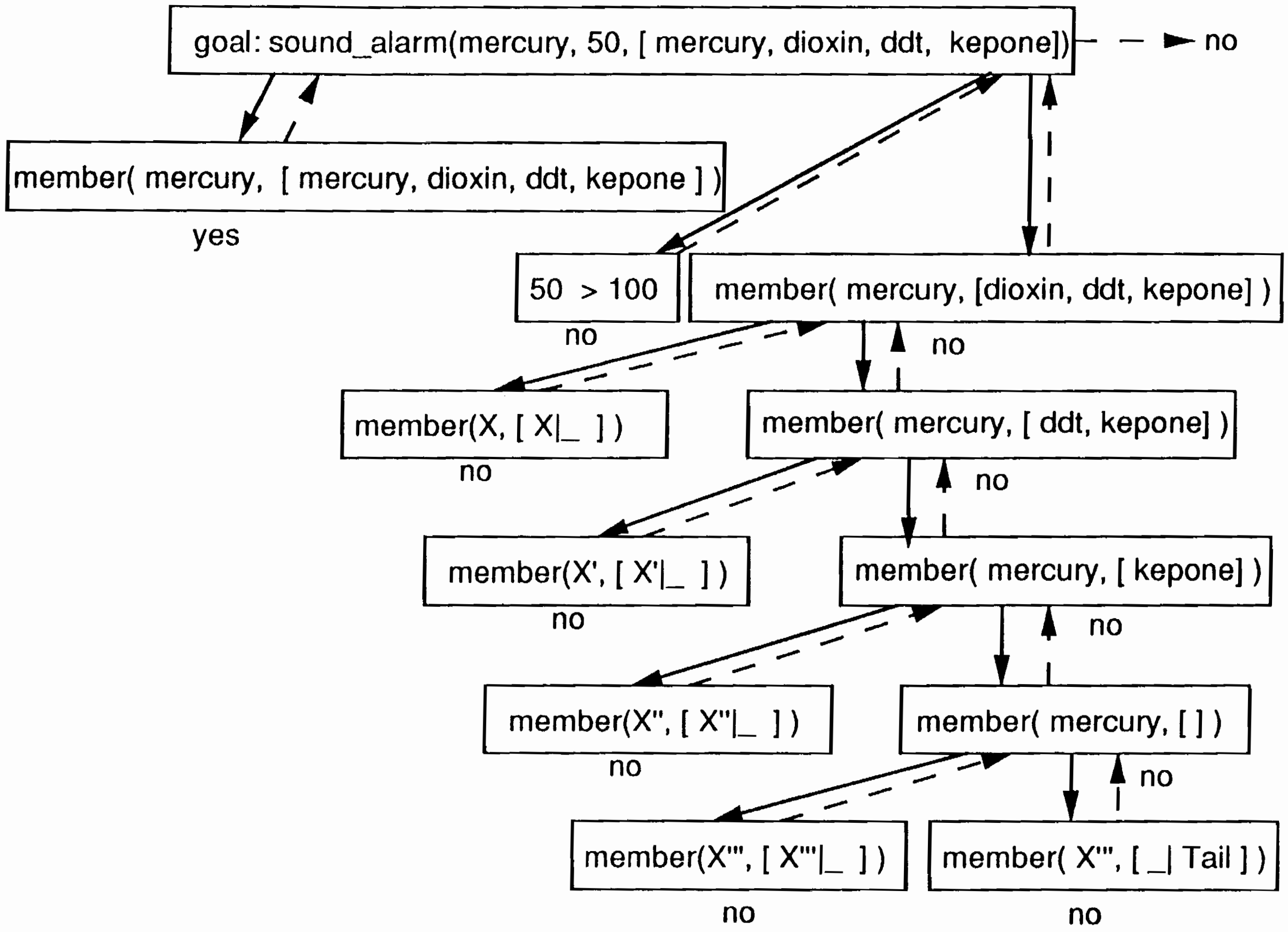

goal: sound_alarm(mercury, 50, [mercury, dioxin, ddt, kepone])
no
member(mercury, [mercury, dioxin, ddt, kepone])
yes
50 > 100
no
member(mercury, [dioxin, ddt, kepone])
no
member(X, [X|_])
no
member(mercury, [ddt, kepone])
no
member(X', [X'|_])
no
member(mercury, [kepone])
no
member(X'', [X''|_])
no
member(mercury, [ ])
no
member(X''', [X'''|_])
no
member(X''', [_| Tail])
no

Figure 4.1. Trace of sound_alarm(mercury, 50, [mercury, dioxin, ddt, kepone]).

Step 1: **sound_alarm(mercury,50,[mercury,dioxin,ddt,kepone])** is the initial goal. Prolog matches the head of the **sound_alarm** clause, instantiating **X** to **mercury**, **N** to **50**, and **BadActorList** to **[mercury,dioxin,ddt,kepone]**. The new goal becomes: **member(mercury,[mercury,dioxin,ddt,kepone])**.

Step 2: The member goal succeeds, since **mercury** is the head of the list. Prolog now continues to check the next goal. Is **50 > 100** ?

Step 3: The **50 > 100** goal fails. But *now* Prolog backtracks, still trying to satisfy the goal. Prolog checked out the first **member** relation, and the relation was true. However, the following **50 > 100** goal failed. Recall that our **member** relation is:

```
member(X,[X|_]).
member(X,[_|Tail]):-
            member(X,Tail).
```

Prolog backtracks with the goal **member(mercury, [mercury, dioxin, ddt, kepone])**, and tries the second **member** relation. The goal matches the head of the second **member** clause, and **X** is instantiated to **mercury** and **Tail** is instantiated to the list **[dioxin,ddt,kepone]**. This **member** clause is a recursive rule. The next goal becomes: **member(mercury,[dioxin, ddt, kepone])**.

Step 4: Prolog now tries to match this goal. Scanning from the top, the base case for **member** fails. The atom **mercury** does not match the head of the list (in this case, the atom **dioxin**). Backtracking, Prolog moves to the second **member** predicate and instantiates **X** to **mercury** and **Tail** to **[ddt,kepone]**. Prolog then makes the recursive call: **member(mercury, [ddt, kepone])**.

Step 5: As in step 4, the first **member** clause fails. Prolog backtracks to the second **member** clause; using the recursive call, it makes the new goal: **member(mercury,[kepone])**.

Step 6: The first **member** relation again fails, and, from the recursive **member** clause, the new goal becomes **member(mercury,[])**.

Step 7: Prolog has now scanned the entire list. Neither **member** predicate matches, so this goal fails. Prolog now tries to backtrack to other **sound_alarm** predicates. No more exists, so the *parent goal*, **sound_alarm(mercury,50,[mercury,dioxin,ddt,kepone])**, fails. Prolog then answers ***no***.

Let us analyze what Prolog did. After Prolog first confirmed that mercury was a member of the list of "bad actors" (**[mercury, dioxin, ddt, kepone]**), it went to the next sub-goal. Here, it tested to see if the concentration was greater than 100 ppm. Since the concentration was only 50 ppm, this test failed. But instead of just replying ***no***, Prolog backtracked through the rest of the **member** predicate in a failed attempt to satisfy the goal. We saw ahead of time that this backtracking was doomed to failure and a waste of time.

This useless backtracking can be prevented by using a new tool, called the *cut*, denoted in Prolog by the exclamation mark, !. We can now write the **sound_alarm** relation as follows:

```
sound_alarm(X,N,BadActorList):-
            member(X,BadActorList),
            !,
            N > 100.
```

In this rule, Prolog does not waste time backtracking uselessly through the **member** relation. The cut blocks backtracking. Now consider the same question:

```
?- sound_alarm(mercury,50,[mercury,dioxin,ddt,kepone])
```

Prolog again answers ***no***, but goes through fewer steps to get the answer. Figure 4.2 traces Prolog's steps, which we explain below.

Step 1: The **sound_alarm(mercury,50,[mercury,dioxin,ddt,kepone])** is again the initial goal. Prolog enters the **sound_alarm** clause, and the next goal, as before, becomes **member(mercury,[mercury,dioxin,ddt,kepone])**.

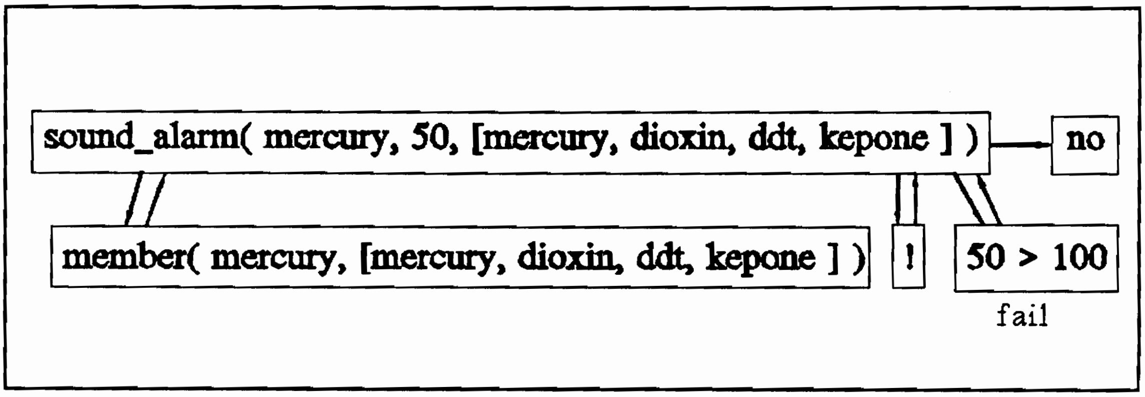

sound_alarm(mercury, 50, [mercury, dioxin, ddt, kepone])
no
member(mercury, [mercury, dioxin, ddt, kepone])
!
50 > 100
fail

Figure 4.2. Trace of sound_alarm goal with a cut.

Step 2: The goal succeeds by matching the first **member** relation. Prolog moves to the next goal, which is the **cut**.

Step 3: The **cut** always succeeds immediately. At this point, all backtracking between the cut and the last successful goal is eliminated. Prolog moves to the next goal, which is **50 > 100**.

Step 4: The **50 > 100** obviously fails. Prolog now attempts to backtrack. All backtracking between the cut and the last successful goal has been eliminated. Prolog has no more options for solving the problem, so it immediately reports ***no***. Importantly, Prolog arrived at the answer much more efficiently than before by *not* backtracking through the **member** relation. In Figure 4.2, Prolog only goes through four search blocks; in Figure 4.1, Prolog goes through twelve blocks to reach the same answer.

The benefits of using the cut are clear. Prolog programs run better if we use the cut correctly. The next section discusses the cut in detail.

4.2 BACKTRACKING IN PROLOG

A. Understanding the Cut

The cut is primarily a *procedural* device. It prevents backtracking, and thus improves program efficiency. Figures 4.1 and 4.2 show how the cut dramatically reduced the amount of search involved before Prolog answered ***no***. However, we may have difficulty seeing from Figures 4.1 and 4.2 what is happening procedurally when the cut is used.

Consider instead Figures 4.3a, b, c, and d. These figures give a better view of the procedural nature of the cut. Here, we have a **parent_goal** calling an **intermediate_goal**. The **intermediate_goal** rule is true if the four sub-goals are true. In Prolog, the relations in the absence of a cut are:

```
parent_goal:-
            intermediate_goal.
intermediate_goal:-
            sub_goal_1,
            sub_goal_2,
            sub_goal_3,
            sub_goal_4.
```

The **intermediate_goal** rule in Figure 4.3a does not have a cut, but

Figures 4.3b through d do. Each example, 4.3a through 4.3d, backtracks differently, as described below.

Figure 4.3a: There is no cut. The Prolog relations are:

```
parent_goal:-
                intermediate_goal.
intermediate_goal:-
                sub_goal_1,
                sub_goal_2,
                sub_goal_3,
                sub_goal_4.
```

If a sub-goal fails *after* the **intermediate_goal** has succeeded (called a "downline fail" in the Figure), Prolog first backtracks to **sub_goal_4**. If **sub_goal_4** fails, it backtracks to **sub_goal_3**. This continues to **sub_goal_2** and **sub_goal_1**. If **sub_goal_1** fails, Prolog backtracks to the **intermediate_goal**.

The lesson here is that without a cut, Prolog simply backtracks to the last satisfied goal and attempts to re-satisfy it in a different way.

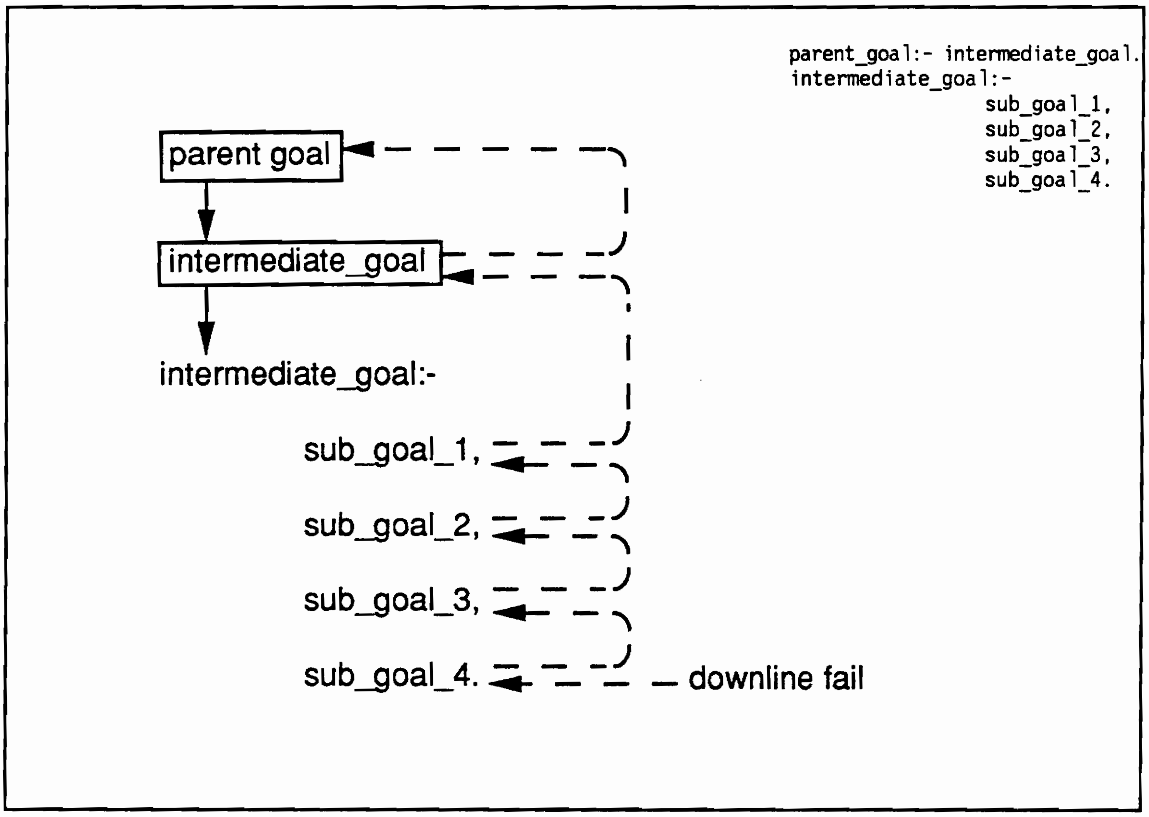


Figure 4.3a. Backtracking path with no cut.

Figure 4.3b: The cut is after **sub_goal_1**, eliminating all backtracking between the cut and the parent goal:

```
parent_goal:-
            intermediate_goal.
intermediate_goal:-
            sub_goal_1,
                    !,
            sub_goal_2,
            sub_goal_3,
            sub_goal_4.
```

With a "downline fail," Prolog backtracks to **sub_goal_4**. Likewise, if **sub_goal_4** fails upon backtracking, Prolog backtracks to **sub_goal_3**, and eventually to **sub_goal_2**.

Now the difference arises. If **sub_goal_2** fails upon backtracking and Prolog is forced to backtrack yet again, it *skips over* **sub_goal_1** and any other **intermediate_goal** rules. It then resumes backtracking at the **parent_goal**.

Note that backtracking is only affected from **sub_goal_2**. Standard backtracking occurs from **sub_goal_4** to **sub_goal_3**, and likewise from **sub_goal_3** to **sub_goal_2**. Standard backtracking even occurs *before* the cut is encountered, i.e., from **sub_goal_1** to the **intermediate_goal**. But when backtracking from **sub_goal_2**, Prolog skips over everything between the cut and the **parent_goal**.

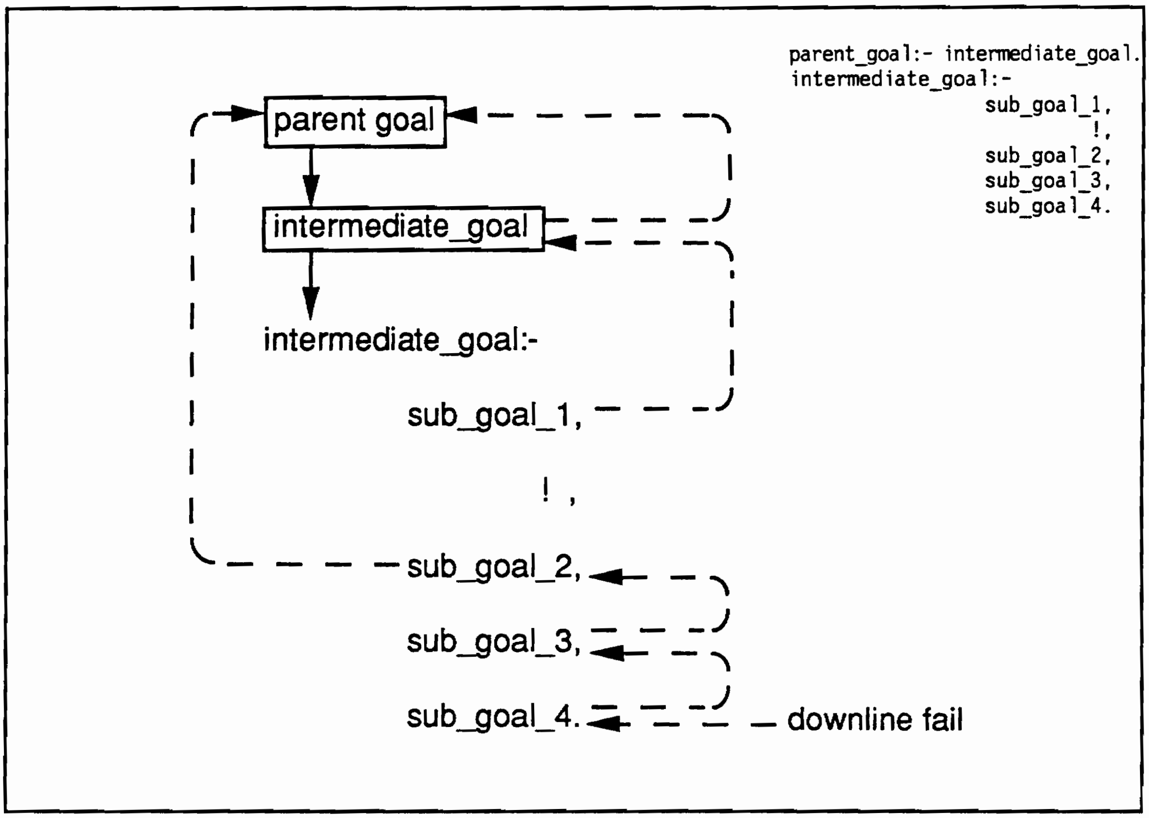


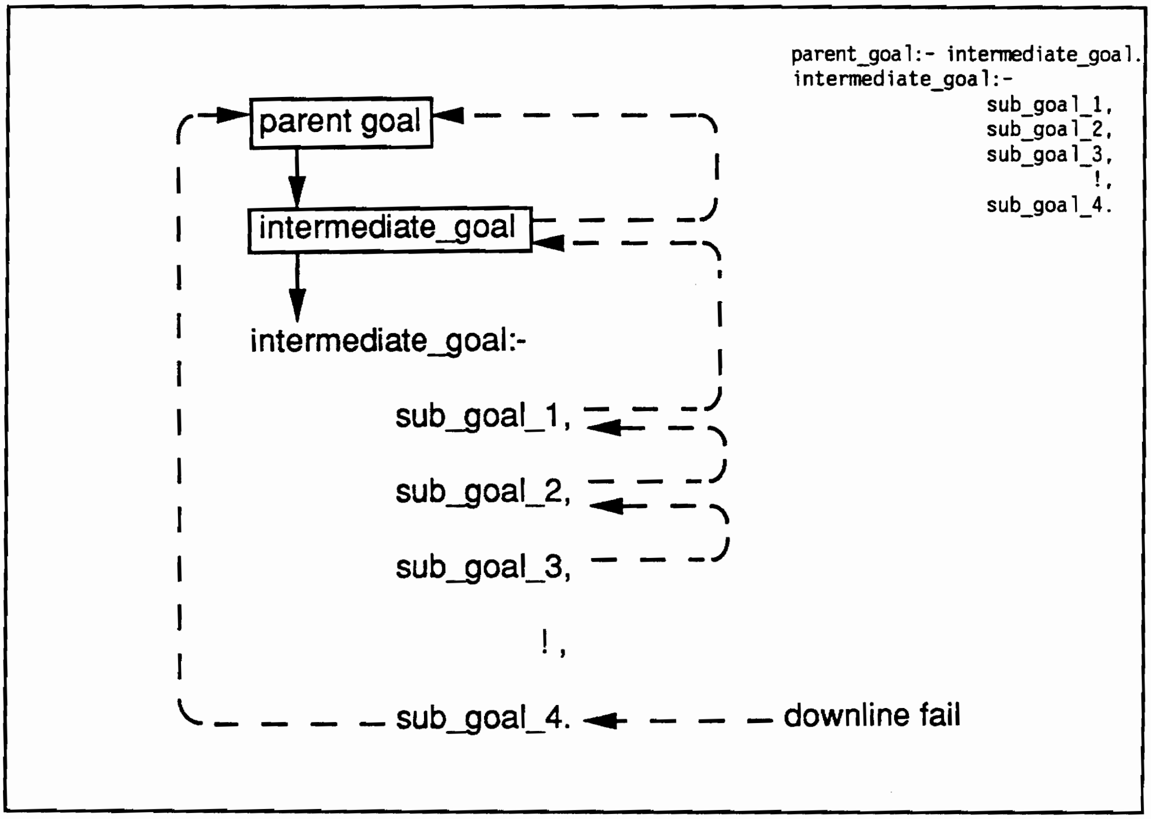
Figure 4.3b. Backtracking path with a cut after sub_goal_1.

Figure 4.3c: This Figure resembles Figure 4.3b in many ways, but the cut is after sub_goal_3:

```
parent_goal:-
            intermediate_goal.
intermediate_goal:-
            sub_goal_1,
            sub_goal_2,
            sub_goal_3,
                    !,
            sub_goal_4.
```

Standard backtracking occurs from: 1) **sub_goal_3** to **sub_goal_2**, 2) **sub_goal_2** to **sub_goal_1**, 3) **sub_goal_1** to **intermediate_goal**, and 4) **intermediate_goal** to **parent_goal**.

When backtracking from **sub_goal_4**, however, Prolog skips right to **parent_goal**. Even if other **intermediate_goal** relations exist, Prolog does not try them.



```
parent_goal:- intermediate_goal.
intermediate_goal:-
                    sub_goal_1,
                    sub_goal_2,
                    sub_goal_3,
                             !,
                    sub_goal_4.
```

Figure 4.3c. Backtracking path with a cut after sub_goal_3.

Figure 4.3d: Here, the cut is at the very end of the rule:

```
parent_goal:-
            intermediate_goal.
```

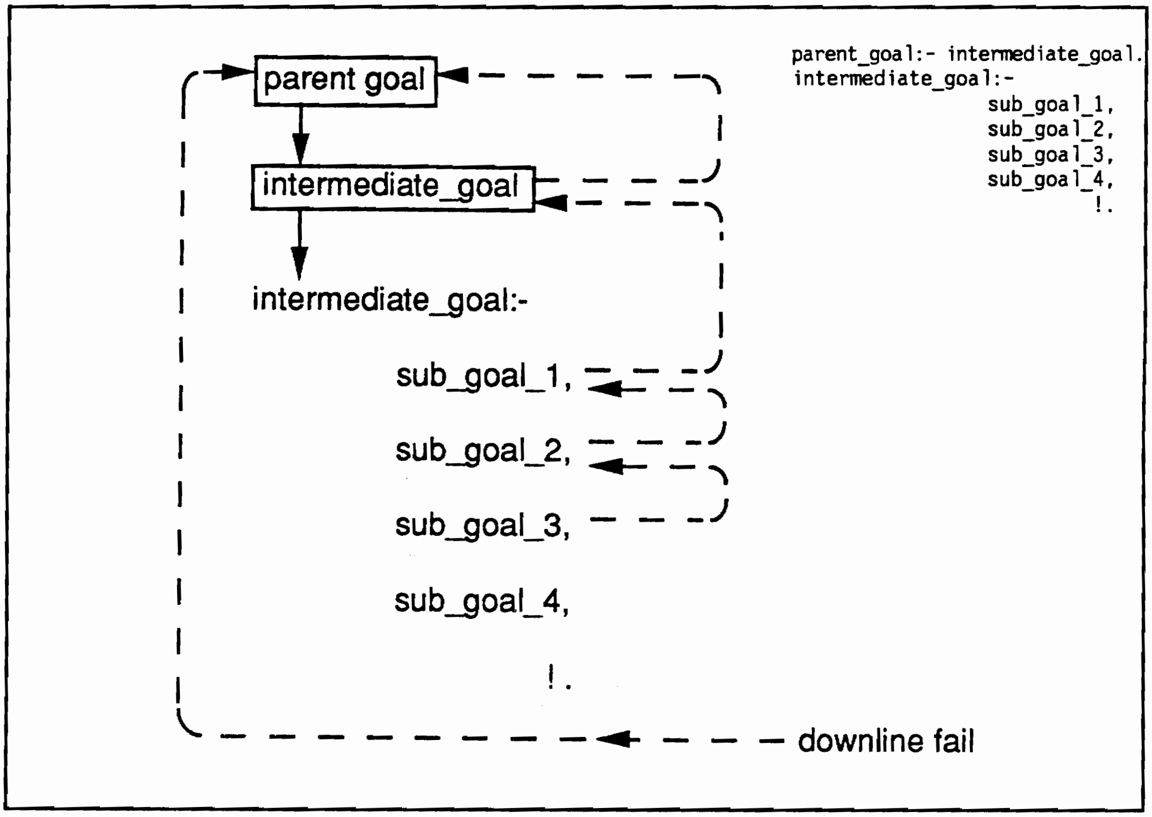
```
intermediate_goal:-
                sub_goal_1,
                sub_goal_2,
                sub_goal_3,
                sub_goal_4,
                            !.
```

If a "downline fail" occurs, forcing Prolog to backtrack, Prolog's built-in backtracking mechanism wants to go to sub-goals within the **intermediate_goal** relation, starting at **sub_goal_4**. The cut prevents it. Prolog skips over all of the sub-goals within the clause upon backtracking. In addition, it skips over other **intermediate_goal** rules, if they exist. Prolog has to backtrack to the last successful rule *prior* to **intermediate_goal**, in this case, the clause **parent_goal**.

Basically, the cut at the end of the rule eliminates *any* attempt at finding alternative solutions to the rule.

B. Why Prolog Backtracks

Backtracking is an essential part of Prolog problem-solving. If a goal fails, Prolog backtracks in an attempt to satisfy the goal. Depending on: 1) the question asked, 2) the ordering of sub-goals inside a rule, and 3) the ordering of rules themselves, Prolog may travel down different pathways to solve a problem. Prior to solving the problem, Prolog knows the *relations* among objects, but does not know the search pathway.



```
parent_goal:- intermediate_goal.
intermediate_goal:-
                    sub_goal_1,
                    sub_goal_2,
                    sub_goal_3,
                    sub_goal_4,
                             !.
```

Figure 4.3d. Backtracking path with a cut at the end of the intermediate_goal rule.

Procedural languages such as FORTRAN focus on the pathway. They require a well-defined and unchanging solution procedure, or *algorithm*. Prolog, however, does not require an algorithm. The pathway can be unknown. Prolog may solve one problem using a particular pathway, and, if the program database changes, use a completely different pathway the next time. By not focusing on the pathway, we are able to solve different, more complex problems. Such complex problems frequently appear in artificial intelligence and are typically *non-algorithmic*. Their pathways are either unknown or too difficult to determine ahead of time. They may have a certain degree of uncertainty also.

Backtracking helps Prolog "carve out" these unknown pathways. If a certain pathway fails, Prolog backtracks in an attempt to satisfy the goal. To backtrack is to look for a new pathway. Importantly, the backtracking procedure is *automatic* in Prolog. We do not have to tell Prolog how to backtrack. It takes that burden on its own shoulders in its relentless attempt to satisfy the goal.

C. How Prolog Backtracks

Clocksin and Mellish (1987,p.9) give a good description of how Prolog satisfies goals and backtracks:

> *"Prolog answers the question by attempting to satisfy the goal. If the first goal is in the database, then Prolog will mark the place in the database, and attempt to satisfy the second goal. If the second goal is satisfied, Prolog marks <u>that goal's</u> place in the database, and we have found a solution that satisfies two goals."*
>
> *"It is most important to remember that each goal keeps its own place marker. If, however, the second goal is not satisfied, then Prolog will attempt to re-satisfy the previous goal (in this case, the first goal). Remember that Prolog searches the database completely for each goal. If a fact in the database happens to match, satisfying the goal, then Prolog will mark the place in case it has to re-satisfy the goal at a later time. But when a goal needs to be re-satisfied, Prolog will begin its search from the goal's own place-marker rather than from the start of the database."*

Summarizing, for every goal, Prolog:

(1) looks from the top-down to satisfy a *new* goal.

(2) picks up where the last satisfied goal is in the database and will attempt to re-satisfy this last satisfied goal via a new pathway.

D. Exercises

4.2.1 Consider the following Prolog program:

```
is_a(benzene,hydrocarbon).
is_a(toluene,hydrocarbon):- !.
is_a(toluene,aromatic).
is_a(xylene,hydrocarbon).
```

How will Prolog respond to the following questions?

a. **is_a(X,Y).**

b. **is_a(X,_),is_a(Y,Z).**

c. **is_a(X,_),!,is_a(Y,Z).**

4.2.2 Develop the set relations **union** and **intersection**. The relation **union(X,Y,Z)** is true if X, Y, and Z are all lists that represent sets, and

X ∪ Y = Z. For example, the question **?- union([a,b,c],[d,e],[c,a,e,d,b]).** is true. The relation **intersection(X,Y,Z)** is true if X, Y and Z are again lists that represent sets, and X ∩ Y = Z. For example, the question **?- intersection([c,a,e,d,b],[d,a,e,f],[e,d,a]).** is true.

Note that in both relations, lists represent *sets*, so the order of elements in the list is not important. Develop these two relations. *Make sure to place cuts in appropriate places such that incorrect answers are not generated if the Prolog backtracks into the relation.*

4.3 COMMON USES OF THE CUT

There are three common uses of the cut. A cut is used for: 1) making relations deterministic, 2) "locking in" a single rule for a particular goal, and 3) using negation as failure, where we use the *cut* and *fail* facilities together.

A. Making Relations Deterministic

A Prolog relation is *nondeterministic* if it generates multiple solutions upon backtracking. Likewise, a relation is *deterministic* if only one solution exists. Using a cut to make a relation deterministic can also be viewed as "freezing-in the first solution" Prolog finds. If Prolog tries to backtrack, it finds that solution "frozen-in" and cannot generated any others.

The **member** relation, which is nondeterministic, is:

```
member(X,[X|_]).
member(X,[_|Tail]):-
            member(X,Tail).
```

We can make the relation deterministic through use of the cut:

```
member(X,[X|_]):- !.
member(X,[_|Tail]):-
                member(X,Tail).
```

Then, when we ask the question:

```
?- member(X,[b,a,t,e]).
```

Prolog responds with only one solution:

```
X = b ;
no
```

When we enter the semicolon ;, we instruct Prolog to backtrack, since entering the semicolon induces a fail. Prolog tries to backtrack in search of alternate solutions, but the cut prevents any backtracking to the **member** relation. Therefore, Prolog responds ***no***.

Using the cut to make procedures deterministic can work for us or against us. We need to fully understand the use of the particular clause involved before using the cut. The benefits of determinism are as follows:

(1) The speed of the program increases, since no time is wasted backtracking.

(2) The program occupies less memory, since backtrack markers are eliminated (i.e., Prolog does not have to "remember" as much).

(3) Unplanned backtracking (which can cause all kinds of havoc) is eliminated.

The cut should be used in this way, only on clauses where it is acceptable to "freeze-in" the first solution and not allow any other solutions to be generated.

B. "Locking in" a Single Rule

Suppose that an engineer is developing a control system for a machine where Prolog performs some qualitative and minor numerical control analyses. As the engineer develops the Prolog relations, he recognizes the need for a rule to calculate the *factorial* of a number. He develops the predicate **factorial(N,Result)** where **N** is the input integer and **Result** equals the factorial of **N**. He develops the following two clauses for **factorial**, the second of which is recursive:

```
factorial(0,1):- !.
factorial(N,R):-
        NN is N - 1,
        factorial(NN,TemporaryR),
        R is TemporaryR*N.
```

The first rule says "the factorial of zero is one." The second rule says

"The factorial of **N** is **R** if: 1) **NN** equals **N - 1**, 2) **TemporaryR** equals the factorial of **NN**, and 3) **R** equals the product of **TemporaryR** and **N**."

The cut is used on the first predicate to "lock in" the rule and prevent Prolog from backtracking to another **factorial** rule. It is saying in effect, "once this rule succeeds (i.e., 0! = 1), you should never backtrack to another **factorial** rule." Note also that the rules are mutually exclusive; either N is 0 or it is not.

Why is this cut needed? The recursive **factorial** rule operates by marching closer and closer to the base case **factorial(0,1)** by subtracting 1 from the value of N. Eventually, the first term in the **factorial** clause is equal to 0. It matches the base case and calculates the factorial.

If Prolog backtracks *without* the cut, we get into trouble. Prolog backtracks to the second **factorial** rule, subtracts 1 from the value of N to get NN (i.e., NN = N - 1 = 0 - 1 = -1), and makes the recursive call:

```
factorial(-1,TemporaryR)
```

This call goes into an infinite loop, since it can never be matched with the base case again. Prolog continues on uselessly with:

```
factorial(-2,TemporaryR')
factorial(-3,TemporaryR'')
factorial(-4,TemporaryR''')
. . .
```

The **TemporaryR** variables, i.e., **TemporaryR'**, **TemporaryR''**, etc. have "prime" markers, ', to show that they are completely separate variables. Prolog will continue these recursive calls until the computer runs out of memory.

The cut prevents the infinite loop. The cut "locks in" the first rule and says, "no more backtracking on the factorial rule after 0! = 1." This lock-in is done because the rules are mutually exclusive.

C. Negation as Failure

The cut is also used for *negation as failure*. Actually, negation as failure uses a *cut-fail* combination to say, "if you get here, no solution exists, so you should stop trying to satisfy the goal."

Consider again our **factorial** relation. Mathematically, if N < 0, the factorial does not exist. We incorporate this into a new set of rules to calculate the factorial:

```
factorial(N,_):-
     N < 0,
     !,
     fail.
factorial(0,1):- !.
factorial(N,R):-
     NN is N - 1,
          factorial(NN,TemporaryR),
          R is TemporaryR*N.
```

The first rule says "If N is less than zero, no factorial exists, so continuing the search for a solution to the goal is pointless. Therefore, perform a cut-fail combination that immediately stops Prolog from trying to satisfy the **factorial** relation any further."

Given the goal **factorial(-1,R)**, for example, Prolog executes the following steps:

Step 1: The initial goal is **factorial(-1,R)**. Prolog encounters the first rule, and matches the head of the clause. The variable **N** is instantiated to -1. The new goal becomes **-1 < 0**.

Step 2: The goal **-1 < 0** succeeds. The new goal is **cut**, which succeeds immediately. The next goal is **fail**.

Step 3: The **fail** statement is a built-in Prolog predicate that immediately fails and forces backtracking. The cut, however, blocks it. This prevents Prolog from attempting any other **factorial** predicates. We are glad about that-- if we got down to the third **factorial** rule with the goal **factorial(-1,R)**, Prolog would go into an infinite loop.

Step 4: The cut-fail combination says that no solution to **factorial(-1,R)** exists. In a larger program, Prolog would backtrack to the clause that originated the **factorial(-1,R)** call, and continue its search. In this simple program, however, Prolog just responds ***no***.

The cut-fail combination successfully says, "if N < 0, no solution exists; therefore, we should stop trying to satisfy the **factorial** goal."

D. Exercises

4.3.1 The **max** relation determines the maximum between two numbers:

```
max(X,Y,X):- X > Y.
max(X,Y,Y):- X =< Y.
```

Change **max** relation by adding cut(s) at the appropriate place(s). In addition, add a third **max** predicate that uses negation as failure if X and Y are equal.

4.3.2 Develop the relation "add without duplication." This relation, denoted **add(X,L1,L2)**, adds item X to list L1 with the resulting list L2, *provided that item X is not already in L1*. For example, the add relation has the following dialogue:

```
?- add(a,[b,c,d],L2).
   L2 = [a,b,c,d] ;
   no

 ?- add(b,[b,c,d],L2).
   L2 = [b,c,d] ;
   no
```

4.3.3 Consider the program for the relation **quicksort**. This relation takes in a list of numbers and outputs a new list with the numbers in an ascending order. The program is:

```
quicksort([H|T],Answer):-
          split(T,H,List_of_littles,List_of_bigs),
          quicksort(List_of_littles,L),
          quicksort(List_of_bigs,B),
          append(L,[H|B],Answer).
quicksort([ ],[ ]).

split([H1|T1],Y,[H1|L],B):-
          H1 <= Y,
          split(T1,Y,L,B).
split([H1|T1],Y,L,[H1|B]):-
          H1 > Y,
          split(T1,Y,L,B).
split([ ],_,[ ],[ ]).
```

```
append([],L,L).
append([H1|T1],L,[H1|T3]):-
        append(T1,L,T3).
```

We ask the question

```
?- quicksort([6,1,9,3,10,7,8,2,5,4],X).
```

and Prolog responds:

```
X = [1,2,3,4,5,6,7,8,9,10]
```

Based on the **quicksort** relation, answer the following questions.

a. Is **quicksort** deterministic or not? What happens if we backtrack into **quicksort**?

b. Where would you place cuts in the program to improve its performance?

4.3.4 A sorting relation that is less efficient than **quicksort** is called the **permutation_sort**, defined as:

```
permutation_sort(X,Y):-
        permutation(X,Y),
                    ordered(Y).

        permutation(X,[H|T]):-
                    select(H,X,Y),
                    permutation(Y,T).
        permutation([ ],[ ]).

        select(X,[X|T],T).
        select(X,[H1|T1],[H1|T2]):-
                    select(X,T1,T2).

        ordered([_]).
        ordered([X,Y|T]):-
                    X =< Y,
                    ordered([Y|T]).
```

Like **quicksort**, **permutation_sort** takes in a list of numbers and outputs a list containing those numbers in an ascending order. Therefore, the question **?- permutation_sort([6,1,9,3,10,7,8,2,5,4],X).** results in the Prolog response: ***X = [1,2,3,4,5,6,7,8,9,10]***. Answer the questions below about **permutation_sort**.

a. Is **permutation_sort** deterministic or not? What happens if we backtrack into **permutation_sort**?

b. Where would you place cuts in the program to improve its performance?

4.3.5 Define the relation **difference_list(L1,L2,DL)** where L1, L2, and DL are all lists. The **difference_list** relation subtracts lists. It takes the elements of L2, and subtracts those that are in L1 from L1 to give DL. For instance, the question:

```
?- difference_list([a,b,c,d],[b,z,w,a],[c,d]).
```

is true. Define the **difference_list** relation. Use cuts where appropriate. Make sure that the relation is deterministic.

4.4 CONSEQUENCES OF THE CUT

Using the cut greatly enhances the performance of a program. It enables Prolog to solve problems faster and use less memory space. There are, however, some consequences to its use, as discussed below.

Á. Procedural Ordering Consequences

When cuts are used in a program, the procedural nature of Prolog plays a larger role. If multiple clauses exist with a cut, their ordering becomes critical to the program's performance.

As an example, assume that we have Prolog tied-in to our water analyzer monitoring the chemical in a process stream. Remember, we sound the alarm under two conditions:

```
sound_alarm(X,N,BadActorList):-
            member(X,BadActorList),
            !,
            N > 100.
sound_alarm(chlorine,_,_).
```

In the relation, **X** represents the detected chemical, **N** is its concentration in parts per million (ppm), and **BadActorList** is a predetermined list of toxins that are "bad actors." These two clauses say to sound the alarm if:

(X is a member of **BadActorList** and N is > 100)

OR

(X is not a member of **BadActorList** and X is **chlorine**)

Now if we reverse these two clauses, we get:

```
sound_alarm(chlorine,_,_).
sound_alarm(X,N,BadActorList):-
            member(X,BadActorList),
            !,
            N > 100.
```

The database now says to sound the alarm if:

(X is **chlorine**)

OR

(X is a member of **BadActorList** and N > 100)

Changing the order of these two clauses has changed the logical meaning of the program. The first relation will sound the alarm if **X** is **chlorine**, *provided* **X** is not a member of **BadActorList**. The second program will sound the alarm if **X** is **chlorine**, *regardless* of whether **X** is a member of **BadActorList**.

Changing the order of the clauses leads to an important point:

> *Because of the cut, the logical meaning of these two clauses changed when ordered differently.*

As we might recognize, using cuts increases the probability of errors. If we edit a program containing cuts, and change the order of clauses, we may unknowingly change the logical meaning of the program. When Prolog backtracks through an edited program, it may not perform accurately. When using the **cut**, we must be aware of procedural ordering consequences, or editing and adjusting the program may change the logical meaning.

B. Goal-Matching Consequences

In section 3.4 A, we introduced the **append** relation:

```
append([],L,L).
append([H1|T1],L2,[H1|T3]):-
      append(T1,L2,T3).
```

The **append** relation takes in lists L1 and L2, and combines them to form list L3. The **append** relation is nondeterministic. In section 3.4 A, we used **append** in different ways. The possible information flows were:

append(*input*,*input*,*output*)

append(*output*,*output*,*input*)

append(*input*,*output*,*input*)

Table 4.1 summarizes all the possible ways to use **append**. (This table is identical to Table 3.6 presented previously, repeated here for convenient reference.)

When we use a cut, the nature of the **append** relation changes, especially when trying to match goals. We can now make **append** deterministic by placing a cut on the base case:

```
append([],L,L):- !.
append([H1|T1],L2,[H1|T3]):-
      append(T1,L2,T3).
```

Everything is fine if we use the **append** relation to join two lists. In that case, the information flow is **append**(*input*,*input*,*output*). To the question:

```
?- append([b,a],[t,e],X).
```

Prolog correctly responds ***X* = *[b,a,t,e]***. As expected, backtracking generates no other solutions.

Now assume that we are in another place in the program. Here, we want to use the **append** relation to split a list, and to generate alternate solutions upon backtracking if required. We ask:

```
?- append(L1,L2,[b,a,t,e]).
```

Prolog responds:

```
L1 = []
L2 = [b,a,t,e]
```

If we ask for alternatives to this split, Prolog responds ***no***, because the cut on the base case has "frozen-in" the single solution.

Without the cut, Prolog responds with up to five alternate splits:

```
L1 = []  L2 = [b,a,t,e];
L1 = [b]  L2 = [a,t,e];
L1 = [b,a]  L2 = [t,e];
L1 = [b,a,t]  L2 = [e];
L1 = [b,a,t,e]  L2 = [];
no
```

Table 4.1. Variations of the append relation.

ARGUMENT STATUS IN APPEND RELATION			RESULTS AND EXAMPLES
FIRST	SECOND	THIRD	
input	*input*	*input*	Prolog answers ***yes*** or ***no***, e.g.
bound	bound	bound	**?- append([a,b],[c,d],[a,b,c,d]).** ***yes***
input	*input*	*output*	Instantiation of third argument:
bound	bound	free	**?- append([a,b],[c,d],L).** ***L = [a,b,c,d]***
input	*output*	*input*	Instantiation of second argument:
bound	free	bound	**?- append([a,b],L,[a,b,c,d]).** ***L = [c,d]***
output	*input*	*input*	Instantiation of first argument:
free	bound	bound	**?- append(L,[c,d],[a,b,c,d]).** ***L = [a,b]***
output	*output*	*input*	Divides third argument:
free	free	bound	**?- append(L1,L2,[a,b,c]).** ***L1 = [] L2 = [a,b,c] ;*** ***L1 = [a] L2 = [b,c] ;*** ***L1 = [a,b] L2 = [c] ;*** ***L1 = [a,b,c] L2 = [ ] ;*** ***no***
output	*input*	*output*	Puts anonymous variables in at beginning of the second argument:
free	bound	free	**?- append(L1,[a,b,c],L2).** ***L1 = [] L2 = [a,b,c] ;*** ***L1 = [_] L2 = [_,a,b,c] ;*** ***L1 = [_,_] L2 = [_,_,a,b,c] ;*** etc.
input	*output*	*output*	Puts anonymous variables in at the end of the first argument:
bound	free	free	**?- append([a,b,c],L2,L3).** ***L2 = [] L3 = [a,b,c] ;*** ***L2 = [_] L3 = [a,b,c,_] ;*** ***L2 = [_,_] L3 = [a,b,c,_,_] ;*** etc.

The bottom line is that when using the cut, we must know how the rule is being used in the program. If the rule is unexpectedly used in a different way, errors can result on goal matching.

Prolog programmers frequently place cuts into one of two classes:

- *green* cuts
- *red* cuts

A green cut does *not* change the declarative meaning of the program. A red cut *does* change the declarative meaning. Using red cuts makes the program more fragile. Such programs are normally more difficult to read and understand too. With a green cut, however, the program reads as is, since the declarative nature is unchanged.

As an example of green and red cuts, let us consider the **permutation_sort** procedure introduced in exercise 4.3.4:

```
permutation_sort(X,Y):-
            permutation(X,Y),
            ordered(Y).

permutation(X,[H|T]):-
            select(H,X,Y),
            permutation(Y,T).
permutation([ ],[ ]).
```

```
select(X,[X|T],T).
select(X,[H1|T1],[H1|T2]):-
            select(X,T1,T2).

ordered([_]).
ordered([X,Y|T]):-
            X =< Y,
            ordered([Y|T]).
```

The **permutation_sort** procedure inputs a list of numbers, and outputs a list containing the same numbers in an ascending order. For example:

```
?- permutation_sort([5,6,2,3,8,1],X).
   X = [1,2,3,5,6,8]
```

The same program with *green cuts only* is:

```
permutation_sort(X,Y):-
            permutation(X,Y),
            ordered(Y), !.

permutation(X,[H|T]):-
            select(H,X,Y),
                permutation(Y,T).
        permutation([ ],[ ]).

        select(X,[X|T],T).
        select(X,[H1|T1],[H1|T2]):-
                select(X,T1,T2).

        ordered([_]):- !.
        ordered([X,Y|T]):-
                X =< Y,
                ordered([Y|T]).
```

This program works using a *generate-and-test* technique which we shall discuss in detail in section 11.3C. The **permutation** predicate takes the input list and permutes it to another form. In essence, the **permutation** predicate *generates* a list that might be in the correct, sorted order.

The **order** predicate takes the list generated by the **permutation** relation. The **order** predicate then *tests* this permuted list to see if it is in an ascending order. If the list is not ordered correctly, **order** fails. Prolog then backtracks to the **permutation** relation, which is nondeterministic. The **permutation** relation permutes the list again, generating another trial list for the **order** predicate to test.

The **permutation_sort** relation relies on backtracking between the

permutation and **order** relations until a list is finally generated that is in the correct order. If this backtracking is improperly disrupted, the relation will not work correctly.

The program above has green cuts only. The declarative nature remains the same. Because both the **order** and **permutation_sort** procedures are deterministic, even in the absence of cuts, placing a cut at the end of these relations does not change their declarative meanings. These cuts improve the efficiency of the program while maintaining the declarative meaning, and are therefore green cuts.

As expected, Prolog respond properly to the question:

```
?- permutation_sort([5,6,2,3,8,1],X).
   X = [1,2,3,5,6,8]
```

Now, however, let us place a *red cut* into the program. We place it on the **select** base case to give:

```
permutation_sort(X,Y):-
            permutation(X,Y),
            ordered(Y).

permutation(X,[H|T]):-
            select(H,X,Y),
            permutation(Y,T).
permutation([ ],[ ]).

select(X,[X|T],T):- !.
select(X,[H1|T1],[H1|T2]):-
            select(X,T1,T2).

ordered([_]).
ordered([X,Y|T]):-
            X =< Y,
            ordered([Y|T]).
```

A red cut here changes the declarative meaning. The **select** predicate is a nondeterministic predicate required to permute a list. We make **select** deterministic through the cut. This cut eliminates the essential backtracking between the **permutation** and **order** relations. The cut "freezes-in" the first permutation of the input list, and prevents the **permutation** relation from generating alternate trial lists.

The declarative meaning of the program has changed. We ask the question:

```
?- permutation_sort([5,6,2,3,8,1],X).
```

and Prolog responds, incorrectly:

```
no
```

The red cut introduces an error into the program. The program relies on backtracking to succeed; the red cut eliminates the essential backtracking.

Logic programmers have historically frowned on the use of the cut, saying that it is "dirty," prone to logical errors, and it clouds the meaning of the program. They are largely correct. However, to use Prolog *efficiently* and *practically* for scientific and engineering applications (expert systems, for example), prudent use of the cut is essential.

C. Exercises

4.4.1 Develop the relation **delete_one(X,L1,L2)** that deletes only the first element X from the list L1 to give the list L2. For example the question:

```
?- delete_one(a,[a,b,c,a],L2).
```

yields the dialogue:

```
L2 = [b,c,a] ;
no
```

4.4.2 Add cuts to the remove duplicate relation, **remove_dup**, shown in Table 3.6, to improve the relation's performance.

4.5 CHAPTER SUMMARY

- The *cut* prevents backtracking. It is a useful tool for program control.

- When used properly, the cut improves program performance by: 1) speeding up the program by cutting out useless backtracking, and 2) reducing memory requirements by eliminating backtrack markers.

- The cut can make a relation *deterministic*. The relation then gives a single solution only.

- The cut can "lock-in" a single rule. This is useful when rules are mutually exclusive.

- The cut-fail combination can perform *negation as failure*. This application useful when we want to say "no solution exists, so we should stop looking."

- Careless use of the cut may cloud the meaning of the program. All uses of a rule containing a cut should be understood before proceeding with the program.

A LOOK AHEAD

With the close of this chapter, we have learned how to write correct, efficient Prolog programs. For a program to be useful, however, we need to be able to communicate with it. The next chapter, titled "Input and Output," addresses this issue.

REFERENCES

Bratko, Ivan, *Prolog Programming for Artificial Intelligence*, second edition, pp. 125-142, Addison-Wesley, Reading, MA (1990).

Clocksin, W.F. and C.S. Mellish, *Programming in Prolog*, third edition, pp. 70-92, Springer-Verlag, New York, NY (1987).

Garavaglia, Susan, *Prolog Programming Techniques and Applications*, pp. 116-135, Harper & Row Inc., New York, NY (1987).

Rowe, Neil C., *Artificial Intelligence Through Prolog*, pp. 238-241, Prentice-Hall, Englewood Cliffs, NJ (1988).

Sterling, L. and Shapiro E., *The Art of Prolog*, pp. 55-56, 157-173, MIT Press, Cambridge, MA (1986).

5

INPUT AND OUTPUT

This chapter covers input and output of terms, lists, characters, files, and even entire programs. Prolog implements input and output through built-in predicates. We saw the use of built-in predicates applied to mathematics in chapter 3. The built-in predicates for input and output discussed here are universal to almost all Prolog systems, though a specific version of Prolog, of course, may differ slightly.

5.1 READING AND WRITING TERMS

A. Input and Output Streams

Before we can fully discuss input and output, we need to understand the concept of a *stream* in Prolog. A program can read data from several input sources, known as *input streams*. Likewise, it can write to several output receptors, known as *output streams*.

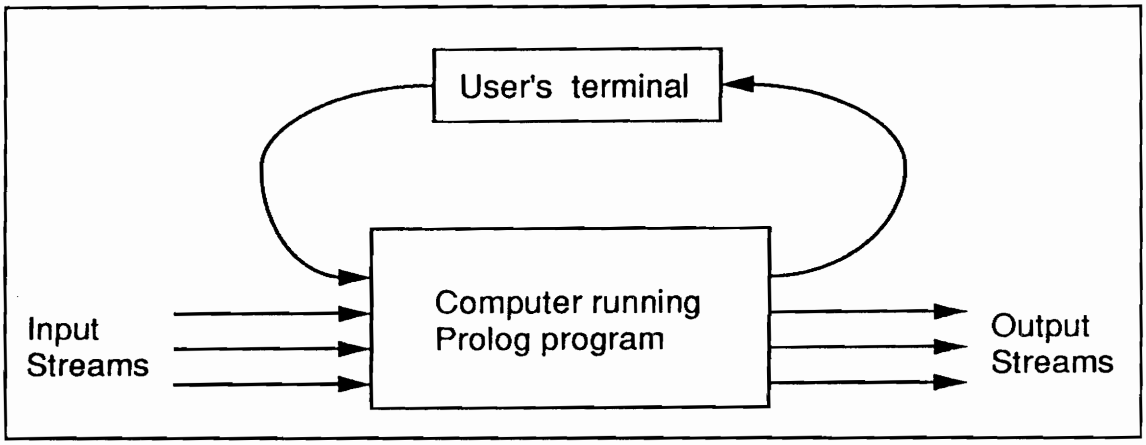


Figure 5.1. Input and output streams.

The user's terminal, for instance, could function as an input and an output stream. If Prolog is accepts a question from the screen, the screen is an input stream. If Prolog writes something to the screen, the screen is an output stream. The same is true for disk storage; the disk can function as an input or output stream, as shown in Figure 5.1.

While Prolog executes, it can receive input from only one stream at a time. Likewise, it can send output to only one stream at a time. The *current input stream* is the input stream currently active during execution. The *current output stream* is the output stream currently active during execution.

When Prolog is runs, the current input and output streams are typically the user's terminal. This mode allows interactive Prolog programs to be written and used. If necessary, we can switch the current input or output stream. For instance, let us consider a Prolog program that needs to access a database on the physical properties of chemicals. This database is stored on a computer disk. We would switch over the current input during execution to enable Prolog to read information from the disk. After Prolog has accessed the database, we would switch the current input stream back to the screen to accept input from the user.

B. The write, tab, nl (new line) and read Predicates

The built-in **write** predicate outputs an object to the current output stream. The goal:

```
write(X)
```

will write the object to which variable **X** is instantiated to the current output stream. The output will be in the same syntax that Prolog uses to display variables.

We use the **write** predicate to explain to the user aspects of program execution. For instance, suppose we wish to tell the user that no solution exists to a particular problem. To do this using the **write** predicate, we place our message inside the single quotation marks, ' ', to create the following Prolog statement:

```
write('No solution to this problem exists')
```

When Prolog encounters this predicate, we see

No solution to this problem exists

on the screen.

The **tab** is another built-in predicate, used to format output. The goal **tab(N)** instructs Prolog to output N blank spaces. The term **N** generally must be an integer (or a variable instantiated to an integer) for the predicate to succeed.

The **nl** predicate stands for "new line." It starts either input or output onto a new line. It has no arguments, and always succeeds.

The **read** predicate inputs an object from the current input stream. The goal:

```
read(X)
```

will read the next term from the current input stream. Importantly, Prolog will also *try to match the variable.* If X is an uninstantiated variable, it will be instantiated to the input term. When X is instantiated already, **read(X)** will succeed if there is a match and will fail if there is not. The **read(X)** procedure is deterministic; therefore, if **read(X)** fails, Prolog will *not* backtrack to another input term.

As an example of these predicates, we consider the program with corresponding dialogue in Table 5.1

Table 5.1. Program using the read, write, and nl (new line) predicates.

PROGRAM	RESULTS
go:- write('Please enter term:'),nl, read(X),nl, write('Please enter term again'),nl, read(X),nl, write('Term is: '), write(X). go:- write('**************************'), nl, write('Input error. Begin again.'), nl, write('**************************'), nl, go.	?- go. *Please enter term:* chemicals. *Please enter term again:* chemists. ************************** *Input error. Begin again.* ************************** *Please enter term:* chemicals. *Please enter term again:* chemicals. *Term is: chemicals* *yes*

We enter **go** and Prolog asks us to enter a term. We enter the term **chemicals**. Prolog asks us to enter the term again for confirmation. We enter a different term this time, **chemists**. But the variable **X** is already instantiated to **chemicals**, so when Prolog tries to match **chemicals** with **chemists**, this match obviously fails. Prolog then backtracks from the second **read** statement in the first **go** rule.

While backtracking, Prolog runs into the **read**, **write**, and **nl** predicates. Because each is deterministic, Prolog's only backtrack point is to the second **go** relation. In the second **go** relation, Prolog writes that there is an input error, then recursively begins the **go** process all over again.

The second time through the **go** relation, we correctly enter **chemicals** twice. This time, the **read(X)** statement matches **chemicals** with **chemicals**, and the procedure is successful. Prolog ends by answering ***yes***, indicating that the goal was successfully achieved.

5.2 WRITING LISTS

We can use the standard **write** predicate to write lists. The goal:

```
write([b,a,t,e])
```

succeeds and writes:

```
[b,a,t,e]
```

to the current output stream. But now let us consider a complex list **[[a,b,c],[d,e,f],[[g,h],[i,j]]]**. The goal:

```
write([[a,b,c],[d,e,f],[[g,h],[i,j]]])
```

results in:

```
[[a,b,c],[d,e,f],[[g,h],[i,j]]]
```

which is difficult to read.

Complex lists are difficult to read, and using the simple **write** statement does not alleviate the problem. We can, however, develop a procedure to output the elements more clearly. Let us call this the **flatten_write** procedure.

The purpose of **flatten_write** is to take the elements in a complex list and write them individually as atoms. The procedure is:

```
flatten_write(List):-                        % flatten_write rule
     flatten(List,FlatList),
     listwrite(FlatList).
flatten([Head|Tail],FlatList):-              % recursive flatten rule
     flatten(Head,FlatHead),
     flatten(Tail,FlatTail),
     append(FlatHead,FlatTail,FlatList).
flatten([],[]).                              % flatten base case no. 1
flatten(X,[X]).                              % flatten base case no. 2
listwrite([]).
listwrite([Head|Tail]):-
     write(Head),tab(2),
     listwrite(Tail).
```

The **flatten_write** procedure contains two steps. It first "flattens" the list using the **flatten** predicate. The **flatten** predicate takes a complex list, e.g.,

```
[[a,b,c],[d,e,f],[[g,h],[i,j]]]
```

and "flattens" it to give a standard list:

```
[a,b,c,d,e,f,g,h,i,j]
```

The rules of **flatten** relation are translated below.

Recursive Rule: To flatten a list, first flatten its head, flatten its tail, and then append these two flattened lists together to give the final output list.

Base Case No. 1: The flattened form of the empty list is the empty list.

Base Case No. 2: The recursive rule asks us to flatten the head of its input list. If we get to this point in the relation, i.e., base case no. 2, we realize that we have successfully flattened an element of the complex, nested input list all the way down to atomic form, **X**. Since we desire a list as output, create the simple list **[X]** as the output term.

Once the **flatten** procedure is complete, the **listwrite** procedure takes a standard list, e.g.,

```
[a,b,c,d,e,f,g,h,i,j]
```

and writes the individual elements to the screen:

```
a b c d e f g h i j
```

When writing lists to the screen, we can use either **listwrite** or **flatten_write**. We ask the question:

```
?- listwrite([[a,b,c],[d,e,f],[[g,h],[i,j]]]).
```

and Prolog responds:

```
[a,b,c]  [d,e,f] [[g,h],[i,j]]
```

If we want an even more simplified output, we ask:

```
?- flatten_write([[a,b,c],[d,e,f],[[g,h],[i,j]]]).
```

and Prolog responds:

```
a  b  c  d  e  f  g  h  i  j
```

We have successfully "flattened" the complex list and written the individual elements to the screen.

Note that the **flatten_write** procedure is nondeterministic and will produce garbage if we backtrack into the procedure. For instance, we ask:

```
?- flatten_write([[a,b,c],[d]]).
```

and Prolog correctly responds:

```
a  b  c  d
```

We force backtracking by entering the semicolon, and the next result is:

```
    a  b  c  d  []
```

This is not what is desired. To avoid problems upon backtracking, we may desire to place a cut at the end of the **flatten_write** procedure, e.g.,

```
flatten_write(List):-
        flatten(List,FlatList),
        listwrite(FlatList), !.
```

EXERCISES

5.2.1 Write the procedure **bar_graph(L)**, where L is a list. The **bar_graph** predicate takes in L, a list of integers, and prints different numbers of asterisks, i.e., *, corresponding to the integer value. Each element in the list has a row of asterisks printed on a new line. For instance, to the question:

```
?- bar_graph([1,3,4,10,5,2,1]).
```

Prolog responds:

```
*
* * *
* * * *
* * * * * * * * * *
* * * * *
* *
*
```

Write this procedure, emphasizing the **write** predicate.

5.2.2 Write the relation **square**, designed to print a square. It asks the user how big he wants the square to be. The user inputs a value. The **square** predicate then draws a square of asterisks with the side dimension given by the user. For example, the following is a typical dialogue of the **square** procedure:

```
?- square.
   Welcome to the square procedure.
   Please enter the side dimension:
   7.
   * * * * * * *
   *           *
   *           *
   *           *
   *           *
   * * * * * * *
   yes
```

Write this procedure using both the **read** and the **write** predicates.

5.3 READING AND WRITING CHARACTERS

For scientific and engineering applications of Prolog, the majority of reading and writing is done with atoms, numbers, and variables. Consequently, the **read** and **write** predicates that process these terms are the key workhorses. But Prolog does allow us to read and write *characters*. Characters are the smallest items that Prolog can process with built-in input and output predicates.

Prolog treats a character as an integer corresponding to its ASCII code (ASCII stands for "American Standard Code for Information Interchange"). The ASCII code assigns an integer between 0 and 255 to each character (see Appendix A for the entire ASCII code). For instance, the ASCII code 66 corresponds to the character B, while 67 corresponds to C, and ⌂ to 127.

A. The put Predicate

The goal **put(N)** *outputs* the character associated with the ASCII code number N to the current output stream. Therefore,

```
put(66),put(67)
```

outputs *BC*. In a similar fashion, the goal

```
put(104),put(105)
```

outputs ***hi***. The **put** predicate is deterministic. It has one solution, and if backtracked into, it cannot be re-satisfied. If the goal **put(N)** arises and the variable N is uninstantiated, an error results.

B. The get and get0 Predicates

Both **get(N)** and **get0(N)** *input* characters from the current input stream. The goal

```
get0(N)
```

reads the character from the input stream, and instantiates the variable N to the integer represented by its ASCII code. For example, if the character **g** is read by **get0(N)**, the variable N is instantiated to the integer 106.

One potential difficulty in using **get0** arises if the input stream has blank spaces. Blanks make an input stream more human-readable, but it is usually more convenient if blanks are not read by Prolog. The **get** predicate accomplishes this. The goal

```
get(N)
```

skips over all non-printable characters (i.e., characters with an ASCII code less than 33, including the blank space, which is ASCII code 32) and instantiates the variable N to the integer value associated with the character in the input stream.

Both **get** and **get0** are deterministic. They are only satisfied once, and cannot be re-satisfied upon backtracking. The input stream, therefore, cannot be reversed by backtracking.

C. Exercises

5.3.1 Write the program **run** to determines whether or not the next character in the current input stream is a letter, a number, or an alternate character (e.g., **&**, **#**, **@**, etc.). If the character is a letter, determine if it is capital or lower case.

5.3.2 Write the program **letter_change** to accept as input only characters that are letters. This program should also change the case of the letter. If the user inputs a capital **I**, the program generates a lower-case **i**. If the user inputs a lower-case **m**, the program generates an upper case **M**. Finally, the program outputs both characters to the screen, telling the user which one he had as input.

5.4 READING AND WRITING FILES

To read and write to files using the built-in predicates, we need to be able to change the current input and output streams. This section discusses built-in Prolog predicates used to adjust current input and output streams.

A. The see and seen Predicates

The **see** and the **seen** predicates change the current input stream. Sections 5.1 to 5.3 introduced built-in predicates that read information from the current input stream. When Prolog starts executing at the beginning of a session, the user's terminal is both the current input and current output stream by default.

The goal

```
see(Filename)
```

opens **Filename** for communication and switches the current input stream to **Filename**. In most versions of Prolog, an error, rather than a fail, occurs if **Filename** is an uninstantiated variable, or if **Filename** does not exist on the disk. In addition, the goal

```
see(user)
```

makes the user's terminal the current input stream.

The goal

```
seen
```

always succeeds. This predicate closes the current input stream, making it inactive and inaccessible for communication. The predicate then makes the user's terminal the current input stream.

Note that both **see(user)** and **seen** switch the current input stream over to the terminal. These two statements differ in that **seen** closes the file when it switches over to the terminal, while **see(user)** keeps the file open when switches over to the terminal.

Frequently, we see the sequence

```
. . .,
see(inputfile),
read(X),
seen,
. . .
```

This set switches the current input stream over to the disk file **inputfile**. Prolog then reads the first term, closes **inputfile**, and switches back over to make the terminal the current input stream. If we had the following sequence:

```
. . .,
see(inputfile),
read(X),
see(user),
. . .
```

the input stream **inputfile** would remain open.

B. The tell and told Predicates

Both the **tell** and **told** predicates change the current output stream. The goal

```
tell(Filename)
```

opens **Filename** for communication and converts the current output stream to **Filename**. If the file does not exist on the disk, Prolog will create the file. One caution here: if the file already exists on the disk, Prolog will *destroy* its contents by *overwriting* the file. If **Filename** is an uninstantiated variable, an error results. The goal

```
tell(user)
```

converts the current output stream to the terminal without closing any streams.

The goal

```
told
```

closes the current output stream, making it inactive and inaccessible for communication. The predicate then makes the user's terminal the current output stream. In many versions of Prolog, **told** also inserts an end-of-file marker on the file it is closing.

In exactly the same fashion as **see** and **seen**, the following sequence

```
. . . ,
tell(outputfile),
write(X),
told,
. . .
```

instructs Prolog to switch the current output stream over to the disk file **outputfile**. Prolog then writes the instantiated term **X** to the file, and finally reverts back to the terminal as the current output stream.

C. The seeing and telling Predicates

The **seeing** and **telling** predicates are used to diagnose the current input and output streams. The goal

```
seeing(Filename)
```

succeeds if Filename is instantiated and matches the current input stream. Under all other conditions, it fails.

The goal

```
telling(Filename)
```

succeeds if Filename is instantiated and matches the current output stream. Under all other conditions, it fails.

Consider the following program:

```
test_streams(Input,Output):-
      seeing(Input),
      telling(Output).
test_streams(_,_):-
      !, fail.
```

This programs tests whether the current input and output streams match the instantiated variables **Input** and **Output**, respectively. If they both match, the procedure is successful. If one or the other fails to match, the **cut-fail** combination in the second rule tells Prolog that no solution exists.

D. Exercises

5.4.1 Shown below are sequences of stream changing commands. Entering each sequence, Prolog has the terminal as both the current input and the current output stream. At the end of each sequence, what is the current input and output stream ?

a. ..., **see(user), tell(user),** ...

b. ..., **seen, told,** ...

c. ..., **X = file_1, see(X), see(file_2), see(X),** ...

d. ..., **X = file_1, tell(X), tell(file_2), told,** ...

e. ..., **X = file_1, telling(X), tell(file_2),** ...

5.4.2 Write a program called **input**. The **input** predicate initiates a procedure that:

(1) Makes sure the current output stream is the terminal.

(2) Asks the user for the name of the disk file (i.e., input stream) to be read.

(3) Converts the current input stream over to this user-defined disk file.

(4) Inputs the object in the current input stream, and writes the object on the user's terminal. After completing this, the program asks the user if any additional objects are to be read. The user can respond **y** or **n** (yes or no).

(5) If the user answers **y**, the program repeats step 4. If the user answers **n**, the program closes the current input stream and makes the terminal the current input stream.

5.5 READING PROGRAMS

We ultimately want to write Prolog programs and store all the relations on a file. Then at the appropriate time, we can instruct Prolog to read the file and apply these relations to our program. To accomplish this, we use the **consult** and **reconsult** predicates.

A. The consult Predicate

The **consult** predicate reads a program from a file and *adds its clauses* to the currently existing program in the database. The goal

```
consult(Filename)
```

instructs Prolog to read the program stored in Filename. Filename must be instantiated to the name of the file containing the clauses to be read. Prolog places all the clauses it reads at the end of the current database. Prolog then uses these clauses (in addition to the currently existing program in the database) to achieve goals.

The goal

```
consult(user)
```

is a special goal that allows Prolog to accept clauses from the terminal.

B. The reconsult Predicate

The **reconsult** predicate reads a program from a file and *replace clauses* in the currently existing program in the database with the clauses it has read. As with **consult**, the goal

reconsult(Filename)

instructs Prolog to read the program stored in Filename. Filename must again be instantiated to the name of the file containing the clauses to be read.

At this point, **reconsult** diverges from the **consult** predicate. Specifically, **reconsult**:

(1) takes the clauses read in Filename,

(2) eliminates all relations in the database that have been previously defined by these clauses, and

(3) supersedes these relations with the new relations defined in Filename.

If a clause has no previously defined relations in the database, **reconsult** introduces the new clause at the end of the database.

As an example, let us consider the relations **even_length(L)** and **odd_length(L)**. The **even_length** predicate succeeds if list L has an even length. In a similar fashion, the **odd_length** predicate succeeds if list L has an odd length. We define the original relation as:

```
even_length(X):-
        length(X,N),
        0 is N mod 2.

odd_length(X):-
        length(X,N),
        1 is N mod 2.

length([],0).
length([_|T],N):-
        length(T,TL),
        N is TL + 1.
```

But suppose we discover another, more efficient way to write the relations, and we desire to update the Prolog database using the **consult** and **reconsult** predicates. The new relations are:

```
even_length([ ]).
even_length([Head|Tail]):-
        odd_length(Tail).
odd_length([_]).
odd_length([Head|Tail]):-
     even_length(Tail).
```

Table 5.2 shows the Prolog database after using both the **consult** and **reconsult** predicates with these new relations. The **reconsult** predicate eliminates the **even_length** and **odd_length** clauses in the database, and replaces them with the new ones. The **consult** predicate simply adds the new relations to the end of database.

As another variation of **reconsult**, we can write

```
reconsult(user).
```

When the atom **user** is the argument, Prolog allows input from the terminal. Previously defined predicates will be eliminated and replaced with the new ones. This procedure is particularly useful for correcting program errors interactively.

Table 5.2. Effects of consult and reconsult on the Prolog database.

ORIGINAL DATABASE	ADDED RELATIONSHIPS

even_length(X):-
    length(X,N),
    0 is N mod 2.
 	
odd_length(X):-
    length(X,N),
    1 is N mod 2.
 	
length([],0).
length([_|T],N):-
    length(T,TL),
    N is TL + 1.
	even_length([ ]).
	even_length([Head|Tail]):-
	    odd_length(Tail).
	
	odd_length([_]).
	odd_length([Head|Tail]):-
	    even_length(Tail).

DATABASE AFTER ADDITION OF NEW RELATIONSHIPS USING	
consult	reconsult

even_length(X):-
    length(X,N),
    0 is N mod 2.
 	
odd_length(X):-
    length(X,N),
    1 is N mod 2.
 	
length([],0).
length([_|T],N):-
    length(T,TL),
    N is TL + 1.
 	
even_length([ ]).
even_length([Head|Tail]):-
    odd_length(Tail).
 	
odd_length([_]).
odd_length([Head|Tail]):-
    even_length(Tail).
	even_length([ ]).
	even_length([Head|Tail]):-
	    odd_length(Tail).
	
	odd_length([_]).
	odd_length([Head|Tail]):-
	    even_length(Tail).
	
	length([],0).
	length([_|T],N):-
	    length(T,TL),
	    N is TL + 1.

C. Exercises

5.5.1 We define the following relation, **equal_lists(X,Y)** that tests if list **X** is equal to list **Y**:

```
equal_list([],[]).
equal_list([HX|TX],[HY|TY]):-
            HX = HY,
            equal_list(TX,TY).
```

Assume we have this **equal_list** relation in the current program database. To optimize the program, this same relation can be improved by using a new, more efficient **equal_list** relation:

```
equal_list(X,X).
```

This new **equal_list** relation is in a file, and we desire to add it to the current Prolog program database.

a. What will be the Prolog database after we **consult** the file containing the new **equal_list** relation?

b. What will be the Prolog database after we **reconsult** the file containing the new **equal_list** relation?

5.5.2

We have the following clauses in two separate files as listed in Table 5.3.

Table 5.3. Examples of clauses in two input files.

<table>
<tr><th>CLAUSES IN FILE input_1</th><th>CLAUSES IN FILE input_2</th></tr>
<tr><td>factorial(N,_):- N < 0, !, fail.
factorial(0,1):- !.
factorial(N,Nf):-
    NN is N - 1,
    factorial(NN,NNf),
    Nf is NNf * N.

append([],L,L):- !.
append([H|T1],L2,[H|T3]):-
    append(T1,L2,T3).</td><td>factorial(0,1):- !.
factorial(N,Nf):-
    NN is N - 1,
    factorial(NN,NNf),
    Nf is NNf * N.

append([],L,L).
append([H|T1],L2,[H|T3]):-
    append(T1,L2,T3).</td></tr>
</table>

Assuming that our current Prolog database is empty, how will Prolog respond to the following questions?

a. ?- consult(input_1), factorial(7,X).

b. ?- consult(input_1), consult(input_2), factorial(-1,X).

```
c. ?- consult(input_1), reconsult(input_2), factorial(-1,X).

d. ?- consult(input_2), reconsult(input_1), append(L1,L2,[a,b,c]).

e. ?- consult(input_2), consult(input_1), append(L1,L2,[a,b,c]).

f. ?- consult(input_1), consult(input_2), append(L1,L2,[a,b,c]).
```

5.6 CHAPTER SUMMARY

- Prolog reads from the *current input stream* and writes to the *current output stream.*

- Prolog uses built-in predicates to read and write.

- Table 5.4 summarizes the predicates used for input and output to the current input and output stream:

Table 5.4. Summary of input and output predicates.

PREDICATE	DESCRIPTION
read(X)	input the next term from the current input stream
write(X)	output the next term to the current output stream
put(Integer)	output the character corresponding to the **Integer** ASCII code
get0(Integer)	input the next character as an **Integer** ASCII code
get(Integer)	input the next *printable* character as **Integer** ASCII code
consult(File)	read **File** and *add* clauses to the program database
reconsult(File)	read **File** and *supersede* clauses in the program database

- The **tab** and **nl** predicates are used for formatting.

- Table 5.5 summarizes the predicates for changing or assessing the current input and output stream:

Table 5.5. Summary of predicates for changing input and output streams.

PREDICATE	DESCRIPTION
see(File)	make **File** the current input stream
tell(File)	make **File** the current output stream
seen	close the current input stream and make the terminal the current input stream
told	close the current output stream and make the terminal the current output stream
seeing(X)	succeeds if **X** is instantiated to current input stream
telling(X)	succeeds if **X** is instantiated to current output stream

A LOOK AHEAD

We can now write Prolog programs and communicate efficiently with them. We are well on our way to becoming proficient enough to do artificial intelligence programming. The next chapter investigates more built-in Prolog tools.

REFERENCES

Bratko, Ivan, *Prolog Programming for Artificial Intelligence*, second edition, pp. 143-160, Addison-Wesley, Reading, MA (1990).

Burnham, W.D. and A.R. Hall, *Prolog Programming and Applications*, pp. 53-60, Halsted Press, a division of John Wiley & Sons, New York, NY (1985).

Clocksin, W.F. and C.S. Mellish, *Programming in Prolog*, third edition, pp. 93-105, Springer-Verlag, New York, NY (1987).

Garavaglia, Susan, *Prolog Programming Techniques and Applications*, pp. 159-182, Harper & Row Inc., New York, NY (1987).

Schnupp, P. and L. Bernhard, *Productive Prolog Programming*, pp.147-161, Prentice-Hall, Englewood Cliffs, NJ (1986).

Sterling, L. and Shapiro E., *The Art of Prolog*, pp. 175-178, MIT Press, Cambridge, MA (1986).

6

BUILT-IN TOOLS

This chapter introduces some *built-in* tools available in Prolog. Built-in tools are predicates with a predefined meaning. We do *not* have to specify these relations in our program. Prolog will automatically recognize them and process them according to their definitions.

We start in section 6.1 by discussing both built-in and user-defined operators. In section 6.2, we discuss tools for handling various types of equality. Section 6.3 covers predicates used for program control. We review the **cut** and **fail**, and introduce additional control relations. In section 6.4, we present three built-in predicates used to retain multiple solutions to given goals. We close the chapter with a summary in section 6.5.

6.1 BUILT-IN AND USER-DEFINED OPERATORS

A. Types of Operators

We introduced operators in chapter three. Operators are merely functors of arity one or two that we write in different ways to improve a program's appearance. Operators generally take no action on their own. The three types of operator notation are *prefix*, *infix*, and *postfix*, named for their location. Recall that a prefix operator appears in front of the arguments (**+ (J,1)**), an infix operator appears between the arguments (**J + 1**), and a postfix operator appears after the arguments (**(J,1) +**).

The **+** is an example of a *built-in operator*, meant to be used in prefix notation. Prolog has several built-in operators, including:

(1) arithmetic operators, e.g., **+**, **-**, *****, **/**

(2) equality operators, e.g., **=**, **is**, and **=:=**.

The principle of *precedence* is an important aspect of operators. As mentioned in section 3.1A, we may interpret the statement **2 + X * Y** two ways, as 2 + (X * Y) or (2 + X) * Y. But, based on the precedence of the + and * operators, we perform the multiplication first, thus interpreting the statement as 2 + (X * Y). The principle of precedence allows Prolog to clearly and unambiguously interpret a statement. We may also use parenthesis to control precedence and avoid confusion.

Prolog also allows us to declare our own operators. When we do, we are specifying *user-defined operators*. User-defined operators must be explicitly declared in the program before Prolog can handle them. In contrast, Prolog *automatically* recognizes *built-in operators* without any explicit declarations.

B. Writing User-Defined Operators

Consider the statement **is_a(benzene,chemical)**. We can convert the functor is_a into a user-defined infix operator. This allows us to write the same statement as **benzene is_a chemical**. Here, **is_a** is the functor, and is a special type of functor, called an *operator*. The objects **benzene** and **chemical** are arguments of the functor, and are called *operands*.

To make **is_a** an operator, we must tell Prolog to treat it as such. We need to fully specify the operator, so the program will know how to treat the operator and its operands. To fully specify an operator, we must declare the following:

(1) the *precedence* of the operator itself relative to other

operators;

(2) the *class* of the operator, i.e., its notation, either prefix, infix, or postfix; and

(3) the *type* of operator, defined by the precedence of each operand relative to the operator.

To declare the operator, we use the built-in Prolog predicate **op**. The syntax for **op** is:

op(*precedence*,*class_&_type*,*operator*),

e.g.,

```
op(1200,xfx,':-').
op(1200,fx,[:-,?-]).
op(1100,xfy,';').
op(1000,xfy,',').
```

where:

- *precedence* is a number between 0 and 1200 that quantitatively identifies the precedence of the operator. The lower the number, the lower is the precedence of the operator.

- *class_&_type* declares both the class and type of operator. We use the symbols **f**, **x**, and **y** to declare the class and type. The declaration options open to us are: **fx**, **fy**, **xfx**, **xfy**, **yfx**, **yf**, or **xf**. They are discussed in section 6.1B.1 below.

- *operator* is a character or string of characters denoting the operator, e.g., the character + for the addition operator.

The required specifications of *precedence*, *class and type*, and the *operator* itself in the **op** predicate are discussed below.

1. Class and Type of Operators

To specify the class (notation) and type of operator, we use the symbols **f**, **x**, and **y**, where **f** represents the functor being declared an operator, and **x** and **y** are operands.

For instance, the statement

fx

says that we are referring to a prefix operator, since the functor **f** comes before the operand, **x**. The **x** signifies the operand precedence; it means that the operand has a *lower* precedence than the operator.

We can write a statement denoting a prefix operator where the operand has an *equal or lower* precedence than the operator. We write:

fy

This notation again denotes a prefix operator, since **f** comes before **y**. The **y** means that the operand can have an equal or lower precedence than the operator.

For example, let us consider the minus sign, -, an **fx** operator in Prolog. We write:

```
- 2.
```

The functor is the atom -, and its argument is the number 2. The formal way to write the expression in Prolog is:

```
-(2).
```

The - is an **fx** operator, meaning that the operator (the minus sign) is a prefix operator, and the operand (the integer 2) has a lower precedence than the operator itself.

We can also write the minus sign - as an infix operator of the type **yfx**. We write:

```
A - B.
```

which is formally written in Prolog as:

- (A, B).

Here, the minus sign functions as an infix operator, with the operand on the right-hand-side, **B**, of strictly lower precedence than the operator, and the operand on the left-hand-side, **A** of equal or lower precedence than the operator.

We can apply the f, **x**, and **y** notation for all operators, including prefix, infix, and postfix operators that have an arity of one or two. Table 6.1. summarizes the seven possible combinations.

Table 6.1. Precedence declarations for operators.

Class	Type	Precedence of Operand Relative to Operator	Example
prefix	fx	Operand has lower precedence	- 2
	fy	Operand has equal or lower precedence	~ 2
infix	xfx	Both operands have lower precedence	2 + 4
	xfy	Left operand has lower precedence; Right operand has equal or lower precedence	X & Y
	yfx	Left operand has lower or equal precedence Right operand has lower precedence	X - Y
postfix	xf	Operand has lower precedence	X !!
	yf	Operand has equal or lower precedence	X @

2. Precedence

We are getting closer to being able to fully specify an operator using the op statement. We know that we must declare its class and type, but we also need to declare its *precedence*. What is precedence? Why do we need it? What does it do for us? Answering these questions is essential to truly understanding operators.

First, let us address the question of what precedence is. Precedence relates to how *tightly* an operator binds its operands; that is, precedence determines which action should be taken first in a statement with multiple operators. Consider the statement introduced previously:

```
2 + X * Y
```

Precedence determines whether we treat this statement as (2 + X) * Y or 2 + (X * Y). Typically, we do multiplication first, and treat the statement as:

```
2 + (X * Y)
```

Comparing the operators + and *, we see that the * is more tightly binding than +, since the multiplication is performed first.

In Prolog, *the lower the precedence, the more tightly binding is the operator*. The multiplication operator * has a lower precedence than the addition operator +. Therefore, * is more tightly binding than +, and we understand **2 + X * Y** to mean **2 + (X * Y)**.

Another way to view precedence is to look at the principal functor. The statement **2 + X * Y** is understood to mean in Prolog:

```
+ (2, * (X, Y))
```

The + operator is the principal functor. In a Prolog statement with operators, the principal functor is the operator with the *highest* precedence, and therefore is the one that *binds the least tightly*.

Why do we need precedence? What does it do for us? Any statement with multiple operators requires precedence to tell Prolog what action to take next. Without precedence, a statement with multiple operators could be interpreted many ways. The answer resulting from the statement depends on which action Prolog takes first. Precedence eliminates this pitfall, making Prolog consistent and reliable.

3. Example of a User-Defined Operator

We can now fully specify an operator using the **op** predicate defined earlier. For example, we wish to develop an operator that says "X is approximately equal to Y". We denote this operator by the atom **~=**. We enter the statement:

```
?- op(700,xfx,~=).
```

Prolog responds ***yes*** to indicate that this user-defined operator is successfully declared. The **op** statement says "declare the operator ~= as an infix operator of the type xfx (i.e., both operands have a lower precedence than the operator) with an operator precedence of 700." As mentioned, Prolog gives an operator precedence scale of 0 to 1200. The lower the number, the lower is the precedence. A precedence of 700 is a mid-range precedence value.

4. Predefined Operators

We enter the **op** predicate to declare *user-defined* operators. In addition, Prolog has built-in, *predefined* operators. These predefined operators have corresponding **op** statements. Unlike the **op** statements for user-defined operators though, the **op** statements for predefined operators are built into Prolog, and do not need to be declared explicitly. Figure 6.1 summarizes all the predefined operators in Prolog.

Let us investigate how we can use Figure 6.1 in a practical way. We consider the statement:

```
2 * 3 + 4 =:= 8 + 4 / 2
```

The statement has the following operators, each with their associated

```
op(1200,xfx,':-').
op(1200,fx,[:-,?-]).
op(1100,xfy,';').
op(1000,xfy,',').
op(700,xfx,[=, is, <, >, =<, >=, ==, =\=, \==, =:=]).
op(500,yfx,[+, -]).
op(500,fx,[+, -, not]).
op(400,yfx,[*, /, div]).
op(300,xfx,mod).
```

Figure 6.1. Predefined operators in Prolog.

precedence (in parenthesis): * (400), / (400), + (500), and =:= (700). The =:= has the highest precedence and therefore, is the principal functor. We can now write the statement as:

```
=:=((2 * 3 + 4), (8 + 4 / 2))
```

Looking inside the above statement, we see two statements:

- (2 * 3 + 4)
- (8 + 4 / 2)

The + has a precedence of 500, while the * and / both have a precedence of 400. Therefore, the +, with the highest precedence, is the principal functor of each statement. We write:

```
=:=( +((2 * 3), 4), +(8, (4 / 2)))
```

Having followed the operator precedence, we now have the final form of the Prolog statement. This form represents the way that Prolog internally stores the statement. Using this final form, Prolog can numerically evaluate the expression unambiguously. Prolog goes through the following stages, evaluating the tightest-binding operators first:

Original statement:	**=:=(+((2 * 3), 4), +(8, (4 / 2)))**
After evaluation 1:	**=:=(+(6, 4), +(8, 2))**
After evaluation 2:	**=:=(10, 10)**
Answer:	***yes***

C. Exercises

6.1.1 How does Prolog interpret the expression **X - Y - Z** if the operator - is:

a. A **yfx** operator?

b. A **xfy** operator?

6.1.2 Declare the following operators with the correct op predicate.

a. The "approximately equal to" operator **~=**, used in the syntax **X ~= Y**. The precedence of **~=** should be the same as the built-in **=** predicate.

b. The "much greater than" operator >>, used in the syntax **X** >> **Y**. The precedence of >> should be the same as the built-in comparison operator >.

c. The **is_a** operator used in the syntax **benzene is_a chemical**. What is a proper precedence for **is_a**? Why?

6.1.3 Consider the following operator declarations:

```
op(300, xfy, and).
op(350, xfx, is_a).
```

Are the following statements syntactically correct? How are they understood by Prolog? What is the principal functor?

a. **benzene is_a hydrocarbon and aromatic_compound.**

b. **phenol is_a alcohol and aromatic_compound and organic_compound.**

c. **benzene and cumene is_a hydrocarbon and aromatic_compound.**

6.2 EQUALITY

Prolog has a number of built-in tools to assess equality between terms. We discussed different types of equality in section 3.1A, with the use of =, is, and =:=. This section reviews those and introduces several other forms of equality.

The statement

```
X = Y
```

is true if X and Y match. The statement involves no arithmetic evaluation. In addition, Prolog's unification (matching) mechanism will attempt to instantiate any free variables to make the terms identical.

In the same fashion, the statement

```
X \= Y
```

succeeds if X and Y do not match. When the \= predicate succeeds, the instantiations of X and Y do not change.

The statement

```
X is Y
```

is true if X matches the numerical value of the arithmetic expression of

Y. Prolog performs numerical evaluation and matching. If X is a free variable, Prolog instantiates it to a number. Some versions of Prolog use the syntax := instead of is. Thus

```
X is Y  and
X := Y
```

are equivalent statements.

The statement

```
X =:= Y
```

is true if the numerical values of arithmetic expressions X and Y are equal. Likewise, the statement:

```
X =\= Y
```

is true if the numerical values of the arithmetic expressions X and Y are not equal. Neither the =:= nor the =/= operator performs instantiation.

Another type of equality, not yet discussed, is the *literal* equality between terms. The statement

```
X == Y
```

is true if X and Y are *identical*. It is a stricter equality test than the = predicate. In particular, the names of free variables must be the same. The following dialogue shows some examples:

```
?- X == Y.
   no
?- X == X.
   yes
?- fuel(methane,hydrocarbon) == fuel(methane,X).
   no

?- fuel(methane,X) == fuel(Y,hydrocarbon).
   no
?- fuel(methane,hydrocarbon)==fuel(methane,hydrocarbon).
   yes
```

The counterpart to identical equality is the statement that succeeds if X and Y are *not* identical. We write:

```
X \== Y
```

We get the following dialogue:

```
?- X \== Y.
```

```
    yes
?- X \== X.
    no
```

6.3 PROGRAM CONTROL TOOLS

A. The Comma and Semicolon Operators

We use the **,** and **;** operators when we write goals in Prolog. The comma represents an *and*, and the semicolon represents an *or*. The comma signifies a *conjunction* of goals. The statement

```
X , Y
```

succeeds only if both X *and* Y succeed. Note that if X succeeds and Y fails, Prolog will backtrack to X in an attempt to satisfy the goal.

The semicolon operator represents a *disjunction* of goals. The statement

```
X ; Y
```

succeeds if either X *or* Y succeeds. Importantly, the **;** operator *can be used to combine separate clauses into one statement*. The statement:

```
flammable(X):-
      petroleum_product(X);
      wood_product(X);
      coal_derivative(X).
```

says "X is flammable if X is a petroleum product, *or* a wood product, *or* a coal derivative." It combines three Prolog rules into a single statement. An equivalent set of Prolog rules is:

```
flammable(X):-
     petroleum_product(X).
flammable(X):-
     wood_product(X).
flammable(X):-
     coal_derivative(X).
```

The semicolon operator reduces the program from three clauses to one. Although the semicolon can be convenient, we should avoid using it excessively, because it can make programs increasingly difficult to read and understand. If a logical *or* is desired, writing separate rules is usually best.

B. The cut Predicate

The cut, denoted by the exclamation mark, !, prevents backtracking. It eliminates or "cuts" backtracking between itself and the parent goal, as discussed in detail in Chapter 4. Using the cut properly can:

(1) speed up a program by eliminating useless backtracking that

wastes time; and

(2) reduce the amount of computer memory required by eliminating backtrack markers.

On the negative side, the cut may make the program more error-prone, since the cut is predominantly a procedural tool.

C. The fail Predicate

The **fail** predicate always fails and forces backtracking, as explained in section 4.3C in the context of the **cut-fail** combination. We used **fail** to say "if you get to this point, no solution exists, so stop searching and start backtracking."

For example, consider the **factorial** relation studied in section 4.3C.

```
factorial(N,_):-
     N < 0,
     !,
     fail.
factorial(0,1):- !.
factorial(N,R):-
     NN is N - 1,
     factorial(NN,TemporaryR),
        R is TemporaryR*N.
```

The first clause uses the **fail** predicate. It says "If N is less than zero, no factorial exists, so stop searching for the answer and start backtracking."

We can also use the **fail** predicate to force Prolog to automatically backtrack through all solutions. Consider the program below:

```
split_all(List):-
            append(L1,L2,List),
            write('L1= '),write(L1),
            write('  L2= '),write(L2),nl,nl,
            fail.
split_all(_).
```

When the **append** relation is called, the variable **List** is instantiated, while **L1**, and **L2** are free variables. Therefore, **append** will decompose **List** into two lists **L1** and **L2**. Upon backtracking, **append** repeatedly splits **List** differently. The **fail** predicate forces backtracking, thereby making the **append** generate all possible ways to split **List**.

If we ask the following question:

```
?- split_all([benzene,methane,ethanol]).
```

Prolog responds:

```
L1 = []  L2 = [benzene,methane,ethanol] ;
L1 = [benzene]  L2 = [methane,ethanol]  ;
L1 = [benzene,methane]   L2 = [ethanol] ;
L1 = [benzene,methane,ethanol]  L2 = [] ;
yes
```

We use the **append** procedure to split the list. Knowing **append** is nondeterministic, we simply need to backtrack into **append** to generate alternate solutions. The **fail** predicate accomplishes this backtracking.

D. The true Predicate

The **true** predicate always succeeds. It is not really needed in Prolog, and exists for largely convenience. If clauses are placed in the proper order, the **true** predicate is unnecessary.

For example, consider the following program:

```
input:-
    write('Please enter input term'),nl,
    (read(X) ; true),
    write('Input term is'),nl,
    write(X),
          enter_rest_of_program.
```

The **true** predicate here conveniently avoids a failure of the **input** clause even if the **read(X)** goal fails. We can avoid using **true** and still get the same program by writing:

```
     input:-
          write('Please enter input term'),nl,
          read(X),
          write('Input term is'),nl,
          write(X).
          enter_rest_of_program.

     input:-
          write('Input term is'),nl,
          write(X),
          enter_rest_of_program.
```

Clearly, the inclusion of **true** makes the programming more convenient.

E. The call Predicate

The goal

```
call(X)
```

succeeds if an attempt to satisfy **X** succeeds. The predicate assumes that **X** is instantiated and treats **X** as a goal. Typically, **X** is instantiated to a structured object. Suppose we wish to test if **Y** is a member of list **L**. We write

```
member(Y,L).
```

An equivalent statement is:

```
call(member(Y,L)).
```

Here, **X** is instantiated to the structured object, **member(Y,L)**.

Initially, it appears that the **call** predicate is unnecessary. If we want the goal **member(Y,L)**, it is easier to execute **member(Y,L)** rather than **call(member(Y,L))**. We use **call(X)** in Prolog when we want to make variables actual sub-goals. For example, consider the following rule:

```
execute(X,Y):-
      X, Y.
```

This relation will not succeed on most Prolog systems. The reason? We made variables **X** and **Y** sub-goals without using the **call** predicate. Therefore, Prolog treats **X** and **Y** as plain variables rather than goals to be satisfied. Using the **call** predicate makes the statement syntactically correct. The way to write **execute** rule so that it will run on most Prolog systems is:

```
execute(X,Y):-
        call(X), call(Y).
```

Here, the **call** predicate transforms variables **X** and **Y** into bona-fide sub-goals that Prolog can recognize.

F. The not Predicate

The statement

```
not(X)
```

succeeds if an attempt to satisfy X fails. The predicate assumes that X is instantiated to a clause and treats that clause as a goal. Using the **member** relation for lists, the goal

```
not(member(a,[b,t,e]))
```

succeeds, while

```
     not(member(a,[b,a,t,e]))
```

fails.

The **not** predicate can replace the **cut-fail** combination to make the programs easier to understand. For example, consider a Prolog program managing the inventory in a warehouse. When a new shipment comes in, Prolog will accept it and perform financial accounting procedures in a normal fashion, provided that the material is not hazardous or explosive. The program is as follows:

```
     accept(Material,PropertiesList):-              % cut-fail program
                member(hazardous,PropertiesList),
                !, fail.
     accept(Material,PropertiesList):-
                member(explosive,PropertiesList),
                !, fail.
     accept(Material,PropertiesList):-
                .
                .
                (financial accounting clauses)
```

The first clause uses the **cut-fail** combination to say "if the material received is hazardous (i.e., **hazardous** is a member of **PropertiesList**), we cannot use these financial accounting procedures. Therefore, we use **cut-**

fail to stop trying to satisfy the goal." Similarly, the second clause stops trying to satisfy the goal if the material entering the warehouse is explosive.

By using the **not** predicate, the cut-fail procedure can be eliminated:

```
accept(Material,Status):-
            not(member(hazardous,PropertiesList)),    % not program
            not(member(explosive,PropertiesList)),
            .
            .
            (financial accounting clauses)
```

This program says, "Perform financial accounting procedures if the material is neither hazardous nor explosive." If **explosive** is not a member of **PropertiesList**, the goal **member(explosive,PropertiesList)** fails. We now have the goal **not(member(explosive,PropertiesList))**, where the **member** goal has failed. Since **not(X)** succeeds when goal **X** fails, the goal **not(member(explosive, PropertiesList))** succeeds. The same procedure occurs if **hazardous** is not a member of **PropertiesList**. Prolog then executes financial accounting procedures, since the material is neither hazardous nor explosive.

The *not program* performs exactly the way the *cut-fail program* does. It will fail and backtrack if either **hazardous** or **explosive** is a member of

PropertiesList. Otherwise, the shipment will be accepted and normal accounting procedures will be used. Moreover, the not program is much easier to understand.

The **not** relation is built-in to Prolog, and therefore, we do not have to include its definition in our program. To better understand the **not** relation, however, it is useful to identify the Prolog rules that specify how **not** behaves:

```
not(X):-
     call(X), !, fail.
not(_).
```

The first relation (a rule) says "We want the clause **not(X)** to succeed if goal **X** fails. Conversely, if goal **X** is successful, then **not(X)** must fail. Therefore, test if **X** is true. If it is, then **not** must fail so implement the **cut-fail** combination to force immediate backtracking."

The second relation (a fact) says "We know from the previous clause that the goal **X** failed. If goal **X** was successful, the previous clause would implement a cut-fail combination and we would not reach the second **not** predicate. Since goal **X** has indeed failed, we want **not(X)** to be successful. The fact **not(X)** succeeds with no additional conditions required."

By understanding this implementation of **not**, we can more easily understand why the **not** predicate can replace the **cut-fail** combination.

G. The repeat Predicate

The **repeat** predicate is a built-in tool used to control backtracking. It repeats a procedure that Prolog has just backtracked from. When Prolog is backtracking and hits **repeat**, backtracking immediately halts. The **repeat** predicate succeeds, and Prolog moves on to satisfy additional sub-goals.

The **repeat** relation is built-in to Prolog, and therefore, we do not have to define it in our program. To better understand the **repeat** relation, however, it is useful to identify the Prolog rules that specify how **repeat** behaves:

```
repeat.
repeat:- repeat.
```

The first predicate says "Succeed every time and do nothing." The second predicate controls backtracking. It recursively calls the **repeat** procedure again, knowing that the first **repeat** predicate will simply succeed and do nothing. This step allows repetition of a process that previously failed.

The **repeat** predicate can be used in input and output to make a program more robust. Consider the following program:

```
sign_on(ID):-
      repeat,
      write('Enter your userid'),nl,
        read(ID),!.
```

When Prolog first enters the **sign_on** clause, it hits the **repeat** predicate. The **repeat** predicate succeeds and does nothing. Prolog moves on to the **write** predicate, and asks the user to enter his identification. The **read** statement then tests if the identification matches that instantiated to the variable **ID**. If they do not match, Prolog backtracks. The first backtrack point is the **repeat** relation. Prolog attempts to re-satisfy the goal **repeat**, and uses the second **repeat** relation:

```
repeat:- repeat.
```

This clause simply calls the **repeat** relation again. Prolog scans downward from the top, and runs into the

```
repeat.
```

statement. This statement immediately succeeds and does nothing. With the backtracked **repeat** goal now satisfied, Prolog moves forward to the **write** and **read** predicates again. The process of asking for identification and testing if it matches is *repeated*.

Table 6.2 summarizes the program control tools mentioned in this section.

Table 6.2. Built-in program control tools in Prolog.

ITEM	SYNTAX	DESCRIPTION	EXAMPLE
comma	X , Y	conjunction of goals; logical *and* (V is true if W *and* X *and* Y *and* Z are all true).	**V:- W, X, Y, Z.**
semicolon	X ; Y	disjunction of goals; logical *or* (V is true if W *or* X is true, and Y *or* Z is true)	**V:- (W;X),(Y;Z).**
cut	!	prevents backtracking	**V:- W, !, X.**
fail	**fail**	forces backtracking	**V:- W, X, fail.**
true	**true**	always true (takes no action)	**V:- (W;true), X.**
not	**not(X)**	succeeds if goal X fails	**not(member(X,L))**
repeat	**repeat**	repeats a procedure if backtracked into	**sign_on(ID):- repeat, read(ID).**
call	**call(X)**	makes variable X a syntactically callable subgoal	**not(X):- call(X), !,fail.**

H. Exercises

6.3.1 Write a program that will: 1) ask the user what question he wishes to have answered, and 2) pose the question to the Prolog database to test if it is true. If the question is true, use the **write** statement to tell the user. If the question is false, use the **write** statement to tell the user.

6.3.2 Develop the relation **screener(L1,L2,L3)**. List L1 is a list of candidate solvents to use in an extraction process. List L2 is a list of forbidden solvents. The **screener** relation takes all the candidate solvents in L1, and outputs them to list L3 as long as they are not a member of the forbidden solvent list L2. Use the **member** and **not** predicates to help you. Make sure the **screener** relation does not produce erroneous results upon backtracking. Do not use the cut anywhere in the relation.

6.4 RETAINING MULTIPLE SOLUTIONS

We saw in section 6.3C the use of the **fail** predicate to force backtracking and generate alternate solutions. An inconvenience in using **fail**, however, is that each time a new solution is generated, the previous solution is permanently lost. Frequently, we want to extract multiple solutions, i.e., generate and *retain* the solutions for further use. The three built-in predicates used to accomplish this are: **findall**, **bagof**, and **setof**. They are closely related.

A. The findall Predicate

The goal

```
    findall(X,G,L)
```

instantiates the free variable L to a list of all X's that satisfy the goal G. Consider the following database of flammable materials, denoted as *db6A*:

```
    flammable(methane,hydrocarbon).                % db6A
    flammable(ethane,hydrocarbon).
    flammable(propane,hydrocarbon).
    flammable(benzene,hydrocarbon).
```

```
flammable(methanol,alcohol).
flammable(ethanol,alcohol).
flammable(acetone,ketone).
```

The goal:

```
findall(X,flammable(X,hydrocarbon),List)
```

instantiates **List** to:

```
[methane,ethane,propane,benzene]
```

Note that **X** remains uninstantiated, since it is just a place marker. The X identifies the component of the functor **flammable** (in this case, the first component) that should be collected and retained in the final answer **List**.

The goal:

```
findall(X,flammable(X,aldehyde),List)
```

instantiates **List** to **[ ]**, since no variable X satisfies the goal **flammable(X,aldehyde)**.

Finally, the goal:

```
findall(X,flammable(X,Y),List)
```

instantiates **List** to:

```
[methane,ethane,propane,benzene,methanol,ethanol,acetone]
```

In this example, both variables **X** and **Y** remain uninstantiated.

B. The bagof Predicate

The **bagof** predicate is similar to the **findall** predicate, with one exception. The **findall** predicate forces backtracking through *all* solutions to the goal G and inserts each solution of X into a final list L. The goal

```
bagof(X,G,L)
```

will also instantiate the variable L to a list of all X's that satisfy goal G, but it does not do force backtracking. Instead, it requires user-directed backtracking. For example, the question posed to database *db6A* gives the following dialogue:

```
?- bagof(X,flammable(X,Y),List).

Y = hydrocarbon
```

```
List = [methane,ethane,propane,benzene];
Y = alcohol
List = [methanol,ethanol];
Y = ketone
List = [acetone];
```

Each backtrack is user-directed and therefore done *manually* by entering the semicolon. When alternate solutions are generated, previous ones are lost.

There is another difference between **bagof** and **findall**. When goal G has no solution, the **findall** predicate instantiates **List** to **[ ]**, while the **bagof** predicate *fails*. For example, the goal

```
findall(X,flammable(X,aldehyde),List)
```

instantiates **List** to **[ ]**, while

```
bagof(X,flammable(X,aldehyde),List)
```

fails, since no solution to **flammable(X,aldehyde)** exists.

C. The setof Predicate

The **setof** predicate resembles the **bagof** predicate in that backtracking is done manually, distinguishing between different solutions of goal G. The goal

```
setof(X,G,L)
```

instantiates the variable L to a list of X's that satisfy goal G; unlike **bagof** however, **setof** *orders* the elements of list L alphabetically, and removes any duplicate items.

For database *db6A*, the goal

```
setof(X,flammable(X,hydrocarbon),List)
```

instantiates **List** to:

```
List = [benzene,ethane,methane,propane,]
```

and the goal

```
setof(Y,flammable(X,Y),List)
```

will give:

```
List = [alcohol,hydrocarbon,ketone]
```

D. Exercises

6.4.1 Consider again database *db6A*:

```
flammable(methane,hydrocarbon).            % db6A
flammable(ethane,hydrocarbon).
flammable(propane,hydrocarbon).
flammable(benzene,hydrocarbon).
flammable(methanol,alcohol).
flammable(ethanol,alcohol).
flammable(acetone,ketone).
```

How will Prolog respond to the following questions?

```
a. ?- findall(Y,flammable(_,Y),L).
b. ?- findall(X,flammable(X,_),L).
c. ?- bagof(Y,flammable(X,Y),L).
d. ?- bagof(X,flammable(X,_),L).
e. ?- setof(Y,flammable(X,Y),L).
f. ?- setof(X,flammable(X,_),L).
```

6.4.2 Use **bagof** to write a procedure **split_all(L,Answer)**. The split_all relation inputs L, a list of materials to be separated and purified. The **split_all** relation generates **Answer**, a list of lists representing all possible overhead and bottoms products. For example, the question:

```
?- split_all([c2,c3,c4,c5],Answer).
```

gives the result:

```
Answer = [[[c2],[c3,c4,c5]],[[c2,c3],[c4,c5]],[[c2,c3,c4],[c5]]] ;
no
```

6.5 CHAPTER SUMMARY

- Prolog has built-in operators and allows user-defined operators. To declare a user-defined operator, we must use the **op** predicate.

    - An example of the **op** predicate for a built-in Prolog operator is **op(300,xfx,mod)**. This statement says, "Declare an operator called **mod** of type **xfx** (i.e., infix operator with both operands lower in precedence than the operator), with an operator precedence of **300** (on a scale of 0-1200).

- Prolog has numerous types of equality, including **=**, **=:=**, and **==**.

    - The **==** is the stricter form of equality, demanding *literal* equality and distinguishing between variables.

- Some built-in tools helpful for program control are the **cut**, **fail**, **true**, **not**, **repeat**, and **call** predicates, summarized in Table 6.2.

- The comma **,** is an operator representing a conjunction of goals (*and*), while the semicolon **;** represents a disjunction of goals (*or*).

- The **findall**, **bagof**, and **setof** predicates retain multiple solutions.

- The **findall** predicate forces backtracking through *all* solutions, while **bagof** and **setof** require manual backtracking.

- The **bagof** and **setof** predicates are similar to each other, except that the output list in **setof** is ordered and free of duplicate elements.

A LOOK AHEAD

In the next chapter, we shall take a look at more built-in predicates in Prolog. We shall focus on the so-called "meta-logical" predicates. Meta-logical predicates are used to assess the status of variables and data objects in the program, as well as convert these variables and objects into other forms. Implementing the meta-logical predicates opens up a new dimension of Prolog programming.

In addition to meta-logical predicates, we shall introduce ways of changing the Prolog database while the program is executing. This gives Prolog the ability to "learn". We shall describe how to assert and retract clauses while Prolog is executing, thereby making the database "dynamic" (i.e., changing with time) and flexible.

REFERENCES

Bratko, Ivan, *Prolog Programming for Artificial Intelligence*, second edition pp. 161-186, Addison-Wesley, Reading, MA (1990).

Clocksin, W.F. and C.S. Mellish, *Programming in Prolog,* third edition, pp. 110-132, 156-158, Springer-Verlag, New York, NY (1987).

Rowe, Neil C., *Artificial Intelligence Through Prolog*, pp. 236-238, Prentice-Hall, Englewood Cliffs, NJ (1988).

Schnupp, P. and L. Bernhard, *Productive Prolog Programming*, pp.103-116, Prentice-Hall, Englewood Cliffs, NJ (1986).

Sterling, L. and Shapiro E., *The Art of Prolog*, pp. 266-282, MIT Press, Cambridge, MA (1986).

7

MORE BUILT-IN TOOLS

This chapter discusses some additional built-in tools. We first look at "meta-logical" predicates used to assess the status of Prolog data objects. We then learn how to insert and withdraw clauses from the database during program execution, and how to construct and decompose atoms and clauses. Finally, we review some of the built-in debugging tools in Prolog.

7.1 ASSESSING PROLOG TERMS

A. The var (Variable) and nonvar (Non-variable) Predicates

Prolog uses object matching, unification (matching), and backtracking to solve goals. If a goal fails and Prolog has to backtrack, a variable can switch from instantiated to uninstantiated. Furthermore, the type of data object that the variable is instantiated to may change. In one clause, variable **X** may be bound to an integer. After backtracking, **X** may be bound to an atom.

Often we need to know what type of data object that a variable represents. For instance, for the goal

```
X =:= Y
```

both X and Y must be instantiated. If they are not, an error will result. To avoid such an error, we can test whether or not the variables are instantiated by using the predicates **var** (variable) and **nonvar** (non-variable). The goal

```
var(X)
```

succeeds if X is currently an *uninstantiated* variable, i.e., a free variable. The goal

```
nonvar(X)
```

succeeds if X is a fixed data object (a constant or a list, for instance), or if X is an instantiated variable.

The following example shows the use of **var**:

```
?- var(X), X = benzene.
X = benzene
```

Here, the first statement, **var(X)**, succeeds, since **X** is a free variable. Prolog moves to the second goal, **X = benzene**. This goal succeeds, and **X** is instantiated to **benzene**. Prolog reports the result.

We switch the order of goals in the question, and see:

```
?- X = benzene, var(X).
no
```

The first goal, **X = benzene**, succeeds, and **X** is instantiated to **benzene**. Prolog moves to the second goal, **var(X)**. This goal fails, since **X** is not a free variable. Prolog answers ***no***, since both goals could not be satisfied.

We use the **nonvar** predicate, and see:

```
?- nonvar(X), X = ethane.
```

```
no
```

The first statement, **nonvar(X)** fails, since **X** is not an instantiated variable or a fixed data object. Prolog answers ***no***.

If we switch the goal-ordering, we see:

```
?- X = ethane, nonvar(X).
X = ethane
```

The first goal, **X = ethane**, succeeds, and **X** is instantiated to **ethane**. The second goal, **nonvar(X)**, also succeeds, since **X** is not a free variable. Prolog reports the result, ***X*** = ***ethane***.

B. The atom, integer, float, number, and atomic Predicates

We frequently want to know if a term is an atom or an integer. For instance, let us consider the statement

```
X is Y + Z.
```

We realize in this statement that the **is** predicate requires both **Y** and **Z** to be instantiated to numerical expressions. If not, an error results. Therefore, before reaching this statement, we want to make sure that both Y and Z are indeed integers to prevent an error. Prolog allows such testing through several so-called "assessment" predicates: **atom**, **integer**, **float**, **number**, and **atomic**.

The goal

atom(X)

succeeds if **X** is or is instantiated to an atom.

The goal

integer(X)

succeeds if **X** is or is instantiated to an integer.

The goal:

float(X)

succeeds if **X** is or is instantiated to a floating-point number, i.e., a real number.

The goal:

number(X)

is more broad. It succeeds if **X** is or is instantiated to an integer *or* a real number.

Finally, the goal:

```
atomic(X)
```

is the broadest of the assessment predicates. It succeeds if **X** is or is instantiated to a constant of atomic data type, i.e., an atom, an integer, a real number, or a string of characters.

Consider a program that assesses Prolog objects. It tests the nature of a variable, then performs some form of manipulation depending on the type of variable.

```
go(X,Y,Z):-              % If either variable is uninstantiated
      (var(X);           % perform a cut-fail since
      var(Y)),           % the program cannot be used.
      write('The first two arguments must be instantiated'),nl,
      !,fail.
go(X,Y,Z):-              % If both X and Y are atoms, then Z is
      atom(X),           % a list with X as the head and Y as
      atom(Y),           % the tail.
      Z = [X|Y].
go(X,Y,Z):-              % If both X and Y are integers, then
      integer(X),        % Z is the product of X and Y.
      integer(Y),
      Z is X*Y.
```

This program gives the following dialogue:

```
?- go(2,3,Z).
Z = 6
?- go(a,X,Z).
The first two arguments must be instantiated
no
?- go(a,b,Z).
Z = [a,b]
?- go(Y,3,Z).
The first two arguments must be instantiated
no
```

Table 7.1 summarizes the built-in predicates available for assessing Prolog objects.

Table 7.1. Built-in predicates to assess Prolog data objects.

PREDICATE	DESCRIPTION
var(X)	succeeds if X is a free variable
nonvar(X)	succeeds if X is a constant or bound variable
atom(X)	succeeds if X is an atom
integer(X)	succeeds if X is an integer
atomic(X)	succeeds if X is an atom, integer, real number, or string
float(X)	succeeds if X is a real number
number(X)	succeeds if X is an integer or real number

C. Exercises

7.1.1 How will Prolog respond to the following questions?

a. ?- float(2).

b. ?- atom(2); atom(a).

c. ?- atomic(2).

d. ?- X is 4/2, integer(X).

e. ?- X = 4/2, number(X).

7.1.2 Write a program that achieves the following objectives:

(1) Reads an object and tests whether the variable that represents the object is a free or bound variable.

(2) If the variable is a free variable, the program tells the user.

(3) If the variable is bound, tests if the object represented by the variable is:

a) an atom,

b) an integer,

c) a real number,

d) a list,

e) an empty list, or

f) a more complex structure.

(4) Once it identifies the type of data object, reports to the user what the object is.

7.2 CONSTRUCTING AND DECOMPOSING ATOMS: THE NAME PREDICATE

Atoms can be constructed and decomposed using the **name** predicate. The goal

```
name(A,L)
```

is true if **L** is a list of ASCII code numbers for the characters in atom **A**. We briefly discussed the ASCII code in chapter 5 when we introduced the **get** and **get0** predicates. ASCII stands for "American Standard Code for Information Interchange." The ASCII code, summarized in Appendix A, assigns each character an integer value.

If we ask the question:

```
?- name(hello,[104,101,108,108,111]).
```

Prolog responds ***yes***. Why? The letter **h** corresponds to ASCII code 104, **e** corresponds to 101, **l** to 108, and **o** to 111. Since each element in the list corresponds to the appropriate character in the atom **hello**, Prolog correctly responds ***yes***.

We can also ask a question using a free variable, **X**:

```
?- name(one,X).
```

Prolog instantiates **X** to the list of ASCII codes for the atom **one**, and responds:

```
X = [111,110,101]
```

As a more complex example using **name**, we can develop a program that classifies chemicals based on their names. We write the following rules:

```
classify(X):-
        name(ane,TestList),
        name(X,ChemicalList),
        append(_,TestList,ChemicalList),
        write(X), write(' is an alkane').

classify(X):-
        name(ol,TestList),
        name(X,ChemicalList),
        append(_,TestList,ChemicalList),
        write(X), write(' is an alcohol').

classify(X):-
        name(one,TestList),
        name(X,ChemicalList),
        append(_,TestList,ChemicalList),
        write(X), write(' is a ketone').
```

The rules test whether X is an alkane, alcohol, or ketone by looking at the last letters of the name. The first rule says that **X** is an alkane if its name ends with the "ane." The second rule says **X** is an alcohol if its name ends with "ol." The last rule says that **X** is a ketone if its name ends with "one." The **append** relation tests to see if the list of integers created by **name** will "fit into" the chemical name. We show below a dialogue with this program:

```
?- classify(ethanol).
ethanol is an alcohol
yes
?- classify(butane).
butane is an alkane
yes
?- classify(acetone).
acetone is a ketone
yes
?- classify(mundane).
mundane is an alkane
yes
```

The program correctly classifies ethanol, butane, and acetone, but we see a weakness in the program. On the not-so-mundane example of the word "mundane," the program incorrectly identifies the word as an alkane when the word is not even a chemical!

Prolog is a logic-based language that will perform consistently according to the relations specified in the database. Using the word "mundane" with the **name** predicate reminds us that for Prolog to perform correctly, we must program in the correct relations.

7.3 ACCESSING AND ALTERING CLAUSES

Accessing and altering clauses allows us to manipulate the database as Prolog is executing. This ability makes Prolog more flexible and consequently, more powerful. With the **assert** and **retract** predicates, Prolog has the ability to adjust its own database.

A. The assert and retract Predicates

The goal

```
assert(X)
```

adds the clause X to the Prolog database. The clause then immediately becomes available for use. The term **X** must:

(1) syntactically represent a clause, and

(2) be sufficiently instantiated to act as a clause (i.e., the principal functor must be known).

If Prolog backtracks into **assert**, the addition of clause **X** to the database will *not* be undone. The **assert** predicate can only add clauses to the database, never remove them, even upon backtracking.

The goal

retract(X)

deletes the clause **X** from the database. The clause is immediately "forgotten" by Prolog and is unavailable for use. For **retract(X)** to succeed, **X** must be instantiated to a clause that matches a clause already in the database.

The following dialogue illustrates the use of **assert** and **retract**:

```
?- explosive(potassium_nitrate).
no
?- assert(explosive(potassium_nitrate)).
yes

?- explosive(potassium_nitrate).
yes
?- retract(explosive(potassium_nitrate)).
yes
?- explosive(potassium_nitrate).
no
```

When we first pose the **explosive(potassium_nitrate)** question, Prolog answers ***no***, because no matching facts exist in the database. If we then **assert** the fact into the database, Prolog responds ***yes***, indicating that the fact has been successfully added. When we ask the **explosive** question again, Prolog responds ***yes***, since the matching fact is now in the database. We then **retract** the fact from the database, and ask the **explosive** question one more time. Prolog responds ***no*** since the matching relation has been removed from the database.

When asserting clauses, we can choose whether to add the clause to the *front* or the *back* of the database. The predicates **asserta** and **assertz** control where we add the clause. The goal:

```
asserta(X)
```

adds clause X to the *beginning* of the database. The goal:

```
assertz(X)
```

adds the clause to the *end* of the database. We can remember the difference between the two by realizing that "a" is at the beginning of the alphabet, and "z" is at the end.

As an example of the use of **asserta** and **assertz**, we consider the program below consisting of facts about explosive materials:

```
explosive(potassium_nitrate).
explosive(trinitrotoluene).
```

Prolog responds with the following dialogue:

```
?- explosive(X).
X = potassium_nitrate ;
X = trinitrotoluene ;
no
?- asserta(explosive(dynamite)).
yes
?- explosive(X).
X = dynamite ;
X = potassium_nitrate ;
X = trinitrotoluene ;
no

?- retract(explosive(dynamite)),
assertz(explosive(dynamite)).
yes
?- explosive(X).
X = potassium_nitrate ;
X = trinitrotoluene ;
X = dynamite ;
no
```

Where we add the clause changes the route Prolog takes to solve the problem. When we add **explosive(dynamite)** to the beginning of the database, it becomes the most prominent clause. When queried, Prolog responds with

X = ***explosive(dynamite)*** as the first solution.

When we assert clauses at the end, however, they have less prominence in the program. Prolog finds all other solutions before attempting to match the last clause. Thus, this principle of asserting at the beginning or end of the database can be very important when ordering rules in a Prolog program.

Importantly, *any* clause can be asserted or retracted. To illustrate this point and the importance of asserting at the beginning or end of the database, let us consider the deterministic **member** relation in the Prolog database:

```
member(X,[X|_]):- !.      % base case- X is a member if X is the head.
member(X,[_|T]):-         % recursive case- X is a member if X is a
        member(X,T).      %                 member of the tail.
```

Note here that the base case is *in front* of the recursive case to minimize needless recursion. Thus, we get following dialogue:

```
?- member(X,[dynamite,trinitrotoluene,potassium_nitrate])
X = dynamite ;
no
```

Prolog takes only one step to answer the question by simply matching the base case. The cut eliminates alternate solutions.

But now we retract the recursive rule:

```
?- retract( member(X,[_|T]):-
                              member(X,T) ).
yes
```

The **retract** statement simply instructs Prolog to remove the matching clause from the database. Prolog answers ***yes***, indicating that there is a matching **member** rule and it has been retracted.

If we now ask the same question again, we get the same answer:

```
?- member(X,[dynamite,trinitrotoluene,potassium_nitrate])
X = dynamite ;
no
```

The reason? The solution involves only the first **member** clause, and that clause contains a cut. The second **member** clause, which has been retracted, is not used anyway. However, when we ask the following question,

?- member(potassium_nitrate,[dynamite, trinitrotoluene, potassium_nitrate]).

Prolog incorrectly responds ***no***. We know that **potassium_nitrate** is certainly a member of the list. But, because the recursive **member** relation is no longer in the database, Prolog cannot investigate every element in the list. Therefore, Prolog incorrectly responds ***no***.

Now we wish to add the recursive relation back to the database using **asserta**:

```
?- asserta( member(X,[_|T]):-
                              member(X,T) ).
yes
```

The **asserta** predicate places the recursive **member** relation at the *top* of the database. Table 7.2 summarizes the Prolog database after each of the above steps.

Has our Prolog program changed? Certainly! Because we used **asserta** (instead of **assertz**) above, our recursive case is now the first clause -- in the reverse order from our original database (see Table 7.2).

If we now ask the question:

```
?- member(X,[potassium_nitrate,trinitrotoluene,dynamite]).
```

Prolog follows a different route to find the answer. Prolog enters the recursive rule first and ends up going through the entire list. After doing so, Prolog then attempts to match the goal with the base case.

Table 7.2. Prolog database after retract and asserta.

<table>
<tr><th>ORIGINAL DATABASE</th></tr>
<tr><td>member(X,[X|_]):- !.
member(X,[_|T]):- member(X,T).</td></tr>
<tr><th>DATABASE AFTER
retract(member(X,[_|T]):-member(X,T)).</th></tr>
<tr><td>member(X,[X|_]):- !.</td></tr>
<tr><th>DATABASE AFTER
asserta(member(X,[_|T]):-member(X,T)).</th></tr>
<tr><td>member(X,[_|T]):- member(X,T).
member(X,[X|_]):- !.</td></tr>
</table>

Prolog's response to the question is now:

```
X = dynamite ;
X = trinitrotoluene ;
X = potassium_nitrate ;
no
```

Because the base case and the recursive case are reversed, the response is different from the response to first question. You may wish to trace through the program yourself to see why Prolog answers this way. This new ordering of clauses changes things in two ways:

(1) Prolog gives multiple solutions rather than just a single solution.

(2) Prolog solves the problem in the reverse order (note that **dynamite** is the first solution rather than **potassium_nitrate**). Careful use of **asserta** and **assertz** is essential to good Prolog programming.

Many versions of Prolog support a variation of **retract**, known as the **retractall** predicate. The relation **retractall(F,A)** retracts all clauses from the database that have the functor **F** with an arity of **A**. For example, let us consider again a database of explosive materials:

```
explosive(potassium_nitrate).
explosive(trinitrotoluene).
explosive(dynamite).
```

We ask the following question:

```
?- explosive(X).
```

Prolog responds with the following dialogue:

```
X = potassium_nitrate ;

X = trinitrotoluene ;

X = dynamite ;

no
```

Now we use the **retractall** predicate:

```
?- retractall(explosive,1).

yes
```

Prolog has successfully retracted *all* **explosive** clauses from the database that have an arity of one. We now ask:

```
?- explosive(X).
```

Prolog responds ***no***, since all of the **explosive** clauses have been removes from the database.

B. The listing Predicate

The goal

```
listing(X)
```

where **X** is instantiated to an atom, will list out all clauses with **X** as the predicate. The output goes to the current output stream. For example, we may think that an error exists in the **member** relation in our database. The question:

```
?- listing(member).
```

instructs Prolog to list all clauses with **member** as the predicate. Prolog responds:

```
member(x,[X|_]):- !.
member(X,[_|T]):-
      member(X,T).
yes
```

We indeed see a mistake in the first **member** clause. The first argument has the atom **x** instead of the variable **X**. We could now correct the relation using **retract** and **assert**. The **listing** predicate is helpful for correcting mistakes, especially when a clause has undergone numerous assertions or retractions.

C. The clause Predicate

The goal

```
clause(X,Y)
```

matches X and Y with a clause in the database. The variable X is matched with the head, and Y is matched with the body. Consider the following program in the database:

```
explosive(potassium_nitrate).
explosive(trinitrotoluene).
explosive(dynamite).
flammable(coal).
flammable(wood).

combustible(X):-
     explosive(X).
combustible(X):-
     flammable(X).
```

The following dialogue shows how **clause(X,Y)** behaves:

```
?- clause(combustible,Y).
Y = explosive(X) ;
Y = flammable(X) ;
no
```

```
?- clause(explosive,Y).
Y = true ;
Y = true ;
Y = true ;
no


?- clause(non_flammable,Y).
no
```

Note that **clause(X,Y)** will match facts. We can view facts as rules where the body consists of the built-in clause **true**:

```
explosive(dynamite):- true.
```

If the **clause** relation matches a fact, it instantiates Y to the atom **true**.

To use the **clause(X,Y)** relation, the variable **X** must be sufficiently instantiated such that the principal functor of the structure it represents is known. If the principal functor is not specified, an error results. For example, if we ask the question:

```
?- X = combustible(Z), clause(X,Y).
```

Prolog responds:

```
Y = explosive(A) ;
Y = flammable(A) ;
no
```

Here, **X** is sufficiently instantiated for **clause** to perform accurately. However, if we ask:

```
?- clause(X,Y).
```

Prolog will responds with an error message, since variable **X** is uninstantiated and the principal functor is unknown.

The **clause** relation has other important attributes. If no match is found, the clause fails. If numerous matches exist, Prolog chooses the first one. Backtracking will yield additional solutions.

D. Exercises

7.3.1 The **findall** predicate is built into Prolog and was discussed in section 6.4. Using **assert** and **retract**, develop Prolog rules that emulate **findall**.

7.3.2 Write a program that inputs a chemical formula from the user's screen (e.g., c6h6 for benzene, c2h5oh for ethanol, ch3cooh for acetic acid, and h3po4 for phosphoric acid) and determines whether the material is organic or inorganic. If the material is organic, the program also determines if it is:

- a hydrocarbon
- a carbohydrate
- a thiol
- an alcohol
- a carboxylic acid

The program states *every* category to which the formula belongs. Note that we must enter lower-case letters, i.e., c2h4 rather than C2H4, since Prolog treats C2H4 as a variable rather than an atom.

7.4 CONSTRUCTING AND DECOMPOSING STRUCTURED OBJECTS

Constructing and decomposing structured objects is useful in Prolog-based artificial-intelligence applications. The three built-in predicates used for this purpose are: **=..**, **arg**, and **functor**.

A. The =.. Predicate

The built-in predicate =.. (traditionally pronounced "univ") functions as an infix operator for constructing new structures. The goal

```
X =.. L
```

instantiates the variable **X** to a Prolog structure built from list **L**, where the first element of **L** is the principal functor and the remaining elements are arguments. The following dialogue illustrates its use:

```
?- X =.. [flammable,coal].
X = flammable(coal)
?- X =.. [explosive,trinitrotoluene,organic].
X = explosive(trinitrotoluene,organic)
?- X  =.. [flammable, coal, impurities(sulfur,heavy_metals)].
X = flammable(coal,impurities(sulfur,heavy_metals))
```

The =.. predicate is useful for creating functors that are *unknown* when the program is originally written. The user can create functors as needed. This feature of creating functors on-demand expands Prolog's abilities.

For example, we can write a Prolog program to *develop* classifications of *any* topic. When we originally write the program, we do not know the topics or classifications, and consequently, cannot name the functors. The program ia as follows:

```
develop_classifications:-
        write('What category would you like to classify?'),nl,
        read(X),
        repeat,
        write('Define Y such that Y is a kind of '),
        write(X),nl,
        read(Y),
        write('Define Z such that Z is a kind of '),
        write(Y),write(X),nl,
        read(Z),
        Clause =.. [Y,X,Z],
        assertz(Clause),
        write('Would you like to develop other classifications?'),nl,
        write('Enter y or n'),nl,
        read(n),!.
```

If we apply this program to the classification of fuels, it produces the following dialogue:

```
?- develop_classifications.
What category would you like to classify?
fuel
Define Y such that Y is a kind of fuel
liquid
Define Z such that Z is a kind of liquid fuel
gasoline
Would you like to develop other classifications?
Enter y or n
y
Define Y such that Y is a kind of fuel
liquid
Define Z such that Z is a kind of liquid fuel
kerosene
Would you like to develop other classifications?
Enter y or n
y
Define Y such that Y is a kind of fuel
solid
Define Z such that Z is a kind of solid fuel
coal
```

```
Would you like to develop other classifications?
Enter y or n
y
Define Y such that Y is a kind of fuel
solid
Define Z such that Z is a kind of solid fuel
wood
Would you like to develop other classifications?
Enter y or n
n
yes
```

At the end of this dialogue, the following classifications are in the database:

```
fuel.
liquid(fuel,gasoline).
liquid(fuel,kerosene).
solid(fuel,coal).
solid(fuel,wood).
```

The last statement says that a solid is a kind of fuel and wood is a kind of solid. We created the functors liquid and solid used to classify fuels. We further classified:

- gasoline and kerosene under the category of liquid fuels.
- coal and wood under the category of solid fuels.

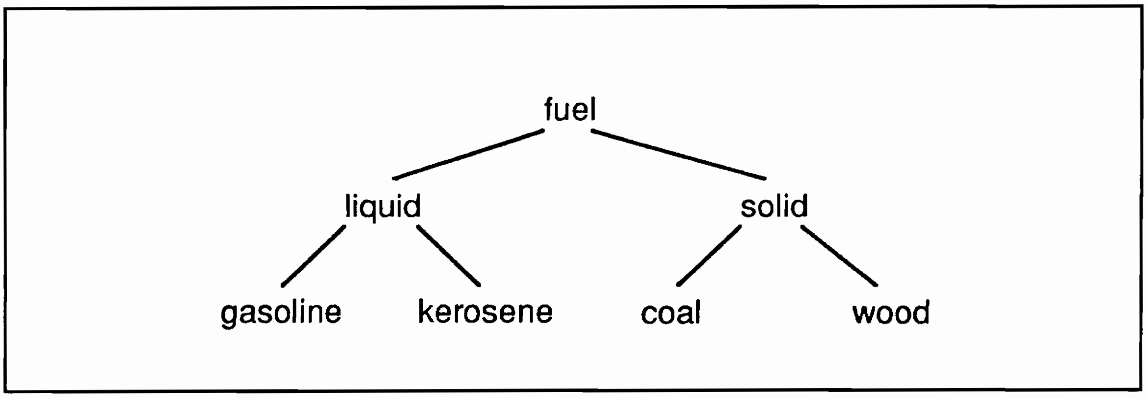


Figure 7.1. Classifications of fuels.

Figure 7.1 summarizes the classification structure just developed. These classifications are a form of artificial intelligence known as *frame-based expert systems*. We shall discuss frame-based expert systems in more detail in Chapters 10 and 11. For now, let us simply realize that the functors in the classification system were *unknown* when we originally wrote the program. The user then created the functors. The success of the procedure depended on the =.. predicate.

The above example demonstrates how we can use the =.. predicate to *build-up* structures and develop a classification system of fuels. We may also use =.. to *tear down* structures. We ask the question:

```
?- liquid(fuel,gasoline) =.. [Functor|Arguments].
```

and Prolog responds:

```
    Functor = liquid   Arguments = [fuel,gasoline]
```

We have taken a single term, the structured object **liquid(fuel,gasoline)**, and have torn it down to two terms, the atom **liquid** and the list [fuel, gasoline]. We can then use these objects to construct other related structures.

B. The functor Predicate

The goal

```
    functor(S,F,N)
```

is true if structure **S** has the principal functor **F** with an arity of **N**. The following dialogues illustrate its use:

```
    ?- functor(liquid(fuel,gasoline),F,A).
    F = liquid
    A = 2 ;
    no
```

We ask this question with structure **S** instantiated to the structure

liquid(fuel,gasoline). The **functor** predicate instantiates **F** to the principal functor, the atom **liquid**, and **A** to the arity of the structure, which is **2**.

We now see:

```
?- functor([solid,liquid,gas],F,A).
no
```

Prolog responds ***no***, since the list **[solid,liquid,gas]** is not a structured object. The **functor** predicate fails if the first argument is not a structured object.

Finally we see:

```
?- functor(liquid,F,A).
F = liquid
A = 0 ;
no
```

Here, the structure **S** is an atom. Prolog views this as a structure with an arity of zero. The **functor** predicate succeeds, and **F** is instantiated to **liquid** and **A** to **0**.

The **functor** predicate can be used to tear down or to build up structures. The following dialogue constructs the structure **S**:

```
?- functor(S,liquid,2).
S = liquid(_X1,_X2)
```

The variable **S** is instantiated to the structure **liquid(_X1,_X2)**. Note that both arguments of the functor **liquid** remain free variables,denoted internally in Prolog as _X1 and _X2.

To tear down a structure, we enter:

```
?- functor(liquid(fuel,kerosene), F, A).
```

and Prolog responds:

```
F = liquid    A = 2;
no
```

Section 7.4C illustrates another application of the **functor** predicate.

C. The arg Predicate

The goal

```
arg(N,S,A)
```

is true if **A** is the **N**-th argument in structure **S**. Arguments are numbered from left to right starting with 1. The following examples illustrate the **arg** predicate:

```
?- arg(2,liquid(fuel,gasoline),Argument).
Argument = gasoline ;
no
?- arg(2,solid(fuel,wood),coal).
no
```

When using **arg(N,S,A)**, **N** must be instantiated, and **S** must be instantiated in such a way that the principal functor is known. We generally use the **arg** predicate to access a certain part of a structure. We cannot build terms with **arg**. When the variable A is uninstantiated, the goal **arg(N,T,A)** will extract out the **N**-th argument from structure **S** and instantiate it to A.

To illustrate the use of **functor** and **arg** together, we consider the following database:

```
liquid(fuel,gasoline).
liquid(fuel,kerosene).
solid(fuel,coal).
solid(fuel,wood).
```

We can write the procedure **assemble** as follows:

```
assemble(X):-
arg(1,X,NewFunct),      % create the functor
functor(S,NewFunct,4),  % create a structure with arity = 4
arg(1,S,gasoline),      % place first argument in structure
arg(2,S,kerosene),      % place second argument in structure
arg(3,S,coal),          % place third argument in structure
arg(4,S,wood),          % place fourth argument in structure
write(S).               % write the structure
```

This program gives the following dialogue:

```
?- assemble(liquid(fuel,gasoline))
fuel(gasoline,kerosene,coal,wood);
no
```

Here, we use **arg** to access the first argument in **(liquid(fuel,gasoline)**, and instantiate **X** to the atom **fuel**. This atom then serves in the **functor** goal to create a new structure with four arguments: **fuel(_X1,_X2,_X3,_X4)**. Note that _X1 through **X4** are all free variables at this point. The following four **arg** goals are used to do a step-by-step instantiation of arguments 1 through 4 in the newly created **fuel** predicate, resulting in

```
      fuel(gasoline,kerosene,coal,wood)
```

as the final answer.

D. Exercises

7.4.1 Develop the relation **retractfacts(F,A)**. The **retractfacts** predicate retracts *all* Prolog facts from the database that have the functor **F** and an arity of **A**.

7.4.2 Develop the relation **instantiated(S)** where **S** is any Prolog structure. The **instantiated** relation succeeds if all the arguments of structure **S** are instantiated (i.e., **S** has no free variables).

7.4.3 We wish to develop the relation **substitute(I1,S1,I2,S2)**. The **substitute** relation succeeds when each occurrence of item **I1** in statement **S1**, is replaced by item **I2** to give statement **S2**. For example, consider the following parametric substitution:

```
    ?- substitute(tan(x),pi*tan(x)*f(tan(x)),t,F)).
    F = pi*t*f(t) ;
    no
```

In this example, all occurrences of the item **tan(x)** in the statement

pi*tan(x)*f(tan(x)) are replaced by **t** to give **pi*t*f(t)**.

The **substitute** relation is used for Prolog applications in mathematics. Bratko (1986) identifies a subtle point about **substitute**. When items within **S1** are free variables, it is important to realize that we are concerned about all *matching* occurrences of item **I1**. Suppose we ask the following question:

```
?- substitute( a + b, f( a, A + B, g(a + b)), v, F).
```

Here, the principal functor of **S1** is known (the atom f), but **S1** is *not* fully instantiated. Variables **A** and **B** are free variables. The question produces:

```
F = f( a, v, g(v))    A = a    B = b ;
no
```

and *not*

```
F = f( a, v+v, g(v))    A = a + b    B = a + b ;
no
```

The atom **v** is not substituted for every free variable in **f(a, A + B, g(a + b))**. Instead, it is substituted only for *matching occurrences* of the structure **a + b**. There are two matching occurrences. One match is with the structure **A + B**, where **A** is instantiated to **a** and **B** to **b**. The other match is with **g(a + b)**, where **a + b** matches **a + b**.

a. Develop the rules (written in English) required to fully characterize

the **substitute** relation.

b. Write the complete Prolog relation for **substitute**.

7.5 DEBUGGING TOOLS IN PROLOG

Debugging in Prolog is usually easier than debugging in a conventional language such as FORTRAN. Prolog debugging is easier for two reasons:

- Prolog is an interactive language.
- All its clauses are directly accessible.

In a conventional language, a bug can be embedded very deeply in a complex procedure and be difficult to resolve.

In Prolog, however, we can instantly call any predicate in the entire program to check if it performs correctly. After testing specific rules, we can move up to "higher level" rules, and eventually to modules and finally to the entire program. Prolog has a number of built-in predicates to aid in this debugging process.

A. The trace and notrace Predicates

The predicate

trace

instructs Prolog to begin an exhaustive, step-by-step display of each step it executes. During the trace, Prolog displays all information regarding goal satisfaction. A good Prolog tracer will display:

- *Goal information:* the current goal, and possibly the listing of all goals required to solve the program.
- *Success information:* after success, what variables are instantiated and the objects to which they are instantiated.
- *Backtrack information:* the goal that needs to be re-satisfied and the next clause Prolog will test to see if it ultimately succeeds.

The predicate

notrace

turns off the exhaustive trace procedure.

When tracing a program, four types of events can happen: CALL, EXIT, REDO, and FAIL. They are summarized below.

- *CALL*- a CALL occurs when Prolog begins to satisfy a new goal.
- *EXIT*- an EXIT occurs when the current goal is satisfied.
- *REDO*- a REDO occurs when Prolog is forced to backtrack into and re-satisfy a goal.
- *FAIL*- a FAIL occurs when a goal fails.

We can view these four events graphically, as shown in Figure 7.2.

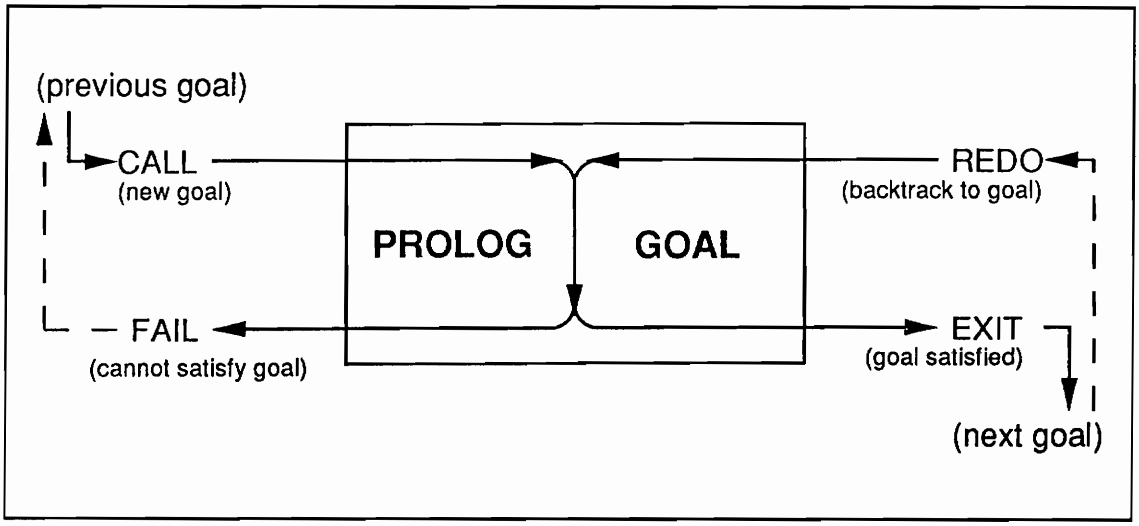


Figure 7.2. The four possible events in a trace.

Let us examine a program and see exactly how Prolog traces through it. Consider the **union** relation, introduced in chapter 4 (section 4.2, exercises). The predicate **union(L1,L2,L3)** succeeds if the union of list L1 with list **L2** gives list **L3**. These lists are treated as sets, so the exact ordering within the list does not matter. We write the **union** relation in database db7A:

```
                         % db7A
union([],A,A).           % base case: the union of A with [] is A
union([H|T],X,I):-       % recursive case: if H is a member of X,
       member(H,X),!,      then the union of [H|T] with X is the same
       union(T,X,I).       as union of T with X.
union([H|T],X,[H|I]):-   % recursive case: H is not a member of X, so
       union(T,X,I).       the union of [H|T] with X is the same as
```

```
                                        the union of T with X with H as the head
                                        the result.
      member(X,[X|_]).
      member(X,[_|T]):- member(X,T).
```

Let us first turn on the trace mechanism:

```
      ?- trace.
      yes
```

Prolog's trace mechanism is now active. If we ask the question:

```
      ?- union([a,b],[b,c],X).
```

Prolog responds with the following trace:

```
      (0) CALL: union([a,b],[b,c],_x1).
```

The _x1 is an internal Prolog variable assigned to the third argument of the **union** predicate.

```
      (1) CALL: member(a,[b,c]).
      (2) CALL: member(a,[c]).
      (3) CALL: member(a,[]).
```

Each additional **member** CALL results from the recursive **member** rule.

```
(3) FAIL: member(a,[]).
(2) FAIL: member(a,[c]).
(1) FAIL: member(a,[b,c]).
```

The **member** relation fails, since **a** is clearly not a member of **[b,c]**.

```
(4) CALL: union([b],[b,c],_x2).
```

This CALL is a backtrack, and results from the third **union** clause. The variable **_x2** is yet another internal Prolog variable.

```
(5) CALL: member(b,[b,c]).
(5) EXIT: member(b,[b,c]).
(6) CALL: !
(6) EXIT: !
```

The cut predicate is always immediately successful.

```
(7) CALL: union([],[b,c],_x2).
```

This CALL is the recursive step in the second **union** clause. Note that Prolog maintains the identity of variable **_x2**. Variable **_x2** was introduced in trace step 4 when the **union** predicate was called. Since the step 4 union call is successful, variable instantiations remain in place.

```
(7) EXIT: union([],[b,c],[b,c]).
```

CALL 7 succeeds by matching the **union** base case. Prolog's internal variable **_x2** is now instantiated to the list **[b,c]**.

```
(4) EXIT: union( [b], [b,c], [b,c]).
(0) EXIT: union( [a,b], [b,c], [a,b,c]).
```

With the success of the original CALL, internal variable **_x1** is instantiated to the list **[a,b,c]**, producing the final answer. Prolog reports the result:

```
X = [a,b,c]
```

B. The spy and nospy Predicates

The **spy** and **nospy** predicates are more selective tracers. The statement:

```
spy(P)
```

instructs Prolog to begin tracing predicate P only. The goal

```
nospy(P)
```

stops Prolog from "spying" on predicate P.

The **spy** tool is useful for tracing a specific predicate, and therefore has some advantages over the **trace** tool. When we enter **trace**, we instruct Prolog to do an exhaustive trace. Prolog may trace through procedures which we know operate correctly, and thus waste time. The **spy** predicate allows us to save time by tracing only through specified predicates.

To illustrate, let us consider the **union** relation again in database *db7A*. This time, instead of tracing exhaustively, we use the **spy** tool. We wish to trace the **union** predicate only, since we know that the **member** predicate is correct. We enter:

```
?- spy(union).
```

and Prolog responds:

```
yes
```

indicating that the **spy** mechanism has been turned on for the predicate **union**.

We then enter, as before, the following question:

```
?- union([a,b],[b,c],X).
```

Prolog responds with the following dialogue:

```
(0) CALL: union([a,b],[b,c],_x1).
```

The _x1 is again an internal Prolog variable assigned to the third argument of the **union** predicate.

```
(1) CALL: union([b],[b,c],_x2).
```

Only the **union** predicates are displayed. All the steps Prolog took in processing the **member** relation are not seen, since we have a **spy** on the predicate **union** only. The variable **_x2** is yet another internal Prolog variable.

```
(2) CALL: union([],[b,c],_x2).
```

As before, this CALL is the recursive step in the second **union** clause. Note that Prolog maintains the identity of variable **_x2**.

```
(2) EXIT: union([],[b,c],[b,c]).
```

CALL 2 succeeds by matching the **union** base case. Prolog's internal variable _**x2** is now instantiated to the list **[b,c]**.

(1) EXIT: ***union([b], [b,c], [b,c])***.

(0) EXIT: ***union([a,b], [b,c], [a,b,c])***.

With the success of the original CALL, the internal variable _**x1** is instantiated to the list **[a,b,c]**, again the final answer, and Prolog reports the result:

X = [a,b,c]

The debugging tools available to us depend on the version of Prolog. Some versions, for instance, do not have the **spy** predicate, but instead use **trace** with an arity of one:

trace(P)

In these systems, the predicate **trace** begins an *exhaustive* trace, and **trace(P)** begins a trace of the predicate P *only*.

C. Exercises

7.5.1 Consider the following program for the **remove_duplicate** procedure, introduced in section 3.4G:

```
remove_duplicate([],L2,L2).                    % rule 1


remove_duplicate([H|T],Accumulator,L2):-        % rule 2
      member(H,Accumulator),
      remove_duplicate(T,Accumulator,L2).


remove_duplicate([H|T],Accumulator,L2):-        % rule 3
      remove_duplicate(T,[H|Accumulator],L2).


member(X,[X|_]).
member(X,[_|T]):-
      member(X,T).
```

Suppose we enter the **trace** command, instructing Prolog to do an exhaustive trace of the program. We then ask the following question:

```
?-remove_duplicate([b,b,a,a,t,t,e],[],X).
```

Using the CALL, EXIT, REDO and FAIL terminology, perform a step-by-step trace of the procedure.

7.5.2 What would the trace be from question 7.5.1 if we entered **spy(remove_duplicate)** instead of **trace** ?

7.6 CHAPTER SUMMARY

- The **var** and **nonvar** predicates are useful for assessing Prolog terms; **var(X)** succeeds if X is uninstantiated; **nonvar(X)** succeeds if X is instantiated.

- The **name** predicate is useful for constructing and decomposing atoms.

- The **assert** and **retract** predicates are the two main ways to dynamically manipulate Prolog's database. The **assert** predicate adds a clause to the database, and has two forms: **asserta** and **assertz**. The **retract** predicate removes a matching clause from the database; **retractall(F,A)** removes all clauses that begin with functor **F** and have an arity of **A**.

- The predicate **listing(X)** will output all clauses with X as the predicate. The predicate **clause(X,Y)** matches X with the head and Y with the body of a clause in the database.

- The built-in predicates **=..**, **functor**, and **arg** are useful for constructing and decomposing structures.

- The **trace** and **notrace** predicates are useful for debugging; **trace** turns on an exhaustive display of each step Prolog executes. The **notrace** predicate turns **trace** off.

- The **spy(P)** and **nospy(P)** are also useful for debugging. They are the same as **trace** and **notrace**, except that they involve only the specific predicate **P**.

A LOOK AHEAD

In the next chapter, we shall discuss programming techniques and helpful hints, and suggest some useful ideas to aid in successfully developing Prolog programs. Once this is complete, we should be able to move full force into practical applications.

REFERENCES

Bratko, I., *Prolog Programming for Artificial Intelligence*, second edition, pp.161-186, Addison-Wesley, Reading, MA (1990).

Burnham, W.D. and A.R. Hall, *Prolog Programming and Applications*, pp. 68-89, Halsted Press, a division of John Wiley & Sons, New York, NY (1985).

Clocksin, W.F. and C.S. Mellish, ***Programming in Prolog***, third edition, pp. 111-126, Springer-Verlag, New York, NY (1987).

Garavaglia, Susan, *Prolog Programming Techniques and Applications*, pp. 326-333, Harper & Row Inc., New York, NY (1987).

Schnupp, P. and L. Bernhard, *Productive Prolog Programming*, pp.67-81, Prentice-Hall, Englewood Cliffs, NJ (1986).

Sterling, L. and Shapiro E., *The Art of Prolog*, pp. 95-99, 179-181, MIT Press, Cambridge, MA (1986).

8

PROGRAMMING TECHNIQUES IN PROLOG

In this chapter, we discuss general principles involved in developing good

Prolog programs. These principles include techniques for program layout, efficiency, and development, some of which have been discussed earlier in the book. This chapter highlights and formalizes those and other principles.

8.1 PRINCIPLES OF GOOD PROGRAMMING

A good Prolog program. What does it contain? Ivan Bratko (1990, pp.187-208), has a good summary. A good Prolog program should possess:

- Correctness Above all, a good program should be correct. That is, it should do what it intends to do. This point may seem a trivial, self-explanatory requirement; however, complex programs often do not perform correctly. Programmers commonly neglect this obvious criterion and pay more attention to other criteria, such as efficiency.

- Efficiency A good program should not waste computer time and memory.

- Readability A good program should be easy to read and understand. It should not be more complicated than necessary. Avoid clever programming tricks that obscure the meaning of the program. The general organization and layout of the program affects its readability.

- Modifiability A good program should be easy to modify and extend. Good readability and modular organization of the program facilitate future modifications.

- Robustness A good program should be robust. It should not immediately crash when the user enters incorrect or unexpected data. Instead, the program should stay "alive" and behave reasonably, reporting any such errors.

- Documentation A good program should be properly documented. The minimal documentation is the program's listing, including sufficient program comments.

For scientific and engineering applications of Prolog, we should pay particular attention to correctness and modifiability. For example, a programmer may have overlooked special circumstances, where a particular Prolog rule should not be used. We can handle such situations more easily when the programmer has emphasized correctness and modifiability.

To create a "good" Prolog program, we need to focus on three primary areas:

- program layout

- program efficiency
- program development

The following sections address these topics.

8.2 PROGRAM LAYOUT

We describe below a number of program-layout techniques that make the program more readable.

A. Comments

We should *use comments liberally*. Though liberal use of comments seems obvious, programmers too frequently ignore it. Effective comments should make programs almost self-explanatory.

Prolog provides two ways to implement comments. The first is to enclose the comment between special brackets, /* and */, as shown below.

```
/*****************************************
 *       This is a Prolog comment        *
 *****************************************/
```

Prolog ignores all characters between the opening bracket /* and the closing bracket */.

Another way to write a comment in Prolog is with the percentage sign, %, as discussed in section 1.2. For example:

```
append([],L,L).                      % the base case
append([Head|Tail],L2,[Head|L3):-    % the recursive case
```

```
          append(Tail,L2,L3).
```

Prolog ignores all characters between the **%** and the end of the line.

Generally, we use the **/*** and ***/** brackets to separate modules or make lengthy comments. The **%** is for smaller, less dominant comments.

B. Meaningful Terms

Make all predicate, variable, and object names as meaningful as possible in the program. For example, when splitting a list,

```
     [H|T]
```

is more meaningful than

```
     [X|Y]
```

since **H** represents "head" and **T** represents "tail."

Predicate names should relate to the objective of the predicate. For example, **delete** deletes an element from a list and **member** tests for membership:

```
     delete(H,[H|T],T).
     delete(X,[H|T1],[H|T2]):-
```

```
          delete(X,T1,T2).


     member(X,[X|_]).

     member(X,[_|T]):-

          member(X,T).
```

C. Consistent Layout

Laying out clauses consistently involves several points:

- Group all clauses with the same head together, and place all sub-goals on successive lines.
- Evenly indent all sub-goals.
- Skip lines between complex clauses.

As an example, let us consider the collection of "utility" list-processing relations. They are "utility" relations because almost all Prolog programs, regardless of their areas of application, use them.

Laying out the utilities as follows:

```
append([],L,L).
append([Head|Tail],L2,[Head|L3):-
     append(Tail,L2,L3).

reverse([],L,L).
reverse([Head|Tail],L2,L3):-
     reverse(Tail,[Head|L2],L3).
     delete(X,[X|Tail],Tail).
     delete(X,[Y|Tail_1],[Y|Tail_2]):-
          delete(X,Tail_1,Tail_2).
```

is more readable than laying them out as:

```
     append([],L,L).
     reverse([],L,L).

     reverse([Head|Tail],L2,L3):-
     reverse(Tail,[Head|L2],L3).
     delete(X,[X|Tail],Tail).
     append([Head|Tail],L2,[Head|L3):- append(Tail,L2,L3).
     delete(X,[Y|Tail_1],[Y|Tail_2]):-
     delete(X,Tail_1,Tail_2).
```

Both layouts represent the exact same program. However, in the second set, the clauses are grouped and indented inconsistently. We did not indent all sub-goals, nor did we skip lines between clauses. Consequently, program readability suffers.

D. Short Clauses

Clauses are more readable if we keep sub-goals to a minimum. Frequently, clauses have the general form:

```
     clause:-

          tests_for_applicability,

          action.
```

When entering a clause, we test certain variables to see if the rule is applicable. If the variables pass and the rule applies, Prolog takes certain actions. If the test fails, the clause fails and Prolog does not use it.

In scientific applications, the test section can get very involved and difficult to understand. To alleviate this problem, we may develop separate test predicates instead of building the test right into the rule itself.

As an example, let us use Prolog to analyze a fluid-flow situation. The rule we have designed applies only if the fluid is:

(1) in the laminar flow regime,
(2) nonreactive chemically, and
(3) a single-phase system.

Testing for each condition explicitly within one rule would make the program difficult to read. The best way to structure the clause would be to use test predicates:

```
clause:-
      test_for_laminar_flow,         % calls the test for laminar flow
      test_for_chemical_reaction,    % calls the test for no reactions
      test_for_single_phase,         % calls the test for a single phase
      take_action.                   % the clause is now applicable
```

The clause is now relatively easy to read and understand.

E. Limited Use of the Semicolon Operator

The semicolon operator can be convenient, but it also clouds programs. We should therefore limit its use. For instance, let us write a relation about the hazardous nature of materials. Material X is hazardous if it is: 1) poisonous, 2) explosive, 3) a toxic liquid that can be absorbed into the skin, 4) highly acidic, or 5) highly alkaline. We can be write this relation as a single Prolog rule:

```
hazardous(X):-
      poisonous(X);explosive(X);
      (toxic(X),liquid(X),absorb(X,skin));
      acidic(X); alkaline(X).
```

While this clause is acceptable, it is also somewhat difficult to read. A more understandable layout would be:

```
hazardous(X):-
      poisonous(X).

hazardous(X):-
      explosive(X).

hazardous(X):-
      toxic(X),
      liquid(X),
      absorb(X,skin).

hazardous(X):-
      acidic(X).

hazardous(X):-
      alkaline(X).
```

The latter case makes the requirements for a hazardous material much clearer.

F. Careful Use of assert, retract, and cut

The use of **assert**, **retract**, and **!** (cut) can cloud the meaning of a program. When using these tools, we should try to isolate them. Embedding assert, **retract** and **!** inside a complex clause makes the clause difficult to understand. Moreover, we may not be able to understand the reason for assert, **retract** or !, so include comments if these tools are used.

For example, consider the following program:

```
adjust_database(X):-    % remove the fact that X is nonhazardous
        nonhazardous(X), % from the database, and assert the
        retract(nonhazardous(X)),     % fact that X is safe into the
        assert(safe(X)).              % database.
```

The program dynamically adjusts the database during execution. We have a single clause **adjust_database** exclusively for this purpose, fully explained with comments. This **adjust_database** clause is an example of the proper way to use **assert** and **retract**.

G. Utility Predicates

Utility predicates (section 8.2C) such as **reverse**, **member**, **append**, **delete**, etc. should be placed in their own section. This technique has a number of advantages. First, it improves program readability. Because utility predicates typically have many variables, they are messy to read. If they are in their own section rather than in a program module, the module will be easier to read.

A second advantage is that this sectioning prevents "reinventing the wheel" on procedures. The utility section acts as a growing library of support procedures for all modules. By looking at the utility library, we can quickly see what relations are available. Relations already available do not have to be redeveloped.

8.3 PROGRAMMING-EFFICIENCY TECHNIQUES

Not only do programs need to be laid out clearly, they also should be designed efficiently. An efficient Prolog program resolves questions rapidly and uses computer time productively. This section introduces some key programming-efficiency techniques.

A. Procedural Program Avoidance

An efficient program allows Prolog's unification mechanism to do the work. The unification mechanism is the built-in set of matching procedures inherent in Prolog. To capitalize on Prolog's built-in matching procedures and thus improve efficiency, we write the program declaratively and avoid procedural programs.

For example, we wish to develop a list-processing relation that tests whether two lists are equal. One way to write this is:

```
equal(L1,L2):-
      length(L1,N1),
      length(L2,N2),
      N1 =:= N2,
      sublist(L1,L2).
```

This definition works. It says "list L1 and L2 are equal if they have the same length, and L1 is a sublist of L2." Although the relation performs correctly, it is inefficient. The above relation is a poor way to define the list equality, because it is procedural, and does not utilize Prolog's unification mechanism. A better way to write the relation is:

```
      equal(X,X).
```

We are done! If we have the program:

```
      . . .,
      X = [a,b,c,d]
      read(Y),
      equal(X,Y),
      . . .
```

Prolog will automatically test the length of lists **X** and **Y** and compare the elements in each list. No explicit procedure needs to be written, because Prolog's unification mechanism does the work for us.

B. More Variables

Efficiency improves when we use more variables instead of calling additional clauses. A classic example of this technique is the *list accumulator*. The accumulator keeps track of the "result so far." If the clause succeeds, the result is instantiated to the final accumulated list.

We used an accumulator in section 3.4B with the **reverse** relation. The original **reverse** relation was:

```
reverse([],[]).
reverse([Head|Tail],ReverseList):-
      reverse(Tail,ReverseTail),
      append(ReverseTail,[Head],ReverseList).
```

which says, "To reverse a list, first reverse the tail of the list, and then append the head onto the reversed tail."

To improve efficiency, we used an accumulator:

```
reverse([],Result,Result).
reverse([H|T],Accumulator,Result):-
            reverse(T,[H|Accumulator],Result).
```

With the first relation, the question

```
?- reverse([b,a,t,e],X).
```

gives ***X = [e,t,a,b]***. With the second relation, the question

```
?- reverse([b,a,t,e],[],X).
```

gives the identical result, ***X* = *[e,t,a,b]***.

This second **reverse** procedure uses more variables, having an arity of three, while the first relation has an arity of two. Using an additional variable eliminated the **append** clause from the rule. With **append** eliminated, Prolog takes fewer steps to reach the final answer. You may wish to trace through both questions to confirm this yourself.

C. Base and Recursive Cases

We use recursion in 1) list processing, and 2) some numerical calculations. To properly implement recursion, we always need a base case and a recursive case. The base case is usually a fact that prevents Prolog from going into an infinite loop. Base cases differ depending on whether the relation is for list processing or numerical calculation. Table 8.1 shows some typical base cases.

Table 8.1. Typical base cases used in recursion.

TYPE OF RECURSION	TYPICAL BASE CASES
Numerical Calculation	N = 0
	N = 1
List Processing	empty list [ ]
	single element list [_]

The **length** relation possesses an example of a list-processing base case:

```
length([],0).                       % base case
length([_],1).                      % base case
length([_|Tail],N):-                % recursive case
      length(Tail,TailLength),
      N is TailLength + 1.
```

An example of a numerical base case where N = 1 is the factorial relation:

```
factorial(0,1):- !.                 % base case
factorial(N,R):-                    % recursive case
      NN is N - 1,
      factorial(NN,TemporaryR),
      R is TemporaryR*N.
```

When developing any recursive rule, we should identify the base and recursive cases clearly and correctly. The general rule for more efficient programs is to place the base case *before* the general case:

```
/**********************************************************
    More efficient ordering with base case first
 **********************************************************/

      member(X,[X|_]).          % base case
      member(X,[_|Tail]):-      % recursive case
            member(X,Tail).
```

```
/*******************************************************
    Less efficient ordering with base case last
 *******************************************************/

      member(X,[_|Tail]):-         % recursive case
            member(X,Tail).
      member(X,[X|_]).             % base case
```

D. Left, Center and Tail Recursions

An example of *left recursion* is the **length** relation:

```
length([],0).
length([_|Tail],N):-
      length(Tail,TailLength),          % left recursive call
      N is TailLength + 1.
```

This is "left" recursion because the first sub-goal in the clause is a recursive call. In *tail recursion*, the last sub-goal in the clause is the recursive call, as shown in the **listwrite** relation:

```
listwrite([]).
listwrite([Head|Tail]):-
```

```
        write(Head),nl,
        listwrite(Tail).                 % tail recursion
```

An example of *center recursion* is the **factorial** relation:

```
    factorial(0,1):- !.
    factorial(N,R):-
        NN is N - 1,
        factorial(NN,TemporaryR),      % center recursive call
        R is TemporaryR*N.
```

This is "center" recursion because the recursive call is not the first or the last sub-goal in the clause; instead, it is somewhere "in the center."

When developing recursive rules, we should use *tail recursion* if possible. Tail recursion has several key advantages over left and center recursions. It:

- reduces the threat of an infinite loop;
- utilizes less computer memory; and
- traverses less search space to find the answer.

These points are addressed in more detail below.

1. Infinite Loops with Left Recursion

Left recursion needs to be done with caution, especially by a beginning programmer. Since the very first call in the clause is recursive, the threat of an infinite loop is high. In section 2.1B, we dealt with a pumping network that had a recursive rule to determine if one pumping station was upstream of another. The worst way to order the goals was:

```
upstream(X,Y):-
        upstream(Z,Y),        % a left recursive call
        feeds_into(X,Z).
upstream(X,Y):-
        feeds_into(X,Y).
```

We coupled these rules with the **feeds_into** database:

```
feeds_into(a,c).
feeds_into(a,d).
feeds_into(b,d).
feeds_into(b,e).
feeds_into(c,f).
feeds_into(d,f).
feeds_into(f,g).
```

With the program above, any **upstream** question, such as:

```
?- upstream(a,b).
```

will send Prolog into an infinite loop. If we instead use tail recursion,

```
upstream(X,Y):-
        feeds_into(X,Z),
        upstream(Z,Y).      % tail recursion
upstream(X,Y):-
        feeds_into(X,Y).
```

Prolog will correctly answer the question.

Because the threat of infinite loops is higher with left recursion and center recursion, we should use tail recursion wherever possible.

2. Computer Memory Consumption with Recursion

Compared to tail recursion, both left and center recursions consume much more computer memory. With left or center recursion, Prolog retains all the interim information, backtracking, and variable instantiations for later use in the clause. With tail recursion, however, the recursive call completes the clause. Thus, Prolog need not retain any interim information; all processing in the clause is complete.

With left recursion and center recursion, Prolog forces a complete copy of all remaining sub-goals within the clause into the computer memory with each recursive call. These copies take up a lot of memory. With tail recursion, no sub-goals remain, and hence the memory requirement is minimized. For these reasons, tail recursion is favored over both left and center recursions.

3. Search Space Traversed with Recursion

Both left and center recursions traverse more search space than tail recursion. Therefore, they normally will take more steps and use more time in finding the answer. Let us consider the predicate **remove_duplicate** introduced in section 3.4G, defined as:

```
remove_duplicate([],L,L).
remove_duplicate([H|T],A,L):-
      member(H,A),
      remove_duplicate(T,A,L).
remove_duplicate([H|T],A,L):-
      remove_duplicate(T,[H|A],L).
```

The second **remove_duplicate** clause above contains a tail-recursive call. The above program gives the following answer to the question:

```
?- remove_duplicate([m,o,o],[],L).
L = [o,m]
```

Prolog goes through twenty-two search blocks to reach the answer. Figure

8.1 shows the complete trace in block form. The same trace is shown below in "CALL, REDO, EXIT, FAIL" form:

```
 (0) CALL: remove_duplicate([m,o,o],[],_x1)
      (1) CALL: member(m,[])
      (1) FAIL: member(m,[])
      (2) CALL: remove_duplicate([o,o],[m],_x1)
           (3) CALL: member(o,[m])
                (4) CALL: member(o,[])
                (4) FAIL: member(o,[])
           (3) FAIL: member(o,[m])
           (5) CALL: remove_duplicate([o],[o,m],_x1)
                (6) CALL: member(o,[o,m])
                (6) EXIT: member(o,[o,m])
                (7) CALL: remove_duplicate([],[o,m],_x1)
                (7) EXIT: remove_duplicate([],[o,m],[o,m])
           (5) EXIT: remove_duplicate([o],[o,m],[o,m])
      (2) EXIT: remove_duplicate([o,o],[m],[o,m])
 (0) EXIT: remove_duplicate([m,o,o],[],[o,m])
```

Prolog reports the answer, *L = [o,m]*.

Now let us make the second clause left-recursive. The entire procedure becomes:

```
remove_duplicate([],L,L).
remove_duplicate([H|T],A,L):-
     remove_duplicate(T,A,L),
     member(H,A).
remove_duplicate([H|T],A,L):-
     remove_duplicate(T,[H|A],L).
```

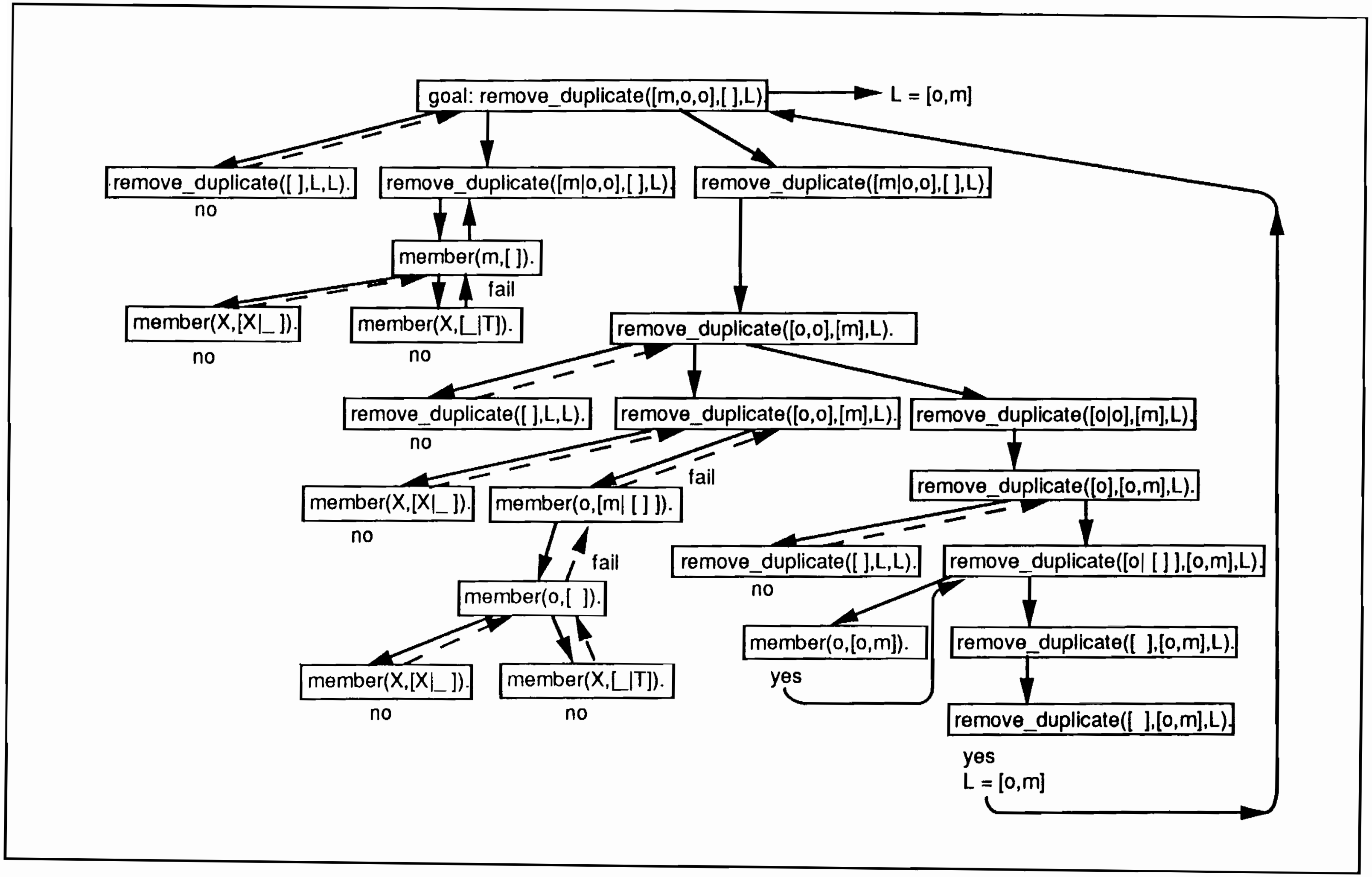

goal: remove_duplicate([m,o,o],[ ],L).
L = [o,m]
remove_duplicate([ ],L,L).
no
remove_duplicate([m|o,o],[ ],L).
remove_duplicate([m|o,o],[ ],L).
member(m,[ ]).
fail
member(X,[X|_]).
no
member(X,[_|T]).
no
remove_duplicate([o,o],[m],L).
remove_duplicate([ ],L,L).
no
remove_duplicate([o,o],[m],L).
remove_duplicate([o|o],[m],L).
member(X,[X|_]).
no
member(o,[m| [ ]]).
fail
fail
member(o,[ ]).
member(X,[X|_]).
no
member(X,[_|T]).
no
remove_duplicate([o],[o,m],L).
remove_duplicate([ ],L,L).
no
remove_duplicate([o| [ ]],[o,m],L).
member(o,[o,m]).
yes
remove_duplicate([ ],[o,m],L).
remove_duplicate([ ],[o,m],L).
yes
L = [o,m]

Figure 8.1. The trace of tail-recursive remove_duplicate procedure in block form.

If we ask the same question with this left-recursive procedure, we get the same result,

```
?- remove_duplicate([m,o,o],[],L).
L = [o,m]
```

but Prolog goes through *many more* steps to get to the answer. The trace of this question is shown below in CALL, REDO, EXIT, FAIL form. The procedure is so much longer that it is not even practical to write it in block form:

```
(0) CALL: remove_duplicate([m,o,o],[],_x1)
      (1) CALL: remove_duplicate([o,o],[],_x1)
            (2) CALL: remove_duplicate([o],[],_x1)
                  (3) CALL: remove_duplicate([],[],_x1)
                  (3) EXIT: remove_duplicate([],[],[])
                  (4) CALL: member(o,[])
                  (4) FAIL: member(o,[])
                  (5) CALL: remove_duplicate([],[o],_x1)
                  (5) EXIT: remove_duplicate([],[o],[o])
            (2) EXIT: remove_duplicate([o],[],[o])
            (6) CALL: member(o,[])
            (6) FAIL: member(o,[])
            (7) CALL: remove_duplicate([o],[o],_x1)
                  (8) CALL: remove_duplicate([],[o],_x1)
                  (8) EXIT: remove_duplicate([],[o],[o])
                  (9) CALL: member(o,[o])
                  (9) EXIT: member(o,[o])
            (7) EXIT: remove_duplicate([o],[o],[o])
      (1) EXIT: remove_duplicate([o,o],[],[o])
      (10) CALL: member(m,[])
      (10) FAIL: member(m,[])
      (1) REDO: remove_duplicate([o,o],[],[o])
            (7) REDO: remove_duplicate([o],[o],[o])
                  (9) REDO: member(o,[o])
                        (11) CALL: member(o,[])
                        (11) FAIL: member(o,[])
                  (9) FAIL: member(o,[o])
                  (12) CALL: remove_duplicate([],[o,o],_x1)
                  (12) EXIT: remove_duplicate([],[o,o],[o,o])
            (7) EXIT: remove_duplicate([o],[o],[o,o])
      (1) EXIT: remove_duplicate([o,o],[],[o,o])
      (13) CALL: member(m,[])
      (13) FAIL: member(m,[])
      (14) CALL: remove_duplicate([o,o],[m],_x1)
            (15) CALL: remove_duplicate([o],[m],_x1)
                  (16) CALL: remove_duplicate([],[m],_x1)
                  (16) EXIT: remove_duplicate([],[m],_x1)
                  (17) CALL: member(o,[m])
                        (18) CALL: member(o,[])
                        (18) FAIL: member(o,[])
                  (17) FAIL: member(o,[m])
                  (19) CALL: remove_duplicate([],[o,m],_x1)
                  (19) EXIT: remove_duplicate([],[o,m],[o,m]
            (15) EXIT: remove_duplicate([o],[m],[o,m])
            (20) CALL: member(o,[m])
                  (21) CALL: member(o,[])
                  (21) FAIL: member(o,[])
            (20) FAIL: member(o,[m])
            (22) CALL: remove_duplicate([o],[o,m],_x1)
                  (23) CALL: remove_duplicate([],[o,m],_x1)
                  (23) EXIT: remove_duplicate([],[o,m],[o,m])
                  (24) CALL: member(o,[o,m])
                  (24) EXIT: member(o,[o,m])
            (22) EXIT: remove_duplicate([o],[o,m],[o,m])
      (14) EXIT: remove_duplicate([o,o],[m],[o,m])
(0) CALL: remove_duplicate([m,o,o],[],[o,m])
```

Prolog reaches the answer, ***L* = *[o,m]***, but the left-recursive procedure covers *over 300% more search space* to reach the same answer!

Left and center recursions inherently traverse more search space than tail recursion. In the left-recursive **remove_predicate** procedure above, we "forced" more recursion before testing if the head was a **member** of the accumulator. By using this recursion, we force Prolog blindly into more branches of the search tree. As it turns out, the majority of those branches simply fail. Prolog is forced to backtrack until it reaches an acceptable answer. Meanwhile, precious time is wasted.

Sometimes left and center recursions are essential, such as in the length and **factorial** relations discussed in section 8.3D. But if we have a choice between: 1) tail recursion and 2) left or center recursion, we should favor tail recursion.

E. Sub-goal Ordering within a Clause

The ordering of sub-goals within a clause is one of the most critical points affecting program efficiency. We saw this effect when comparing left, center, and tail recursions, where we favor tail-recursive ordering.

Why is sub-goal ordering so important? It directly affects the search tree. Depending on the ordering, Prolog may go into a long, drawn-out search and eventually fail in that branch of the search tree. Before Prolog embarks on a search, it has no prior knowledge of whether or not the search is doomed to failure. If Prolog finally does fail, it has to backtrack. This wastes time. Proper ordering of sub-goals can prevent Prolog from wasting time in a doomed search.

The rule of thumb in sub-goal ordering is:

fail as soon as possible

Following this rule will "prune" the tree and minimize fruitless search.

To implement this critical rule of thumb, we recommend the following clause structure, seen previously, in section 8.2 D. The general format is:

```
clause:-
      tests,
      action.
```

This format places all tests first to determine the applicability of the clause. If the clause is applicable, action is taken. If the clause is not applicable, minimal time is spent within the clause. As a more specific example, let us consider the clause introduced in section 8.2D.

```
clause:-
      test_for_laminar_flow,
      test_for_chemical_reaction,
      test_for_single_phase,
      take_action.            % the clause is now applicable
```

This clause tests for laminar flow, a chemical reaction, and a single phase. If all tests come out as specified, the clause is applicable to the situation and Prolog takes action.

If we desire to "lock-in" a single rule, then we can include a cut:

```
clause:-
      tests,
      !,
      action.
```

This format, with or without the cut, promotes ***fail as soon as possible***. Essential backtracking occurs as soon as possible, and no time is wasted searching a tree doomed to failure. Once we get passed the key tests, we know we can apply the rule and go through the full search.

F. Iteration and Recursion

In section 3.3B, we compared recursion with iteration. Recursion ***remembers*** each variable in loop. Iteration ***replaces*** all variables in a loop. Continued recursive calls place a growing load on the computer memory. Continued iterative calls do not increase the memory consumption at all. Thus, we favor iteration over recursion wherever possible.

Using iteration instead of recursion relies on the procedural aspects of Prolog to do the work. Because iterative loops are not very declarative, they can sometimes be hard to read and understand. Nevertheless, for complex operations, an iterative loop can save valuable computer memory.

How do we implement iteration in a recursion-dominated language like Prolog? We use the **fail** predicate to force backtracking. Recall that when Prolog backtracks, it releases variable instantiations inside that clause.

1. Iteration through the repeat-fail Combination

The repeat-fail procedure is particularly useful for the input of data into programs. Consider the following clause that accepts data from the user:

```
input:-
     repeat,
     read_input_from_user,
     process_some_data,
     report_interim_results,
     write('Do you wish to enter main program?'),nl,
     write('Enter y or n'),nl,
     read(y),!,
     main_program.
```

The **input** clause does some preliminary analysis and asks for approval to enter the main program. If the user does not grant approval (i.e., an **n** is entered), the **read(y)** fails and Prolog backtracks to the **repeat** predicate. All variables are uninstantiated and the computer memory starts over clean.

If we call **input** recursively when **n** is entered, the computer memory continues to grow, since a complete copy of the goals is forced into the memory. With the repeat-fail combination, no extra memory is consumed. As a final point, we strongly recommend placing a cut **!** after the **read** statement. This prevents Prolog from backtracking into the **repeat** predicate, protecting the program from a repeat-fail infinite loop.

2. Iteration through assert Followed by the retract-fail

Another way to implement iteration in Prolog is through database manipulation. Specifically, we **assert** facts and **retract** them one by one, and then fail, forcing backtracking to other facts. After backtracking, the procedure is repeated. Each time, the computer memory starts out clean.

Assume that we have a list of metals being considered for an application. We write the following recursive program:

```
choose_metals([]).
choose_metals([Head|Tail]):-
        perform_analysis(Head),
        choose_metals(Tail).
```

If the **perform_analysis** clause is complex, this recursive definition can take up a lot of computer memory. A way to implement the procedure iteratively is:

```
initialize_database([Head|Tail]):-
        assertz(metal(Head)),
        initialize_database(Tail).
initialize_database(_).
```

```
choose_metals:-
        metal(X),
        perform_analysis(X),
        retract(metal(X)),
        fail.
choose_metals.
```

If we have the list

```
[iron,zinc,copper,steel]
```

the initialization clause asserts the following facts into the database:

```
metal(iron).
metal(zinc).
metal(copper).
metal(steel).
```

The first time through the **choose_metals** clause, **X** is instantiated to **iron**. Prolog performs the analysis, and then retracts the fact **metal(iron)** from the database. Prolog hits the **fail** predicate, and backtracks to **metal(X)**. This goal succeeds, and **X** is instantiated to **zinc**, since **metal(iron)** is no longer in the database. The same iterative loop occurs, until **X** has been instantiated to all four metals, i.e., **iron**, **zinc**,

copper, and **steel**. The end result is an iterative, rather than a recursive analysis of all four metals.

G. Drivers

When developing a Prolog program, we often use a "driver" clause that always succeeds. This driver clause controls the whole program. A driver clause usually calls an entire module, or implements a primary objective in the program. We shall see a practical example of a driver clause in section 15.3B, where we use driver clauses to control an entire expert system.

As a general example, let us consider the following program:

```
driver:-                            % overall program driver
      get_input,                    % input driver
      generate_potential_solutions, % solution generator driver
      test_potential_solutions,     % preliminary tester driver
      check_output_with_user,       % user confirmation driver
      retest_potential_solutions,   % rigorous tester driver
      final_output.                 % final output driver
```

The names of each driver clause clearly spell out the major objectives of this program.

Driver clauses have many advantages:

(1) This one clause reveals the entire structure of the program. The structure clearly spells out each clause, and its objectives.

(2) The driver format allows the program to be developed in modules by conveniently segmenting the program. Each driver can be developed as a module.

(3) Special clauses within each driver easily handle errors and irregular events.

(4) We can write each driver such that the program always succeeds, and thus avoid the abrupt **no** that results when Prolog fails.

(5) If a solution is questionable or if no solution exists, Prolog can report the result with an explanation rather than simply answering **no**. The report can be generated from the driver where the difficulty arises.

(6) The program is more flexible and more easily modified. We can isolate problems or areas for expansion more easily, since the program is separated into drivers. We can then focus on the individual driver that needs adjustment, rather than trying to tackle one huge program.

8.4 PROGRAM DEVELOPMENT

There are a number of points to take into account to aid in overall program development.

A. Evolutionary Approach

Large Prolog programs, especially expert systems, require development in an evolutionary fashion. The following steps are typical in program development:

(1) rapidly develop prototype;
(2) assess performance and make adjustments;
(3) develop large system;
(4) commercialize system; and
(5) maintain and expand system.

The rapid prototype stage is important to program development. The two main goals of this stage are to:

(1) get a system running; and
(2) assess needs in knowledge representation.

With dedicated programming, a good prototype system can usually be developed in two to six months (depending on the complexity of the problem). Having a prototype running early maintains interest in the project.

As the prototype grows, the program almost inevitably evolves. Unanticipated situations develop. In some situations, certain clauses are not applicable. "Holes" in the knowledge base become evident. We realize that we have not fully characterized the problem; we need to perform further analysis and adjust the program.

The prototype should be running *correctly* before we attempt to develop the large system. A frequent error is to rush full-system development before working out some of the tougher problems in the prototype. But the problems do not go away; they just get tougher to solve.

Evolutionary programming can be a surprise to programmers used to conventional languages such as FORTRAN. Experienced FORTRAN programmers frequently use a regimented, project-style approach to writing programs. They: 1) develop the scientific, engineering, and user-oriented objectives, 2) develop the algorithm, and 3) write the code. In FORTRAN, if the algorithm is inaccurate or incomplete, the program will not run correctly.

Prolog is different. It may run well (depending on the problem) with an incomplete knowledge base. And as we test the program, we usually find that the rules need evolutionary adjustments. Fortunately, Prolog facilitates evolutionary programming, since it is written in clause form.

B. Modular Development Using Drivers

We discussed using drivers in the previous section. One advantage of an overall program driver is that it allows the program to be developed in modules. Modular development has the following advantages:

(1) it generally takes less overall time to develop a modular program;

(2) the corrections and adjustments to a modular program are easier;

(3) the program is more flexible and easier to read and understand; and

(4) program features are more easily expanded.

Since Prolog is written in clause form, modular development is easy and natural. We discuss modular development using drivers with an actual expert system in section 15.3B.

C. Problem Characterization

As more complex engineering and scientific applications of Prolog have grown, so has the importance of problem characterization. In the past, programmers tended to spend inadequate time up-front to properly analyze and represent the problem at hand. Instead, they have taken a poorly represented problem and relied on sophisticated search techniques, statistics, and programming tricks to solve the problem.

Fortunately, this tendency is changing. More emphasis is being put on problem characterization rather than Prolog search techniques. The general rule for problem characterization that we recommend is:

Represent knowledge used to solve the problem
so well that the Prolog search required is easy.

Of course, this recommendation is an ideal. Knowledge is hard to characterize. In addition, we may have a problem that has inherent uncertainties. In that case, it may be impossible to completely characterize the knowledge, and we will have to rely more on the search. Indeed, the complexity of artificial intelligence problems is so high that some aspects of search are absolutely essential.

Usually, though, the better we can characterize the problem, 1) the easier it is to write the program, 2) the more accurate the results, and 3) the less time it takes for the program to solve it.

D. Exercises

8.4.1 Add comments that explain what each rule accomplishes in the following relations:

```
A. append([],L,L).
   append([H1|T1],L2,[H1|T3]):-
         append(T1,L2,T3).
```

```
B. remove_duplicate([],L2,L2).
   remove_duplicate([H|T],Accumulator,L2):-
         member(H,Accumulator),
         remove_duplicate(T,Accumulator,L2).
   remove_duplicate([H|T],Accumulator,L2):-
         remove_duplicate(T,[H|Accumulator],L2).
```

8.4.2 The length relation, developed in section 3.4F, has a left-recursive definition:

```
length([],0).
length([_|Tail],N):-
            length(Tail,TailLength),
            N is TailLength + 1.
```

Redefine the relation using tail recursion.

8.4.3 Convert the following clause to multiple clauses by eliminating all semicolon operators from the relation:

```
A:-
      (B;C),
      D;
      ((E,F);(G,H));
      I.
```

8.5 CHAPTER SUMMARY

- Principles of good Prolog programming include:
    - Correctness
    - Efficiency
    - Readability
    - Modifiability
    - Robustness
    - Documentation

- Principles of good program layout include:
    - Using of comments liberally
    - Making terms meaningful
    - Having a consistent layout
    - Keeping clauses short
    - Limiting the use of the ; operator
    - Using **assert**, **retract**, and ! carefully
    - Placing utility predicates in their own section

- Principles of program efficiency include:
    - Making programs declarative rather than procedural
    - Using more variables rather than clauses
    - Identifying the base and recursive cases
    - Distinguishing among left, center and tail recursions

        - Tail recursion is the least likely to result in an infinite loop.
        - Tail recursion consumes the least amount of memory.
        - Tail recursion traverses less search space.
    - Ordering sub-goals within a clause efficiently
    - Substituting iteration for recursion
    - Using drivers to control the entire program

- Principles of overall program development include:
    - An evolutionary approach, further characterizing the problem as the program unfolds
    - Modular program development
    - Proper problem characterization

A LOOK AHEAD

We now know how to develop programs in Prolog. Now we can switch gears and focus on applications. But before we focus on applications in-depth, let us develop an understanding of another prominent AI language, LISP. In chapter 9, we introduce the language LISP, and contrast LISP with Prolog. We identify the advantages and limitations of both languages. Based on the characteristics of each language as well as the chemical engineering problem that we are trying to solve, we make suggestions on whether to choose Prolog or LISP as the language of choice. Once we have a clear understanding of the two main AI languages (Prolog and LISP), we can move directly into AI applications in chemical engineering.

REFERENCES

Bratko, I., *Prolog Programming for Artificial Intelligence*, second edition, pp. 187-208, Addison-Wesley, Reading, MA (1990).

Clocksin, W.F. and C.S. Mellish, *Programming in Prolog*, third edition, pp. 175-200, Springer-Verlag, New York, NY (1987).

Garavaglia, Susan, *Prolog Programming Techniques and Applications*, pp. 315-338, Harper & Row Inc., New York, NY (1987).

Schnupp, P. and L. Bernhard, *Productive Prolog Programming*, pp.181-198, Prentice-Hall, Englewood Cliffs, NJ (1986).

Sterling, L. and Shapiro E., *The Art of Prolog*, pp. 95-99, 192-203, MIT Press, Cambridge, MA (1986).

9

PROLOG AND LISP

A number of programming languages, such as LISP, Prolog and object-oriented C, have been used in artificial intelligence. By far, the two most important languages for symbolic computing are Prolog and LISP. LISP was developed in the United States, and is the language of choice for most AI applications in the U.S.. Prolog was introduced in Europe, and both the Europeans and the Japanese tend to favor Prolog over LISP.

In the preceding chapters, we discussed how to program in Prolog. This chapter gives a brief introduction to LISP, contrasts it with Prolog, and then discusses some future trends for programming language development.

9.1 LISP

A. Origins of LISP

LISP stands for *LISt Programming*. The language was developed at MIT in 1959, and was formally presented by John McCarthy in 1960. LISP is the second oldest computer language around-- the oldest is FORTRAN.

Throughout the 1960's and 1970's, researchers quietly worked on optimizing LISP. They developed and enhanced software tools and refined the programming environment (the hardware, keyboard, and man-machine interface). During this period, AI was still a sleepy research "cult" in academia, and was too expensive for industrial applications.

The explosion of AI into the industrial markets in the late 1980's did not result from any spectacular AI software breakthroughs. Instead, it happened because of the growing availability of low-cost and powerful hardware. Initial investment in a LISP system in 1975 may have ranged from $1 million to $3 million; in 1990, we can implement a LISP system on a personal computer for an initial investment of less than $10,000 . In addition, because the day is approaching where LISP will be usable on any personal computer, a much wider range of users will have access to AI programs.

LISP is designed specifically for symbolic manipulation. It has more built-in functions than any other language. Its computing environment, including editing and debugging tools, is well developed and very effective. LISP is an excellent language for developing large, complicated AI programs.

B. The Language

LISP is a mature language. Many different versions of LISP have floated around during the 1970's and 1980's (e.g., FRANZLISP, INTERLISP, Common LISP, etc). Most versions are "incompatible" with each other; although they are all LISP dialects, programs developed with one version cannot run on a computer operated by another version.

Fortunately, the incompatibility problem is changing. In the "quiet" stages of the 1960's to the early 1980's, researchers at MIT, Stanford, Xerox Palo Alto Research Center, Carnegie Mellon, and other AI laboratories worked on developing tools for enhancing LISP software development. This effort produced some of the most powerful software-development tools in the world. Now that AI has "gone commercial", some essential standardization is setting in, and *Common LISP* is becoming the standard.

1. LISP Data Types

LISP is a complex procedural language having over 300 built-in functions. These functions operate on and manipulate data structures. LISP has only two data types: *atoms* and *lists*. An example of a list with three atoms is:

```
( BUTANE  PENTANE  HEXANE )
```

The following list also contains three atoms:

```
( 2  3.55 4.1)
```

Numbers in LISP are also atoms. They are *numeric atoms*, or more simply, *numbers*. In contrast, atoms such as **BUTANE** and **PENTANE** are *symbolic atoms*, or more simply, *symbols*.

Figure 9.1 summarizes the data types in LISP. All expressions are made up of either atoms or lists. Atoms can be symbols or numbers, and within numbers, we can have integers or real (floating-point) numbers.

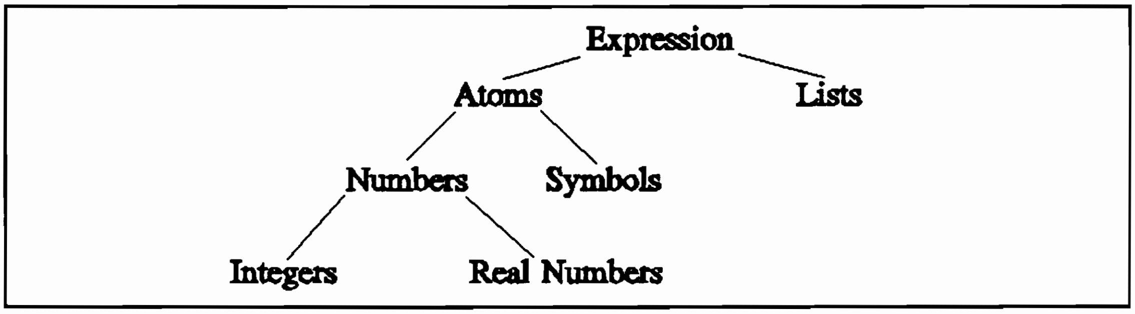


Figure 9.1. Data types in LISP.

Recall in Figure 2.1 that in Prolog there are a number of distinct data types (atoms, numbers, lists, structures, etc.). Consequently, Prolog programs do not rely too heavily upon lists. In LISP, however, the list is the dominant data structure.

As in Prolog, LISP supports nested lists:

```
( (BUTANE PENTANE) HEXANE )
```

The first element is the list **(BUTANE PENTANE)**. The second element is the atom **HEXANE**. One strength of LISP is its ability to amass and access large amounts of data in a list. A typical data structure in LISP is the "property list":

```
((MATERIAL  N-BUTANE)
 (MOLECULAR-WEIGHT 58.124)
 (BOILING-POINT  R  490.8)
 (CRITICAL-TEMPERATURE  R  765.3)
 (CRITICAL-PRESSURE  PSIA  550.7)
 (CRITICAL-COMPRESSIBILITY-FACTOR  0.274)
 (ANTOINE-CONSTANTS (A1  5.741624)
                    (A2  4126.385)
                    (A3  409.5179))
 (HEAT-CAPACITY-CONSTANTS  (C1  20.79783)
                           (C2  0.3143287E-01)
                           (C3  0.1928511E-04)
                           (C4  -0.4588652E-07)
                           (C5  0.2380972E-10)))
```

2. Introduction to Built-in Procedures

LISP has an extensive number of built-in procedures. Indeed, the heart of LISP programming is invoking procedures that process data to give desired results. LISP uses *prefix notation* to identify these procedures. A LISP procedure can be viewed as:

(*PROCEDURE* A_1 A_2 A_3 A_4 ... A_i)

where the A_i's are *arguments*.

For example, consider the addition procedure:

```
( +  1  2 )
```

The first element of the list, the plus sign, names the procedure (in this case, addition). The subsequent numbers, 1 and 2, are the arguments in the list to be added. Now consider the following addition procedure:

```
( + ( + 1 2 ) 3)
```

This procedure is the mathematical equivalent of (1 + 2) + 3, and is called a *nested* procedure in LISP, since one mathematical procedure is embedded inside another.

Addition is a built-in procedure for numerical manipulation. LISP also has built-in procedures for symbolic processing. The three most basic procedures are:

- **CAR** - returns the first element of a list.
- **CDR** - returns all of the list except the first element.
- **CONS** - takes two atoms or lists and combines them into a bigger list.

CAR, **CDR**, and **CONS** are each a type of procedure called a *function.* In LISP, a function is a procedure that returns only one value based on its arguments. If the procedure does anything in addition to returning a single value, it is not a function.

The built-in function **CAR** means "Contents of the Address Register," while **CDR** stands for "Contents of the Decrement Register." If we view these functions in terms of Prolog procedures, we can say that **CAR** returns the head of a list, while **CDR** returns the tail. The **CONS** function parallels the Prolog dot operator as a means of building up lists.

Below are some examples using these built-in LISP functions. In Prolog, the **?-** symbol is a prompt to indicate that the computer is waiting for input from the user. In LISP, we use the -> prompt. The -> prompt means "evaluate the following statement."

```
-> (CAR '(BUTANE PROPANE ETHANE))
```

```
BUTANE
-> (CDR '(BUTANE PROPANE ETHANE))
(PROPANE ETHANE)
-> (CONS 'BUTANE '(PROPANE ETHANE))
(BUTANE PROPANE ETHANE)
-> (CONS (CDR '(BUTANE PROPANE ETHANE))
         (CAR '(BUTANE PROPANE ETHANE)))
(PROPANE ETHANE BUTANE)
```

We see something different in these statements: the presence of the single-quote character, '. Why do we need this single quote in LISP? The single quote is used to tell LISP to treat the list following the quote as *data to be manipulated rather than another embedded procedure to be invoked*. So, if we write:

```
->(CAR (CDR '(BUTANE PROPANE ETHANE)))
```

LISP responds with:

```
PROPANE
```

Now, however, if we write the same statement with a single quote in front of the embedded **CDR** procedure:

```
    -> (CAR '(CDR '(BUTANE PROPANE ETHANE)))
```

LISP responds with:

```
    CDR
```

Why? What happened here? In the first statement, with no single-quote mark, LISP views the embedded list, **(CDR '(BUTANE PROPANE ETHANE))** as a procedure to be *executed*. Thus, LISP executes the inner procedure first, takes the **CDR** of **(BUTANE PROPANE ETHANE)** and returns **(PROPANE ETHANE)**. LISP then takes the **CAR** of this list, giving the result, **PROPANE**.

In the second procedure, however, there is a single quote in front of the embedded **CDR** list. Therefore, LISP treats this list as *data to manipulate* rather than a *procedure to be execute*. The **CAR** of the list **'(CDR '(BUTANE PROPANE ETHANE))** is simply the atom **CDR**, and this is the result that LISP reports.

3. Built-in Procedures in LISP and Prolog

Functions, an important aspect of LISP, do not exist in Prolog. For example, we ask LISP to evaluate the statement **(CAR '(BUTANE PROPANE ETHANE))**, and LISP returns the answer *in place of the statement*. The statement is ***(CAR '(BUTANE PROPANE ETHANE))***; this statement "disappears" and is "forgotten." Instead, the atom ***BUTANE*** stands in its place. The atom

BUTANE is formed by taking the CAR of the list ***(BUTANE PROPANE ETHANE)***. The CAR returns the first element, in this case, the atom ***BUTANE***.

This aspect of functions, replacing the statement with the value that the statement represents, does not exist in Prolog. Prolog utilizes predicate logic only, and a clause cannot be evaluated and *replaced* by a single value. For example, if we wish to add 2 and 3, in LISP we use the addition function:

```
-> ( + 2 3 )
6
```

The complete statement is replaced by the number 6. In Prolog, however, we write:

```
?- add(2,3,X).
X = 6 ;
no
```

We invoke the **add** relation, and the variable **X** is instantiated to the number **6**. The **add** statement has *not* been replaced.

Most built-in procedures in LISP exist in the form of functions. For example, we can use the built-in LISP function **reverse**:

```
-> (reverse '(A B C D) )
```

```
(D C B A)
```

Many LISP functions exist that do the same thing as Prolog predicates defined in Table 3.7. However, these LISP functions are *built-in*, while those Prolog relations must be explicitly included in our database. In practice, the difference is small. When we build large Prolog programs, we normally have a "library" of utility relations that we rely upon. Once we write these relations, we can retain them for future use in our Prolog utility library. The net effect is that we almost view the relations as being built-in. Table 9.1 shows some built-in LISP functions and their Prolog counterparts.

Table 9.1. Built-in LISP functions and corresponding Prolog relations.

LISP		PROLOG	
FUNCTION	RESULT	RELATION	RESULT
(APPEND '(A B C) '(D E F))	(A B C D E F)	append([a,b,c], [d,e,f], X).	X = [a,b,c,d,e,f]
(REVERSE '(A B C D))	(D C B A)	reverse([a,b,c,d],X).	X = [d,c,b,a]
(CONS 'A '(B C D))	(A B C D)	add(a,[b,c,d],X).	X = [a,b,c,d]
(TAILP '(B C) '(A B C D))	true	sublist([b,c], [a,b,c,d]).	yes
(CAR '(A B C D))	A	first(X, [a,b,c,d]).	X = a
(LAST '(A B C D))	D	last(X, [a,b,c,d]).	X = d
(REMOVE-DUPLICATES '(A A B B C C D))	(A B C D)	rem_dup([a,a,b,b,c,c,d],X).	X = [a,b,c,d]
(LENGTH '(A B C D))	4	length([a,b,c,d],X).	X = 4
(ATOM 'A)	true	atom(a).	yes
(INTEGERP '2)	true	integer(2).	yes
(FLOATP '1.3)	true	float(1.3).	yes
(NUMBER '3.14)	true	number(3.14).	yes

LISP and Prolog are different languages, and consequently, many LISP functions and Prolog relations do not have direct counterparts in the other language. Two very common Prolog relations with no direct LISP counterpart are **delete** and **member**. They are explained below.

DELETE

In Prolog, the relation **delete(X,L,Z)** will delete the *first* occurrence of the atom **X** from list **L**, and return the result, **Z**. Thus, the following dialogue results from the **delete** relation:

```
?- delete(a,[a,b,c,d,a],Z).
   Z = [b,c,d,a] ;
   Z = [a,b,c,d] ;
   no
```

In LISP, **DELETE** is different; it deletes *all* occurrences of the desired atom from the list. Thus:

```
-> (DELETE 'A  '(A B C D A)
(B C D)
```

MEMBER

In Prolog, the **member(X,L)** relation is true if the atom **X** is a member of list **L**; otherwise, the relation is false and fails. Thus:

```
?- member(a,[a,b,c,d,e]).
   yes
?- member(c,[a,b,c,d,e]).
   yes
?- member(f,[a,b,c,d,e]).
   no
```

In LISP, the MEMBER relation returns a sublist beginning with the matching element when it finds a match. If the atom is not a member of the list, LISP returns the nil result. Thus:

```
-> (MEMBER 'A '(A B C D E) )
   (B C D E)
-> (MEMBER 'C '(A B C D E) )
   (D E)
-> (MEMBER 'F '(A B C D E) )
   NIL
```

4. Recursive Processing

Like Prolog, LISP supports and relies on recursion. One example of recursion in LISP is the familiar **append** procedure. To recall, the **append** procedure takes two lists and "appends" them together to form a single list. For example, the Prolog response to the question:

```
?- append( [a,b], [c,d], X).
```

is ***X* = *[a, b, c, d]***. The Prolog procedure for **append** (see section 3.4A) is:

```
append( [ ], L, L ).
append( [X|L1], L2, [X|L3] ):-
      append(L1,L2,L3).
```

Although **APPEND** is a built-in procedure in LISP, we write it as:

```
(DEFUN APPEND (X Y)
       (IF  (NULL  X)  Y
            (CONS   (CAR X) (APPEND  (CDR X)  Y))))
```

The **DEFUN** stands for "DEfine FUNction." We use the **DEFUN** statement to custom-design LISP functions. In this case, we are designing the **APPEND** procedure. The idea behind the LISP **APPEND** statement is that appending (A B C) and (D E F) is the same as appending (B C) to (D E F) and then using **CONS** to place A on the front of the result. Because the **APPEND** function is calling the **APPEND** function, the procedure is recursive. The **IF** statement breaks the recursive loop when X is empty (null). It is the base case, saying that if X is null, appending X and Y yields Y.

Recursions in Prolog and LISP are very closely related. To properly perform recursion, we must:

- have a way to tell when we are done recursing;
- break the procedure down into a small operation and the "remaining portion;" and
- reduce the size of the "remaining portion" each time through the recursive loop.

Prolog realizes that it is done recursing by matching the *base case*; in the **append** relation, the base case is **append([ ], L, L)**. LISP realizes that it is done recursing typically through use of the **IF** statement, since LISP is fundamentally a procedural language. In the **APPEND** procedure, we include the statement **(IF (NULL X) Y ...)** to break the recursive loop when **X** is null (i.e., the empty list).

To perform proper recursion, we must also break the procedure down into a small operation and the "remaining portion." In Prolog, we typically attain this goal by separating the head and tail of the list. In **append**, we break the input list down using **[X | L1]**. In LISP, we break the input list down using **(CAR X)** and **(CDR X)**.

With recursion in the **append** relation in both Prolog and LISP, we keep breaking the input list down into smaller remaining portions. The end of the recursive loop is the empty list. When this point is reached, we break the loop.

5. Program Execution and Control in Prolog and LISP

LISP is a procedural language geared for symbolic processing. Prolog is a relational language (with a necessary procedural nature) geared for logic programming. *Prolog has a built-in search and unification (matching) mechanism. LISP does not.* Therefore, when we develop an artificial intelligence program, we must take different approaches depending on whether we use Prolog or LISP. To develop a LISP program, we must define all relations *and explicitly what inference steps the LISP system is to take* to solve the problem. When we develop a Prolog program, we define the relations *only*. We *do not* need to define the inference steps. That task is already done -- the inference mechanism is built right into Prolog.

When executing a Prolog program, Prolog always begins at the top of the program and marches down in search of a match. Prolog's built-in search and unification mechanism uses backward-chaining in a depth-first search mode. If a match occurs, the variable is instantiated. If a clause fails, variables instantiated within that clause become uninstantiated, free variables. Importantly, backtracking occurs *automatically* in Prolog when a fail occurs.

To summarize program control in Prolog, we note that the details of procedurally executing a program are *built into* Prolog. We focus on the relational nature of the problem, and place minimal emphasis on the procedural aspects. We cannot totally remove ourselves from the procedural aspects of Prolog programming, however. For example, we rely on the cut (!) and clause-ordering techniques to adjust and control the program procedurally.

Program control in LISP is much different. LISP is a procedural language, and to properly control a LISP program, we must explicitly state each step that LISP is to take. We can emulate the Prolog-style inference mechanism in LISP. We introduce a procedure, **PROLOG**, to achieve this goal. This procedure takes a list of potential relations to which to expand (analogous to a set of Prolog rules). These relations are stored in a list that is instantiated to the variable **QUEUE**. The final relation, i.e., the goal that we are ultimately trying to satisfy, is instantiated to the variable **GOAL**. The **PROLOG** procedure converts the list **QUEUE** into, as we might expect, a formal queue, i.e., a list representing the set of relations to use to get from the starting relation to the goal. **PROLOG** then calls the procedure **PROLOG-HELP**. The **PROLOG-HELP** procedure examines the queue, and tests the relations in this list for success (i.e., reaching the goal). If the goal is found, **PROLOG-HELP** returns the path travelled. If the goal is not found, the search must continue, and **PROLOG-HELP** expands the queue to a new relation and recursively calls itself:

```
(DEFUN PROLOG (QUEUE GOAL)
      (PROLOG-HELP (LIST (LIST START)) GOAL))

(DEFUN PROLOG-HELP (QUEUE GOAL)
      (COND ((NULL QUEUE) NIL)       ; return nil if queue is null
            ((EQUAL GOAL (CAR (CAR QUEUE))) ; goal is reached
             (REVERSE (CAR QUEUE)))    ; reverse to improve output
            (T (PROLOG-HELP (APPEND (EXPAND (CAR QUEUE)) ; expand if
                                  (CDR QUEUE)      ; goal is not
                           GOAL))))                ; reached
```

In LISP, the semicolon ; functions as a comment indicator. It is analogous to the percentage sign (%) in Prolog. LISP ignores all material between the ; and the end of the line.

The **EXPAND** procedure, which we will not define explicitly, takes the path, determines all "connecting" relations to the current relation, and returns a list of new paths. If the **QUEUE** becomes **NULL**, there is no solution to the problem and **PROLOG-HELP** returns the result **NIL**.

Actually, the procedure defined above is a called a *backward-chaining depth-first search*. At this point in time, we really do not care what that terminology means -- we discuss search in detail in section 11.1. What is important to realize is that this procedure that we just defined is *automatic* in Prolog, but must be explicitly written in LISP. This procedure is actually a modification of that defined by Winston and Horn (1984, pp. 171-175).

6. More Complex Data Representations

Mature LISP systems, such as Common LISP, also support a data representation more powerful than the simple list. They support *records*, sometimes called *structures*. The term *structure* has different meanings in Prolog and LISP: in LISP, a structure is a record, explained below; in Prolog, a structure is a compound data object consisting of a functor and its arguments.

Records are programmed in LISP with slots, default values, and even attached procedures (we discuss these aspects in more depth in 10.2F, frame-based expert systems). To define a record or structure in LISP, we use the **DEFSTRUCT** command with the following format:

DEFSTRUCT *class fields*

where *class* is the name of the record class and *fields* is a list of field names. The advantage of using a record is that it *allows us to extract information from a data structure without worrying about where the information is in that structure*. Consider the following **DEFSTRUCT** for a distillation-column design:

```
(DEFSTRUCT  distillation-column-D2
            (number-of-trays : 40)
            (condenser-type : partial)
            (condenser-coolant : cooling-water)
            (reboiler-type : direct-fired-heater)
            (feed-stage-location : 12))        ; sixth slot of structure
                                               ; feed stage counted from
                                               ; the bottom of column
```

A Prolog list of above information would look something like this:

```
[ distillation_column_D2, [ number_of_trays,40 ],
                          [ condenser_type, partial ],
                          [ condenser_coolant, cooling_water ],
                                        [ reboiler_type, direct_fired_heater ],
                                        [ feed_stage_location, 12 ] ]
```

In our Prolog list, if we want to know the feed-stage location for distillation column D-2, we have a problem; we must remember that the feed-stage location is in the sixth slot of the list. It is tedious to be forced to constantly remember the location of a specific piece of information in a complex list. The program also becomes prone to error.

There are Prolog methodologies that we can use to overcome this problem. Instead of using a single list to represent a block of information, we can use a set of Prolog facts:

```
      distillation_column(d2, number_of_trays, 40).
      distillation_column(d2, condenser_type, partial).
      distillation_column(d2, condenser_coolant, cooling_water).
      distillation_column(d2, reboiler_type, direct_fired_heater).
      distillation_column(d2, feed_stage_location, 12).
```

This information is now integrated into uniform Prolog facts. As we might expect, data retrieval from this set of facts is certainly much easier than retrieval from a complex list. We discuss these concepts further in sections 10.2F and G, as well as 11.2C and D (frame-based systems and object-oriented programming).

If we program in LISP and use the **DEFSTRUCT** tool, we can retrieve this data quickly and easily. Why? Because **DEFSTRUCT** defines a *record*. We can execute a function, called **GET**, to retrieve information from a record:

```
-> (GET distillation-column-D2 feed-stage-location)
```

LISP returns the answer **12**. Importantly, we simply refer to the information we need, ***without worrying about its location in the structure.*** LISP then returns the result.

We have completed a brief introduction to LISP. Common LISP has over 300 built-in functions, so clearly we have only scratched the surface. The concepts discussed reflect both the similarities and the differences between Prolog and LISP. The next section contrasts Prolog and LISP in more detail, and in particular, from an AI standpoint.

9.2 COMPARING PROLOG AND LISP

In the AI realm, nothing has seems to ruffle feathers and invoke battle lines like the debate over Prolog versus LISP. Sometimes, the debate even borders on hostility. Each side scoffs at the other, boastfully (or is that comically?) exclaiming, "We'll show you who is right!" This debate has been further exasperated by its international nature-- Europe and Japan generally favor Prolog, and the U.S. favors LISP. Recently, the debate has subsided somewhat as both sides have realized that each language has its own strengths and limitations.

We want to begin our comparison of Prolog and LISP by quoting the thoughtful statements of Patrick Winston, Director of the Artificial Intelligence Laboratory at the Massachusetts Institute of Technology in his Foreword to the Prolog text by Bratko (1990, p. vii):

> *In the Middle Ages, knowledge of Latin and Greek was essential for all scholars. The one-language scholar was necessarily a handicapped scholar who lacked the perception that comes from seeing the world from two points of view. Similarly, today's practitioner of Artificial Intelligence is handicapped unless thoroughly familiar with both LISP and Prolog, for knowledge of the two principal languages of Artificial Intelligence is essential for a broad point of view.*

A. Understanding Language Development

To properly compare Prolog and LISP, we need to understand *why* the two languages were developed. Computer scientists experiment with new theories of computing by developing new languages. They then test and adjust these prototype languages. As new ideas develop, new features are added to the languages.

Typically, in the early stages of language implementation, researchers emphasize new ideas over known ideas. In fact, the language may intentionally contain new ideas and exclude everything else. If these new ideas do not achieve much, the language stagnates and dies from lack of interest. If the achievements are sufficient and people show interest in the language, it gets support and tends to grow. The language makes a transition; it leaves the "initial" stage of development and enters the "intermediate" stage.

In the intermediate stage of development, many researchers experiment with a host of additions, adjustments, and changes to the language. Frequently, the language has no standard form, and many incompatible versions exist. Each version is closely related (since they have the same origin), but a program developed with one version typically cannot operate on a computer run by a different version.

Here again, in the intermediate stage, the language can stagnate or continue to grow. Growth occurs if 1) the language is sufficiently "proven" to be able to do its job, and 2) some form of commercial interest in the language exists. If commercial interest is shown, the language leaves the intermediate stage and enters the "commercial" stage. At this point, the language is usually fairly well-developed and on the road to maturity and standardization. After the language has been standardized and few additions are being made to it, it is called "mature."

Well-known languages such as FORTRAN, Pascal, C, and LISP are "mature" languages. Prolog, though, is in the intermediate stage of development, and therefore is still a developing language.

Each computer language has its own particular purpose. FORTRAN and C, for instance, are designed for numerical processing and arithmetic. They achieve their goals accurately and efficiently, and consequently, are well-known and accepted. LISP is primarily designed for *procedural symbolic processing*. As LISP developed, researchers added the ability to do arithmetic and logic programming.

Prolog is designed for *logic programming*. Logic programming is an attempt to model predicate logic, which is a formal way to deduce truth about objects and their relations. Logic programming uses pattern matching and has rules to infer truth, to reason and to draw conclusions, and model actual predicate logic. If we know how to program in Prolog, we know how to use predicate logic. For example, if we want to say in predicate logic, "for all X, if X is a hydrocarbon, this implies that X is flammable," we write:

$$\forall X \; \mathbf{hydrocarbon(X)} \rightarrow \mathbf{flammable(X)}$$

The ∀ symbol means "for all." In Prolog, an equivalent statement is:

```
    flammable(X) :- hydrocarbon(X).
```

To stress Prolog's logical inference capabilities, some versions of Prolog use the syntax ← instead of :-, as seen below:

```
    flammable(X)  ← hydrocarbon(X).
```

Thus, the main goal behind Prolog's original development was to model the predicate logic.

B. Advantages and Limitations of Prolog and LISP

Which language is better, LISP or Prolog? We cannot answer this question directly. Instead, we have to ask, "Better for what?" Prolog was developed for logic programming, and since it is still in the intermediate stage, it is a specialty language. LISP, in contrast, was developed for procedural symbolic computation. It is a mature, general-purpose language. Each has its strengths and limitations

1. Simplicity and Ease of Use

One advantage Prolog has over LISP is that it is easier to use and learn. A new programmer can learn about AI faster and easier by beginning with Prolog. LISP fans counter that how one learns a language does not matter much. Instead, *what you can do* with the language matters most. The LISP fans say that learning to operate a row boat is easier than learning to operate a tanker. But if we are going to transport oil across the ocean, we better use the tanker. LISP has a wider scope, and can do more than Prolog.

Prolog proponents acknowledge that LISP has a wider scope. They note, however, that the row boat and tanker analogy is not entirely correct. Prolog is amply powerful enough for AI programming. In addition, the wider scope of LISP is not without cost. First, Prolog fans claim that LISP is simply too complex an environment to program in efficiently. The very successful language C has approximately 30 reserved words. Prolog has about 50. LISP, on the other hand, has *over 300*. Because LISP has so many built-in functions, Prolog proponents argue that LISP lacks a clear theme, and most of the built-in functions will not be used anyway.

2. Efficient Hardware Utilization

To program in LISP, we need a special editor and dedicated hardware. Companies sell "LISP machines," i.e., computers dedicated exclusively to LISP. Prolog fans argue that this dedication is an expensive and inefficient use of hardware. Prolog can function with almost any editor on any system. It can also share hardware with other systems, making it more cost-effective. For example, a programmer can use Prolog along with other software on an inexpensive personal computer. In contrast, LISP requires dedicated, specialized hardware and a minimal $10,000 investment. In addition, once a LISP program is developed, its user group also needs specialized hardware, which is another $10,000 per user. The lack of accessibility of LISP systems to a wide range of users is, in our view, a serious drawback.

Prolog fans agree that LISP is more powerful, but claim that associated cost of that scope is simply too high. Most programmers do not come near to utilizing over half of the functions built-in to LISP. If these built-in functions are not used, then why pay for all that fixed cost? Why pay for a complex system that: 1) places it out of reach for most users, and 2) is expensive to use and operate, when that complexity is not utilized? Is it not more economical to develop a Prolog library of relations that we know we will use?

At this point in time, Prolog has a distinct advantage over LISP in hardware utilization. However, that gap may be closing. As personal computers get more powerful, the day may come when LISP will be accessible on a standard personal computer as well. Unfortunately, at the time of this writing, that day is not here yet.

3. Language Flexibility and Power

LISP fans argue that LISP is simply more flexible and more powerful than Prolog. We can write a Prolog program in LISP, but we cannot necessarily write a LISP program in Prolog. Prolog limits us (for the most part) to predicate logic. LISP can use predicate logic, plus a whole lot more. In LISP, we can implement records; in Prolog, we cannot.

However, Prolog is not yet a mature language; it is still undergoing changes. LISP proponents, on the other hand, note that many of the recommended changes do nothing but make Prolog more like LISP. For instance, some researchers have recommended adding a record feature to Prolog. LISP already has this feature through the built-in function **DEFSTRUCT**.

Prolog fans counter that, again, although LISP is more flexible than Prolog, Prolog is amply powerful enough for AI programming. Moreover, a compiled Prolog program will typically run faster than the same program compiled in LISP. In addition, Prolog has a built-in search and unification (matching) mechanism, while LISP does not. Therefore, Prolog programs typically "get off the ground" sooner than LISP programs, and can enter the prototype stage sooner and with less time invested.

4. Conclusions

Prolog and LISP both have their place in the AI world. We cannot make a blanket statement about which is better. LISP is certainly more flexible and powerful, and as microcomputer hardware improvements develop, LISP may be able to operate on a standard personal computer. The majority of expert systems are written in LISP, and it is the primary language of AI in the U.S. today. Prolog is amply powerful for expert-system implementation. It is easier to use, and generally, we can get a prototype expert system to run sooner in Prolog than we can in LISP. The initial investment of both time and money is lower with Prolog.

The decision about whether to use Prolog or LISP for an AI application may depend on the application itself. If most of the system can be represented using rules, and is more relationship-driven than data-driven, Prolog is the wiser choice. If the situation requires a more complex technique such as default reasoning, extensive and complex frame-based knowledge representation (see sections 10.2F and 11.2C), or object-oriented programming (see sections 10.2G and 11.2D), then LISP is probably the better choice. LISP is also preferred if the problem is more data-driven, and requires a technique known as "forward chaining" (see sections 10.2D and 11.2A).

Commercially, the vast majority of expert systems implemented use fairly simple knowledge-structuring techniques. Consequently, many of the functions included in a LISP system go unused, thereby negatively impacting LISP's price-to-performance ratio. For most rule-based and frame-based expert systems implemented commercially today, we feel that Prolog is the better choice for the following reasons:

(1) Initial investment in both time and money is much lower;

(2) Prototypes can be brought to the operational stage sooner;

(3) Prolog is accessible to personal computer users; and

(4) The complexity of commercial expert systems does not warrant nor require all the built-in LISP functions.

9.3 CHAPTER SUMMARY

- The two main languages used in AI today are Prolog and LISP.

- LISP is a well-developed, mature language. Prolog is still in the intermediate, or developing, stage.

- Prolog was originally developed to do logic programming, and is more of a specialty language. LISP was developed for general-purpose, procedural symbolic processing. LISP has a broader scope than Prolog.

- The question, "Which is better, Prolog or LISP ?" is not appropriate to ask. Each language was developed to meet specific goals, and each has its own strengths and weaknesses.

- The choice between Prolog and LISP depends on the AI application under consideration. Factors include what hardware restrictions exist, whether the problem is data-driven or relationship-driven, and whether the knowledge is suitable for the representation to be for the most part, rule-based or frame-based.

- For most rule- and frame-based expert systems utilized commercially today, the author believes that Prolog is the wiser choice. Compared to LISP, Prolog systems are: 1) less expensive, 2) accessible to personal computer users, and 3) have a superior performance/cost ratio that LISP.

A LOOK AHEAD

With the completion of this chapter, we have are familiar with the two most common languages used in AI: Prolog and LISP. Each language has its strengths and limitations and, depending on the problem we are trying to solve, we may wish to favor one language over the other. For overall cost/performance, however, we believe that Prolog is the wiser choice.

We now move from Prolog and LISP to a general discussion of artificial intelligence. In the next chapter, we answer questions such as, "What is AI?" and "What is knowledge representation?". With a firm grasp of both Prolog and AI concepts, we will be able to do productive AI applications using Prolog.

REFERENCES

Two recommended books on LISP programming are Winston and Horn (1986), and Friedman and Fellesin (1986). A good overview book on AI for engineers is Taylor (1988).

Batt, R., "Fifth-Generation Threat by Japanese Overrated, Industry Expert

Warns," *ComputerWorld*, 19 September (1983).

Brooks, R.A., *Programming in Common LISP*, John Wiley & Sons, New York, NY (1985).

Bylinsky, G., "Where the U.S. Stands: Computers, Chips, and Factory Automation", *Fortune*, 13 October (1986).

Davis, N.W., "U.S., Japan Compared in Computers, Research", *Japan Times*, 29 June (1985).

Feigenbaum, E.A., and P. McCorduck, *The Fifth Generation*, Addison-Wesley, Reading, MA (1983).

Friedman, D.P., and M. Fellesin, *The Little LISPer*, second edition, Science Research Associates, Inc., Chicago, IL (1986).

Mueller, R.A. and R.L. Page, *Symbolic Computing with LISP and Prolog*, John Wiley & Sons, New York, NY (1988).

Taylor, W.H., *What Every Engineer Should Know About AI*, MIT Press, Cambridge, MA (1988).

Unger, J.M., *The Fifth Generation Fallacy*, Oxford University Press, New York, NY (1987).

Winston, P.H., and B.K.P. Horn, *LISP*, second edition, Addison-Wesley, Reading, MA (1986).

PART TWO

INTRODUCTION TO ARTIFICIAL INTELLIGENCE

10

INTRODUCTION TO ARTIFICIAL INTELLIGENCE

This chapter provides an introduction to artificial intelligence (AI). We first introduce the broad field of AI and the specific subset called *expert systems*. We provide a historical perspective of AI, and discuss applications and challenges in AI. We then discuss formal ways of organizing AI systems and describe various types of knowledge representation in expert systems. At the end of this chapter the reader should have a broad view of AI, particularly of expert systems and of techniques of knowledge representation.

10.1 INTRODUCTION TO ARTIFICIAL INTELLIGENCE (AI)

A. Description of AI

1. Definitions

What is artificial intelligence? This question has been debated over the last several years, and no formal definition exists. The goal of AI has always been to make computers "think," to solve problems requiring human intelligence. Based on this idea, Rich and Knight (1991, p.3) define AI as:

> *The study of how to make computers do things which, at the moment, people do better.*

A. Barr and E. Feigenbaum (1981, Vol. I, p.3) have proposed the following definition:

> *Artificial Intelligence is the part of computer science concerned with designing intelligent computer systems, that is, systems that exhibit characteristics we associate with intelligence in human behavior.*

These two definitions focus on the goals of AI. B. Buchanan and E. Shortliffe (1983) offer a definition that focuses on the means of achieving the goal:

> *Artificial Intelligence is that branch of computer science dealing with symbolic, non-algorithmic methods of problem-solving.*

2. Prolog as an AI Language

Buchanan and Shortliffe's definition is useful if we want to understand how Prolog fits into the AI field.

Prolog is used in AI because it is does *symbolic* processing (section 2.1A). It utilizes:

- pattern matching (matching clause with clause, functor with functor, data object with data object, etc.);
- relations between objects; and
- qualitative and logical approaches

to solve problems. Prolog is in stark contrast to a procedural language such as FORTRAN that relies on numerical representation of information.

In addition, Prolog is used in AI because it involves *non-algorithmic* methods of problem-solving. Table 10.1 contrasts algorithmic and non-algorithmic methods.

Table 10.1. Algorithmic and non-algorithmic processing.

DESCRIPTION	TYPE OF PROCESSING	
	ALGORITHMIC	**NON-ALGORITHMIC**
Languages	FORTRAN	Prolog LISP
Procedure	step-by-step pre-defined pathway	dynamic procedure no pre-defined pathway
Input	well-defined fixed	can be incomplete flexible
Results	guaranteed fixed stopping point	not guaranteed unknown stopping point
Consistency	same input → same output	same input → different output

To solve AI problems with Prolog, we do not use an algorithm. Instead, we use a *search*. The search in Prolog is the pattern-matching procedure used to solve problems or achieve goals. Prolog scans from the top of the database, attempting to match the goal with an object in the database. Prolog matches clauses with clauses, functors with functors, lists with lists, and objects with objects in its attempt to satisfy the goal. By utilizing this built-in pattern-matching procedure, we are able to use Prolog to emulate intelligent behavior.

Almost all AI programs use *rules* and *heuristics*. Heuristics are general rules of thumb that apply in specific situations. As Prolog searches, it develops an *inference chain* that eventually leads to the solution. The inference chain is the sequence of rule-applications used to reach a conclusion.

Figure 10.1 presents a simple comparison between AI and conventional programming.

ARTIFICIAL INTELLIGENCE = SYMBOLS + SEARCH

CONVENTIONAL PROGRAMMING = NUMBERS + ALGORITHM

Figure 10.1. Artificial intelligence vs. conventional programming.

Using the tools of symbols and search, we can better model intelligence, and thus solve more complex problems. One drawback to AI, though, is that it does not guarantee a solution to the problem. However, human intelligence does not guarantee results either. When our car breaks down, there is no guarantee that a skilled and intelligent mechanic can solve the problem.

We now have an operational definition of AI, that is, *AI = symbols + search*, that we will use throughout the rest of the chapter.

B. A Brief History of AI

1. The 1960's

Most engineers are familiar with FORTRAN, one of the oldest computer languages. LISP, a symbolic computer language used today in AI applications, was introduced at MIT in 1959-60, only a few years after the introduction of FORTRAN. Therefore, the conception of "conventional" computing with FORTRAN and of "symbolic" computing (i.e., artificial intelligence) with LISP are very close to each other in history.

Compared to AI, conventional computing experienced a much more rapid growth in the 1960's, primarily because of the tremendous value of mathematical modeling. Computer systems based on FORTRAN were introduced into the market and were highly successful financially. Thus, conventional computing and programming became the standard.

During this period, however, research continued in AI. Researchers found ways to add to the computing power of AI systems, and confidence began to grow. All the while, AI was a "sleepy" academic discipline that showed little immediate commercial promise.

AI has its roots in trying to develop a "thinking machine" that emulates human intelligence. Therefore, the early stages of AI saw as much participation from neurologists, psychologists, and cognition experts as from the mathematicians and computer scientists.

In the late 1960's, the attitude toward computing could be described as: "Computers are fast, and they will continue to get faster. All we need to do is give them the ability to reason, and let them do all the work." There was tremendous optimism in computing, and little focus on its limitations.

Based on both the attitude in computing as well as the emphasis on human behavior and learning, AI research efforts in the 1960's focused on general problem-solving methodologies grounded in human cognition.

Computer speeds did increase as the AI professionals had hoped, but most AI systems proposed and prototyped in academia were impractical for commercial implementation. By the close of the decade, it became clear that more was needed if AI was going to be practical and commercially feasible.

2. The 1970's

The 1970's saw a change in approach from the 1960's. Emphasis on human cognition decreased as computer scientists and mathematicians began to apply more theoretical and rigorous approaches supported by mathematical proofs. AI became firmly grounded in mathematical logic.

In addition, the 1970's saw many computer scientists investigating principles and strategies of *search* for AI problem-solving methodologies. Techniques such as *the depth-first search*, *breadth-first search*, and *best-first search* (to be discussed in Chapter 11) were introduced.

In American universities, the 1970's saw single academic departments of "Mathematics and Computer Science" split into two departments: "The Department of Mathematics" and "The Department of Computer Science." As the computer-science discipline began to change and take on a character of its own, so did AI. Indeed, since computer science focuses on developing ways to make computers do things, AI became a clear computer-science discipline.

The early 1970's still saw tremendous potential in increasing computer speed. The AI mentality remained the same as that in the 1960's, with a slight change: "Computers are fast, and they will continue to get faster. All we need to do is tell them how to search, define some rules for them to follow, and we will get superhuman results."

A major emphasis of the 1970's was systematic problem-solving and search to enable AI programs to deal with practical problems. The result of this effort helped boost AI's applicability. Using these techniques, the power of these AI programs rose rapidly. However, they were still not quite powerful enough, and as the 1970's came to a close, it was clear that something more was needed.

3. The 1980's

The 1980's saw accelerating changes in the field of AI. Many of these changes were "inside-out", i.e., improvement in AI techniques, coming from *within* the AI community, to make AI more powerful and useful. More changes, however, were "outside-in", i.e., improvements in hardware, as well as other factors *outside* of the AI community, that brought previously impossible AI computing tasks within reach.

Let us first look at the "inside-out" forces of change. It became evident in the AI community that the reliance on mathematical logic, search and inference techniques coupled with increasing computer speed would be insufficient to solve practical problems. In the early 1980's, AI was mature enough that researchers could take a critical look at historical developments. This led to the conclusion:

General reasoning strategies are insufficient, and contextual reasoning is essential.

The 1960's and 1970's were characterized by developing techniques of general reasoning, i.e., inference that was independent of the problem being solved. GPS (General Problem Solver, Newell and Simon, 1963, Ernst and Newell, 1969) is a typical example of this era. GPS focused on the logical mechanisms required for general problem-solving, independent of the specific characteristics of the problem that we are trying to solve. GPS views problem-solving as an attempt to get from the current state to the goal state. The premise behind GPS is that logical operators and techniques exist (or could be built) that will get us from where we are to where we want to be. Importantly, these constructs are completely logical and consistent, and exist *independent* of problem context.

While GPS had some value and benefit, it became clear that context-independent reasoning had limitations. Further, it was realized that it was *impossible to get from the desired state to the goal state without problem-specific, contextual information.* General reasoning strategies were insufficient. High-quality, problem-specific, contextual reasoning was required.

This conclusion led to one of the biggest changes in the 1980's, the emphasis on *knowledge*.

Specific knowledge is essential if computers are to emulate intelligent behavior.

What is knowledge, and why was it emphasized? In the AI realm, we say that knowledge is the accumulation of facts, rules, and heuristics required to reason through, and solve a program:

KNOWLEDGE = FACTS + RULES + HEURISTICS

Importantly, knowledge can be quantitative or qualitative. We have seen this with Prolog facts. Examples of quantitative and qualitative knowledge are:

Quantitative: **normal_boiling_point(water,100).**

Qualitative: **corrosive(hydrochloric_acid).**

Few AI systems emerged from the 1960's and 1970's as being truly successful. It was noticed that these systems relied heavily on problem-specific reasoning and knowledge. The successful implementation of these AI systems led to the conclusion that knowledge was paramount. The first system, MACSYMA, was developed in the late 1960's at MIT. MACSYMA performs mathematical manipulations, including symbolic differential and integral calculus, algebra, simplification, limits, and solutions of equations. Another system, DENDRAL (DENDRitic ALgorithm), was developed at Stanford in the 1970's (Buchanan and Feigenbaum, 1978). DENDRAL infers molecular structure from mass-spectroscopy and nuclear magnetic-resonance data. Both MACSYMA and DENDRAL can consistently perform better than or equal to humans in their specific area of expertise. MACSYMA and DENDRAL demonstrated the power of high-quality, problem-specific knowledge.

Figure 10.2 illustrates AI trends from 1960-1980.

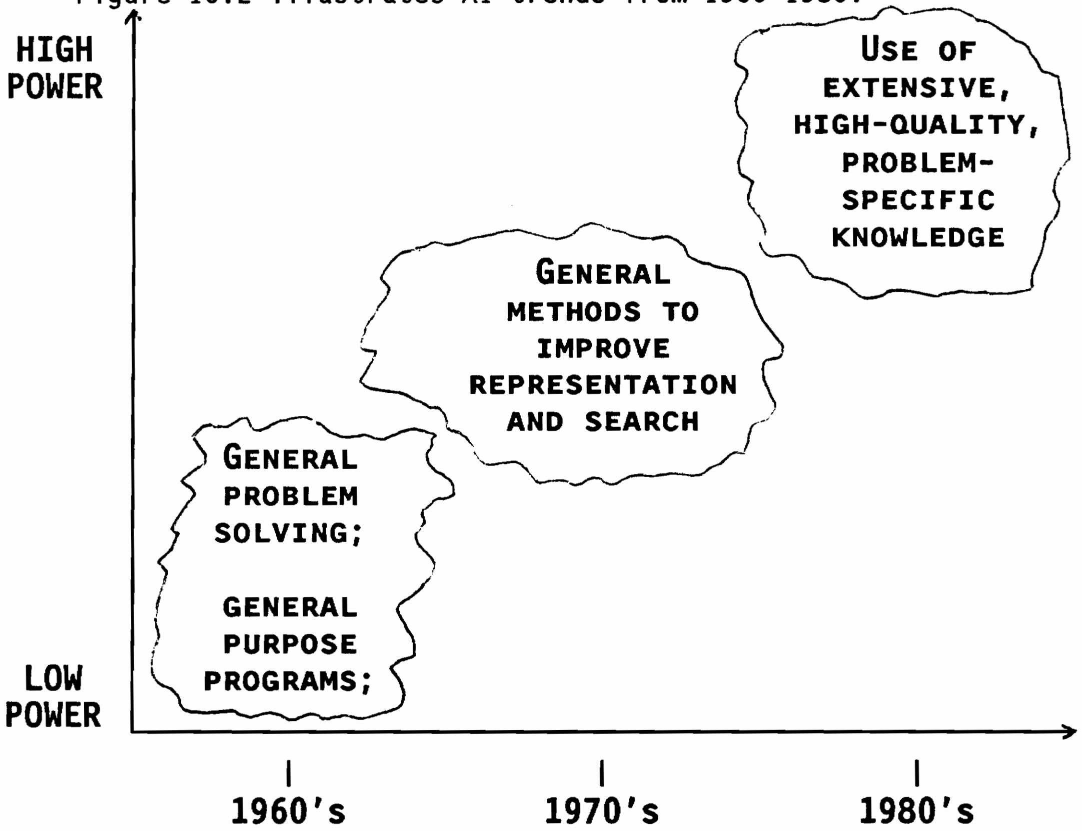

HIGH POWER
LOW POWER
GENERAL PROBLEM SOLVING;
GENERAL PURPOSE PROGRAMS;
GENERAL METHODS TO IMPROVE REPRESENTATION AND SEARCH
USE OF EXTENSIVE, HIGH-QUALITY, PROBLEM-SPECIFIC KNOWLEDGE
1960's
1970's
1980's

Figure 10.2. AI trends in the 1960's-1980's.

Now, let us focus on AI trends in the 1980's by looking at "outside-in" effects. Some of the major trends are described below.

3.1 Hardware Improvements Make AI Commercially Feasible

The rapid growth of AI in the 1980's was not due to any revolutionary software breakthroughs within the AI community. Instead, it was due to the widespread availability of low-priced, powerful hardware. Microcomputers were introduced; these microcomputers got smaller and smaller, yet more and more powerful. Prices fell, and computers became accessible to a wide range of users. This accessibility opened up AI to a much wider range of scientists and engineers, and fueled AI's growth in the 1980's.

3.2 Other Disciplines Begin to Apply AI Techniques

From dairy science to business and finance, chemical engineering, and even music and the arts, other disciplines applied AI techniques. These applications, coupled with and fueled by the availability of low-priced hardware, brought AI into the commercial realm. According to George Stephanopoulos (1990), this interaction with other disciplines proved extremely useful for the following reasons:

(1) AI researchers left behind toy problems (e.g., puzzles, checkers, chess) and addressed significant, real-world scientific and engineering problems with tangible value.

(2) Scientists and engineers in various disciplines possess the amount of specific knowledge that computers need to exhibit advanced problem-solving skills. Thus, as the need for problem-specific knowledge became central in AI research, it was essential to have not simple overtures, but strong interaction with other disciplines.

(3) The solution of specific scientific and engineering problems

using AI techniques started generating a very positive impact, which tremendously helped AI's posture and subsequent rebirth.

3.3 Media Attention and Overselling Cause Problems

The 1980's saw a lot of media attention focused on AI. Some touted it as the breakthrough that would revolutionize computing throughout the world. Nations and governments announced and began large-budget, national research programs in AI. What developed was a sort of "AI bandwagon" that many people joined.

With all the media attention focused on AI, small, start-up AI companies were formed to address this rapidly growing market. They sold AI hardware and software to industry. Unfortunately, most of these companies charged high prices and oversold their products' capabilities. While this tactic may have been good for profits in the short term, it was not good for the long run. As can happen in these types of markets, many of the machines sold had an unfavorable price-to-performance ratio.

Unfortunately, academic researchers applying AI to the scientific and engineering disciplines were partly to blame for the overselling. In research publications, they stressed the good points of their work, but seldom addressed the drawbacks, limitations, and potential problems. These publications, in turn, created unrealistic expectations in industry.

Many companies purchased expensive "LISP machines" in the early and mid-1980's, only to find that they were expensive, high-overhead systems that were difficult to integrate with other hardware and were inaccessible to the average user. The computing breakthrough promised by the small AI enterprises did not materialize. Instead, companies that purchased these machines found themselves in possession of expensive hardware and software that were being under-utilized. The market clearly showed that the price-to-performance ratio for these systems was insufficient. With the "hype" now gone, criticism and an "AI backlash" began to erupt.

4. Where Are We Today?

Many of the small, entrepreneurial AI companies felt the AI backlash. It was clear that the price-to-performance ratio on some products was not good enough. Thus, in 1990-91, we see these AI companies either consolidating with other companies or just plain going out of business.

Realism has set in, and both the benefits *and limitations* of AI techniques are better recognized. In particular, companies selling AI systems are improving the price-to-performance ratio; we have seen the cost of implementing an expert system decrease substantially over the past five years. In academic circles, researchers are working on ways to improve AI approaches.

Thus, as the 1990's begin, there is cautious optimism in the AI community. Efforts are focused on overcoming some of the limitations of AI. Probably the biggest trend that we see developing in the 1990's is the effort to mesh both quantitative information (e.g., mathematical modeling) and qualitative reasoning (e.g., expert systems) into a single, integrated system.

C. Uses of AI

Artificial Intelligence has a number of applications. After giving a brief overview, we focus in section 10.2 on expert systems in science and engineering.

Uses of AI include:

- *Natural Language Processing-* Called "natural language" for short, the goal of this area is to understand and comprehend spoken and/or written language. Knowledge of grammar and vocabulary is, of course, essential. But an even bigger challenge here is understanding the many hidden assumptions in spoken language.

  Natural-language processing has been used as an intelligent computer-interface, and provides an easy-to-use "front end" to complex computer programs. Another application of natural language has been automatic translation between languages.

- *Computer Vision-* The goal of this area is to develop visual interpretive skills equivalent to those of humans. Humans take their visual skills for granted, but visual interpretation requires intelligence. Discerning 1) partially obstructed views, 2) shadows, and 3) objects from different viewpoints and angles are the challenges facing this area. Applications include machine vision, aerial-photograph interpretation, and computer-aided manufacturing.

- *Robotics*- Work in this area seeks to develop more flexible robotic systems. A robotic system would become much more useful if it could 1) perceive intelligently (visually and/or audibly), 2) assess hazards in its pathway, 3) react to a changing workplace layout, and 4) intelligently assess cause-and-effect sequences.

- *Theorem-Proving*- This area focuses on proving complex theorems that require an extensive amount of inference knowledge. The effort here, of course, is driven primarily by mathematicians.

- *Expert Systems*- An expert system is a man-machine system possessing in-depth, specialized knowledge about a specific area. Because of this focused knowledge, it can solve complex problems. An example is DENDRAL, an expert system developed at Stanford University. DENDRAL takes in mass-spectroscopic and nuclear magnetic-resonance data, and uses this information to infer the structure of an unknown chemical. Expert systems have found use in areas such as business, mathematics, chemistry, medicine, and engineering.

- *Learning*- The goal of this area of AI research is to develop system that can: 1) take in data in a specific area, 2) apply inferences to the data and any information in the system's knowledge base, and 3) use this information to do new things or adapt to new situations.

- *Artificial Neural Networks*- These are also called ANNs, and are a network of highly interconnected nodes that can map a complex input pattern with a complex output pattern. ANNs use numerical rather than symbolic processing. Each node has multiple inputs, and once the input is above a specific level, the node generates a single output. This output, in turn, can be fed to any node in the network. What results is a highly interconnected set of nodes that are capable of learning, control, and adaptive behavior. We shall discuss ANNs in Chapter 17.

The applications above are by no means an exhaustive list. New applications are growing daily, especially for expert systems, which have jumped from the realm of AI research to wide-scale commercial use.

C. Challenges in AI

By incorporating symbols and search, AI systems have the ability to solve complex problems. There are some major challenges in AI, though. One of the biggest lessons learned through research in the last twenty years is that *intelligence requires knowledge*. The corollary to this lesson is that *AI needs more than just faster computers*.

When AI was first developed in the 1960's and 1970's, the emphasis was reliance on computational speed. Although computer speed has increased, many AI systems did not deliver, and in general, the results did not always materialize. Since then, we have learned that an effective AI system requires not only a fast computer, but properly represented knowledge.

Knowledge acquisition, representation, and utilization are some of the biggest challenges facing AI today. What is so difficult about knowledge? Rich and Knight (1991, p.8) gives some challenging properties involved with knowledge:

- It is voluminous.
- It is hard to characterize accurately.
- It is constantly changing.
- It differs from data by being organized in a way that corresponds to the ways it will be used.

These four points are unlikely to change. They are inherent, undesirable characteristics that must be overcome to effectively utilize AI.

As the 1990's are now here, we wish to add another challenge to the AI area:

Most scientific and engineering-based AI systems require both quantitative and qualitative analysis.

The growth of these "integrated systems" that combine quantitative and qualitative reasoning will aid the AI effort all the more.

E. Exercises

10.1.1 Contrast algorithmic and non-algorithmic processing. How are they different?

10.1.2 Shown below are some chemical engineering problems that are suitable for computer-aided solutions. Answer whether we should use artificial intelligence or a conventional programming to solve each problem.

a. Assessing yield and quality trends from a chemical reactor, for the purpose of recommending shutdown for a catalyst change.

b. Determining the fluid velocity in a fluid-flow problem.

c. Making a robot correct its arm position in a welding operation before the weld quality begins to suffer.

d. Developing a chemical process flowsheet.

e. Designing a distillation column.

f. Qualitatively modeling a chemical process.

g. Monitoring a chemical process and assessing the origins of process disturbances.

h. Modeling the performance of a plug-flow reactor.

10.2 INTRODUCTION TO EXPERT SYSTEMS

A. Description of Expert Systems

Expert systems are one of the fastest growing applications of AI in the scientific and engineering fields. Expert systems attempt to match the performance of human experts in a given field. To do so, these systems rely on in-depth, *expert knowledge*. The better the knowledge, the better is the performance of the system.

Knowledge is usually incorporated into expert systems through *relationships*. An expert system keeps track of relations and inferences invoked. Therefore, the knowledge used by the system is explicit and accessible to the user. An expert system can explain *why* certain information is needed, and *how* certain conclusions are reached.

Some advantages of expert systems are that they:

(1) can assimilate large amounts of knowledge; and

(2) never forget that knowledge.

These properties distinguish expert systems from conventional computer programs. Ideally, an expert system can build its *own* knowledge base, although achieving that goal has been very challenging. Another ideal is for field experts who are not programmers to expand the knowledge base (again, a major challenge). Expert systems use a combination of user interface and inference mechanisms, sometimes called the *expert system shell*, for the expansion of the knowledge. An ideal expert system, shown in Figure 10.3, contains:

(1) a knowledge base;

(2) an inference engine; and

(3) a user interface.

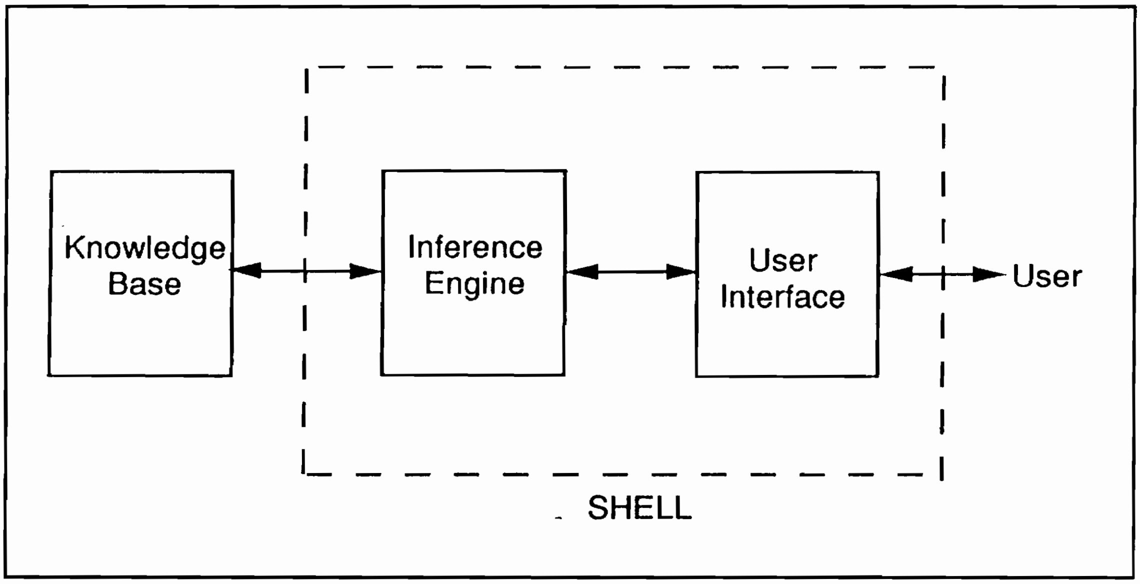


Figure 10.3. The structure of an expert system.

The *knowledge base* contains specific, in-depth information about the problem at hand. That knowledge consists of facts, rules, and heuristics, as shown in Figure 10.4.

KNOWLEDGE = FACTS + RULES + HEURISTICS

Figure 10.4. Knowledge in expert systems.

To utilize the knowledge (facts, rules and heuristics), an expert system relies on its *inference engine*. The inference engine uses inference mechanisms to process the knowledge and draw conclusions. The user interface provides smooth communication between the program and the user.

As stated previously, the inference engine and the user interface are combined under the term *expert system shell*, or simply the shell. Ideally, the shell:

(1) answers *how* a conclusion was reached;

(2) answers *why* certain information is needed; and

(3) has the ability to add knowledge to the knowledge base.

Note that in the absence of the knowledge base, the shell is just that-- an empty shell that can do nothing. But importantly, the shell has the ability to work with and update the knowledge base. Knowledge can be added 1) by a human expert who is not a programmer, or 2) by the expert system itself as a result of inference. The ability to perform these duties makes expert systems powerful, flexible "thinking machines."

B. Uses of Expert Systems in Science and Engineering

F. Hayes-Roth (1983, p. 14) has identified a number of applications of expert systems, summarized in Table 10.2.

Table 10.2. General expert-system applications.

CATEGORY	PROBLEM ADDRESSED
Interpretation	Inferring situation descriptions from sensor data
Prediction	Inferring likely consequences of given situations
Diagnosis	Inferring system malfunctions from observables
Design	Configuring objects under constraints
Planning	Designing actions
Monitoring	Comparing observations to plan vulnerabilities
Debugging	Prescribing remedies for malfunctions
Repair	Executing a plan to administer a prescribed remedy
Instruction	Diagnosing, debugging, and repairing student behavior
Control	Interpreting, predicting, repairing, and monitoring the system

Expert systems have gone commercial in several of these listed areas. Table 10.3 lists a number of scientific and engineering applications; expert systems have been applied to financial management and planning, marketing, and military operations as well.

Table 10.3. Applications of expert systems in the process industries.

- Process design
- Process simulation and optimization
- Plant-layout decision support
- Training
- Process-fault diagnosis
- Process control
- Mechanical and structural design
- Planning
- Start-up and shut-down analysis
- Critiquing a design for flexibility, reliability, and safety
- Monitoring and assessing the origins of process trends
- Automatic programming

As seen in Table 10.3, expert systems are in no way "futuristic." They are applied today in a wide range of areas in the process industries.

To indicate just how practical expert systems are, let us look at some actual implementations. Table 10.4 lists examples of several expert systems developed.

Table 10.4. Examples of some expert systems developed.

SYSTEM	DESCRIPTION
DENDRAL	Infers molecular structure from mass spectroscopy and nuclear magnetic resonance data (DENDRitic ALgorithm, developed at Stanford University).
MACSYMA	Performs mathematical manipulations, including symbolic differential and integral calculus, algebra, simplification, limits, and solutions of equations (developed under project MAC at MIT).
PROSPECTOR	Aids geologists in their search for ore deposits. The system takes in field data and estimates the likelihood of locating a specific type of deposit (developed by SRI International).
MYCIN	Diagnoses and recommends treatment for various infectious blood diseases. The system identifies the infecting organism based on symptoms, laboratory data, and patient history. It recommends drug type and dosage for treatment (developed at Stanford University).
FALCON	Identifies the causes of disturbances in a commercial chemical plant. From plant data, FALCON identifies probable sources of disturbances that affect plant performance (Fault AnaLysis CONsultant, developed as a joint project between the University of Delaware, E.I. duPont de Nemours, and Foxboro on duPont's Victoria, Texas adipic acid plant).
ISA	Aids in scheduling customer orders of computer equipment. The system inputs customer orders and develops a schedule based on the current material allocation. It also exposes difficulties and recommends alternatives (Intelligent Scheduling Assistant, developed by Digital Equipment Corporation, and used in their manufacturing facilities).

DENDRAL, MACSYMA, and MYCIN, in particular, perform consistently better than or equal to the best human experts in their fields. Interested readers may refer to Waterman (1986, pp. 203-235), for further descriptions and applications of some available expert systems. In addition, the *American Association for Artificial Intelligence* publishes *The AI Magazine*, a general news magazine in the field of AI. The AI Magazine is not overly technical; it is appropriate for almost any reader. For further information, write to: American Association for Artificial Intelligence, 445 Burgess Dr., Menlo Park, CA, 94025.

C. Representing Knowledge in Expert Systems

Now that we have a sense of what expert systems are, we can focus on how to represent knowledge (facts, rules, heuristics). Knowledge representation is critical to the success of expert systems. For instance, if we want to add two numbers together to get a result, we may represent it several ways:

Arabic numerals:	5 + 6 = 11
Roman numerals:	V + VI = XI
Boolean algebra:	101 + 110 = 1011

Clearly, the ease of solving this addition problem depends on which representation we use. This point is true throughout artificial intelligence; the ease of problem-solving depends highly on the AI knowledge representation.

AI researchers have developed a host of different knowledge representations. For a knowledge representation to be useful, it must be able to:

(1) correctly capture problem characteristics, relationships, and information into a single, unifying framework; and

(2) process this information is a systematic way to draw conclusions and solve problems.

We focus on two representations that have found the most use in expert system applications: *logic-based*, and *frame-based systems*. We also discuss another useful approach to representing knowledge, called *object-oriented programming*. All of these systems use, for the most part, a qualitative knowledge representation (although quantitative analysis can also be incorporated). Thus, they perform a qualitative or semi-quantitative analysis to draw conclusions and solve problems. They can also be used for *qualitative modeling*, a topic we shall address in chapter 16.

D. Logic-Based Systems

Logic-based systems use principles and constructs of mathematical logic to represent and process information. Many logic-based knowledge representations exist. The two most common logic-based representations used in science and engineering are *rule-based* and *fuzzy-logic* systems.

1. Rule-Based Systems

Using *rules* is one of the most common ways to represent knowledge in expert systems. We have talked extensively about rules in Prolog. Rules are conditionally true, and can be viewed as IF-THEN statements. For instance, the Prolog statement

```
flammable(X):-  petroleum_derivative(X).
```

says that X is flammable if X is a petroleum derivative.

Rule-based expert systems use pattern-matching. As an expert system consults its knowledge base to answer a question, it develops an *inference chain*. The inference chain is a sequence of steps or rule-applications used by the expert system to analyze and solve problems. In the above **flammable** statement, the inference chain says that we can infer that **X** is flammable if we can prove that **X** is a petroleum derivative.

1.1 Methods of implementation

We can implement an inference chain in rule-based expert system in one of two ways: *forward chaining* or *backward chaining*.

Forward Chaining

Forward chaining establishes the premises first, and then concludes that the rule is true. For instance, we could say that if there is:

(1) a flammable material (F);

(2) oxygen (O); and

(3) a source of fire (S),

then an extreme fire hazard may exist. To infer this conclusion using forward chaining, we write:

```
F,O,S --->  extreme fire hazard      % IF ---> THEN (Forward Chaining)
```

Forward chaining is *data-driven*. It relies on IF-THEN-type rules; that is, IF the data are in a certain configuration, THEN certain conclusions can be drawn. Forward chaining is a "bottom-up" inference mechanism that tries to *reduce* a number of premises into a single conclusion.

Backward Chaining

Backward chaining treats the conclusion of the rule as a goal and attempts to satisfy the goal by proving that the premises (sub-goals) are true. In the fire hazard example, a backward-chaining inference would be:

```
extreme fire hazard <---- F, O, S    % THEN <--- IF (Backward Chaining)
```

Backward chaining is *goal-driven*. It relies on THEN-IF-type rules; that is, THEN a certain goal is true, IF certain sub-goals are proven true. Backward chaining is a "top-down" inference mechanism that tries to *instantiate* variables through matching with facts. *Prolog uses backward chaining*.

Which is better, forward or backward chaining? The answer depends on the specific application. If an application is very data-dependent, with less need for inferences, we may choose forward chaining. If we have complex relationships that require in-depth inferences with less data, we may prefer backward chaining. Expert systems can use either forward or backward chaining.

One advantage backward chaining has over forward chaining is that backward chaining is *goal-oriented*. A forward-chaining system has a tendency to establish everything it can before stopping, and therefore tends to be "trigger happy." Without careful control, rules can "fire" uselessly. Consequently, forward-chaining systems tend to *wander* in search of the result if much inference is needed.

1.2 Advantages and Disadvantages of Rule-Based Systems

A rule-based expert system is the easiest and most common way to represent knowledge today. Table 10.5 summarizes some advantages and disadvantages.

To eliminate some of the disadvantages of rule-based systems, AI researchers have developed fuzzy-logic, truth maintenance, as well as frame-based systems. We discuss fuzzy-logic and frame-based systems in this chapter. We shall discuss truth-maintenance systems briefly in chapter 16.

Table 10.5. Advantages and disadvantages of rule-based expert systems.

Advantages

- Easily developed knowledge base
- Promotes a modular program
- Flexible
- Easily modified
- Flexible control
- Readable and understandable
- Suitable for dynamic, changing knowledge

Disadvantages

- Knowledge is not integrated into overall structure
- Fails to capture large blocks of interrelated knowledge
- Undesirable rule interactions
- Obscure control; undesirable rule firing
- Inefficient for large amounts of static knowledge
- Excessively focused; fails to grasp overall picture or trends. Cannot develop expectations.

We now move on to fuzzy-logic systems.

2. Fuzzy-Logic Systems

Fuzzy logic grew out of a desire to quantify rule-based systems. Rule-based reasoning is grounded in qualitative knowledge representation, and fuzzy logic allows us to mesh a quantitative approach with the qualitative representation. Fuzzy logic is used to quantify certain qualifiers such as *approximately*, *often*, *rarely*, *several*, *few*, and *very*.

Fuzzy logic is not a substitute for statistics. Indeed, fuzzy logic is used when statistical reasoning is inappropriate. Statistics is used to express the extent of knowledge (or lack thereof) about a value, and relies on tools such as variance, standard deviation, and confidence intervals. Fuzzy logic, on the other hand, is used to express the absence of a *sharp boundary* between sets of information. For example, we may write:

- Crude oil fractionation is an energy-intensive unit operation, 1.0.
- Thermal cracking is an energy-intensive unit operation, 0.9
- Catalytic reforming is an energy-intensive unit operation, 0.6.
- Catalytic cracking is an energy-intensive unit operation, 0.3.
- Open-air evaporation of brine to produce salt is an energy-intensive unit operation, 0.0.

We use fuzzy logic to delineate the lack of a sharp boundary between *clearly* energy-intensive (1.0) and *not at all* energy-intensive (0.0). Crude fractionation is very energy-intensive, while open-air evaporation of brine is not at all energy-intensive. Thermal cracking, catalytic reforming, and catalytic cracking cannot be considered either very energy-intensive or not at all energy-intensive. Thus, fuzzy logic is not used in a statistical sense to quantify lack of knowledge. Instead, fuzzy logic is used to quantify the *degree or extent* of certain words and boundaries between sets of information.

To use fuzzy logic, we first need a *fuzzy set*. In a fuzzy set, the transition from membership to non-membership is not well-defined. We quantify the *degree of membership* with values between 0 (not a member) and 1 (definitely a member). With our energy-intensive unit-operation example, the fuzzy set is:

{crude oil fractionation/1.0, thermal cracking/0.9, catalytic reforming/0.6, catalytic cracking/0.3}

The open-air evaporation of brine to produce salt has a degree of membership of 0.0, and therefore, is not a member of the set.

Once we construct fuzzy sets, we use *fuzzy reasoning*. Many are familiar with the concepts of *union* and *intersection* in classical set theory. We apply the union and intersection operations to fuzzy sets too. Let us define two fuzzy sets:

$$I = \{x_1/i_1 \ , \ x_2/i_2 \ , \ \ldots \ , \ x_n/i_n \}$$

$$J = \{x_1/j_1 \ , \ x_2/j_2 \ , \ \ldots \ , \ x_p/j_p \}$$

where x_1, x_2 ... are members of the set with nonzero degrees of membership i_1, i_2, ... (for set I) and j_1, j_2, ... (for set J). Note that the sets *do not* need to have the same number of members; set I has n members, and set J has p members.

The union of two fuzzy sets is the fuzzy set containing the members of each set with the maximum degree of membership of that element in either set:

$$I \cup J = \{x_1/(\max(i_1,j_1)) \ , \ x_2/(\max(i_2,j_2)) \ , \ \ldots\}$$

The intersection of two fuzzy sets is the fuzzy set containing the members of each set with the minimum degree of membership of that element in both sets:

$$I \cap J = \{x_1/(\min(i_1,j_1)) \ , \ x_2/(\min(i_2,j_2)) \ , \ \ldots\}$$

For example, we consider the two sets:

I = {crude oil fractionation/1.0, thermal cracking/0.9, catalytic reforming/0.6, catalytic cracking/0.3}

J = {crude oil fractionation/0.8, thermal cracking/0.75, catalytic reforming/0.7, catalytic cracking/0.2, polymerization/0.1}

We perform both union and intersection:

$I \cup J$ = {crude oil fractionation/1.0, thermal cracking/0.9, catalytic reforming/0.7, catalytic cracking/0.3, polymerization/0.1}

$I \cap J$ = {crude oil fractionation/0.8, thermal cracking/0.75, catalytic reforming/0.6, catalytic cracking/0.2}

Davis and Gandikota (1990) discuss fuzzy sets. We use their simple example here to demonstrate reasoning with fuzzy sets. If we have qualitative values for flow rate (F) and pressure (P) of a chemical process, we may write a rule that say the system is abnormal:

The system is abnormal if:

(1) Both F and P are high, OR

(2) F is low, OR P is low.

Let us make F a fuzzy set of flow rates, and P a fuzzy set of pressures, with the following degrees of membership:

F = { low_F/0.5, high_F/0.3, normal_F/0.2}
P = { low_P/0.8, high_P/0.15, normal_P/0.05}

Now let us determine the following:

(1) *Certainty of high_F and high_P*: determined by the intersection of fuzzy sets F and P. Thus, certainty is the minimum degree of membership of high_F and high_P: certainty = min(0.3,0.15) = 0.15.

(2) *Certainty of low_F or low_P*: determined by the union of fuzzy sets F and P. Thus, certainty is the maximum degree of membership of low_F and low_P: certainty = max(0.5,0.8) = 0.8.

(3) *Overall uncertainty*: determined by taking the maximum certainties of both results, i.e., certainty = max(0.15,0.8) = 0.8.

Note again that the certainty in these rules is *not* to be interpreted as some type of "confidence limit" in the conclusion drawn. Instead, the certainty represents confidence in the qualitative values of the flow rate and pressure in the fuzzy sets.

E. Semantic Networks

1. A Description of Semantic Networks

A semantic network (also called semantic net) is a graphical representation of knowledge, where related facts (called *nodes*) are interconnected with links (called *arcs*). Quillian (1968) introduces and discusses semantic networks. Relatively speaking, semantic networks are one of the older knowledge representations. A semantic network for a distillation column is shown in Figure 10.5.

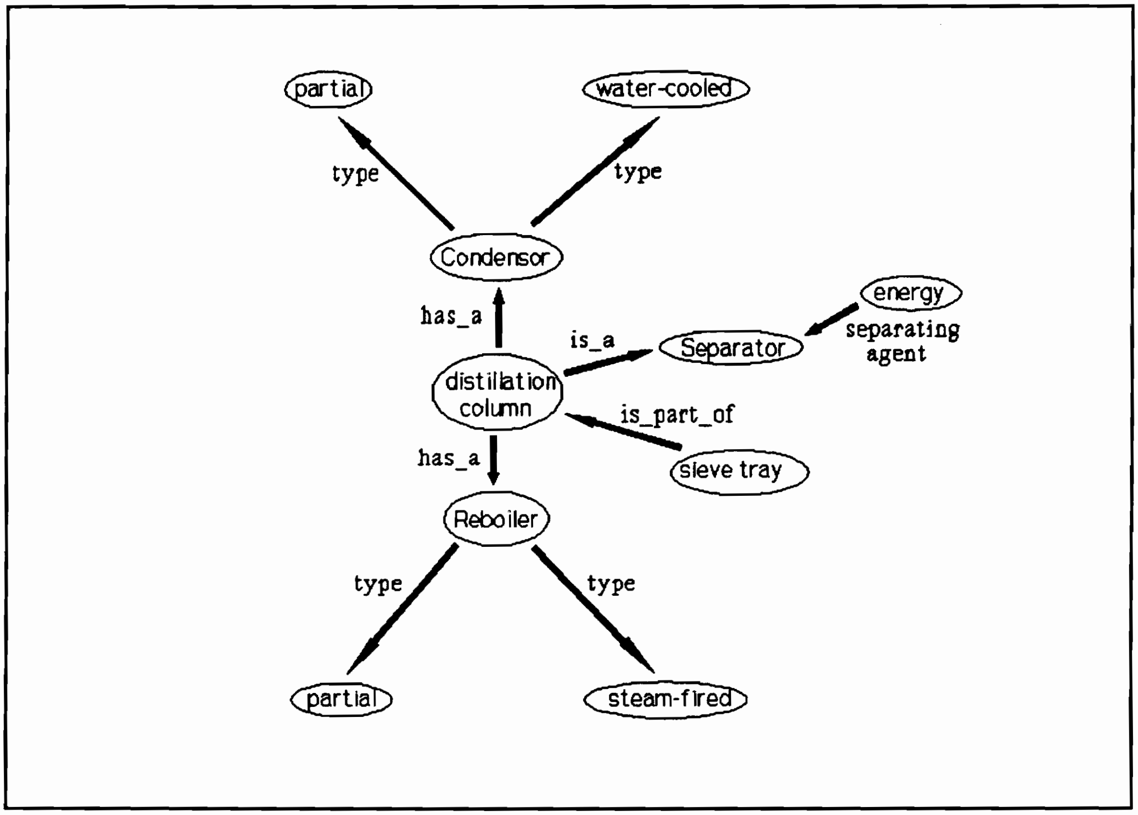


Figure 10.5. A semantic network of a distillation column.

The nodes in this semantic network are: *distillation column*, *reboiler*, *condenser*, *partial* (used twice), *water-cooled*, *steam-fired*, *sieve tray*, *separator*, and *energy*. The arcs linking the nodes represent relationships, and consequently, are labeled with qualifiers such as *is_a*, *has_a*, *is_part_of*, *type*, and *separating_agent*.

Semantic networks, originally designed for understanding language, can be used for a number of purposes. For the most part, they are used to properly characterize a problem and simplify the deduction process. They allow us "get started" by providing a graphical representation of relationships. Semantic networks can answer questions such as "What is the connection between a distillation column and a reboiler?"

Because semantic networks are graphical, we can easily follow the arrows to deduce relations. An important principle we can follow when deducing relations is that of *inheritance*. Nodes "upstream" of arrows inherit all properties expressed "downstream." For example, the reboiler is a partial reboiler and is steam-fired. Since the distillation column has a reboiler, the distillation column inherits those properties.

When implemented in a computer program, a semantic network, of course, cannot be used graphically. However, we can easily implement the semantic network as a set of facts in Prolog:

```
is_a(distillation_column,separator).
has_a(distillation_column,condenser).
has_a(distillation_column,reboiler).
is_a_part_of(sieve_tray,distillation_column).
separating_agent(energy,separator).
type(reboiler,partial).
type(reboiler,steam_fired).
type(condenser,partial).
type(condenser,water_cooled).
```

We see above that a semantic network gives us a *symbolically traceable* means of characterizing relationships.

2. Semantic Networks Versus Rule-Based Systems

Rule-based systems are ideal for problems that are amendable to an "if-then" knowledge representation. Most, but not all problems can be characterized in this fashion. Semantic networks have some unique properties that we can take advantage of to solve some specific problems. Particularly, semantic networks provide:

(1) a hierarchical method for organizing knowledge;

(2) a means of *inheritance* from upper hierarchies to lower hierarchies; and

(3) a means of organizing spatial relationships, e.g., the overhead vapor stream from a distillation column feeds into a condenser.

These problem characteristics *could* be implemented in a rule-based system. However, they fit much more naturally into the semantic network form of knowledge representation.

3. Uses of Semantic Networks in Chemical Engineering

Semantic networks can and have been used to achieve numerous objectives in process engineering. One of their primary uses is an organizing tool for expert systems. Semantic networks clearly identify and delineate relationships. They give us the means of coordinating and organizing our knowledge before actually writing an expert system.

An example of the use of semantic network is in the so-called *Fishbone Diagram*, also known as the *Ishikawa Diagram* or *Cause-and-Effect Diagram*. An example of a fishbone diagram is shown in Figure 10.6. This diagram identifies potential causes of a "break-out" of a roll of a textile-like synthetic fabric in the production of diapers. Causes are separated into the following categories: rewind, line tension, finishing rolls, unwind and turn, coating, cooling, and pull module.

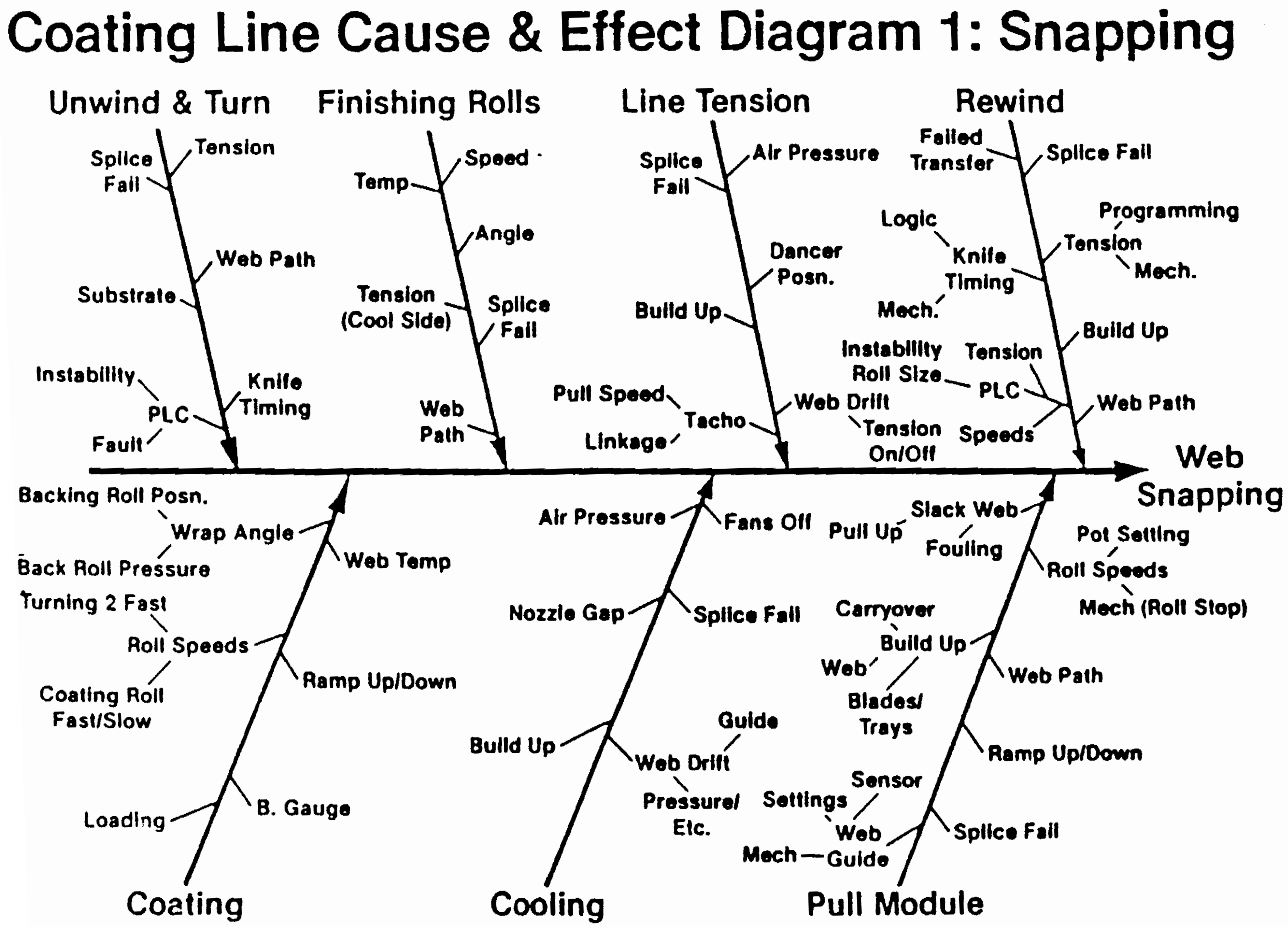


Figure 10.6. Fishbone diagram for a roll break-out.

]

Finally, semantic networks have been used to represent process and/or chemical reaction structures. An example of these is discussed in Ungar and Venkatasubramanian (1990). Figure 10.7 is the semantic network for the ammonia synthesis reaction. Figure 10.8 is the semantic network for the ammonia process structure.

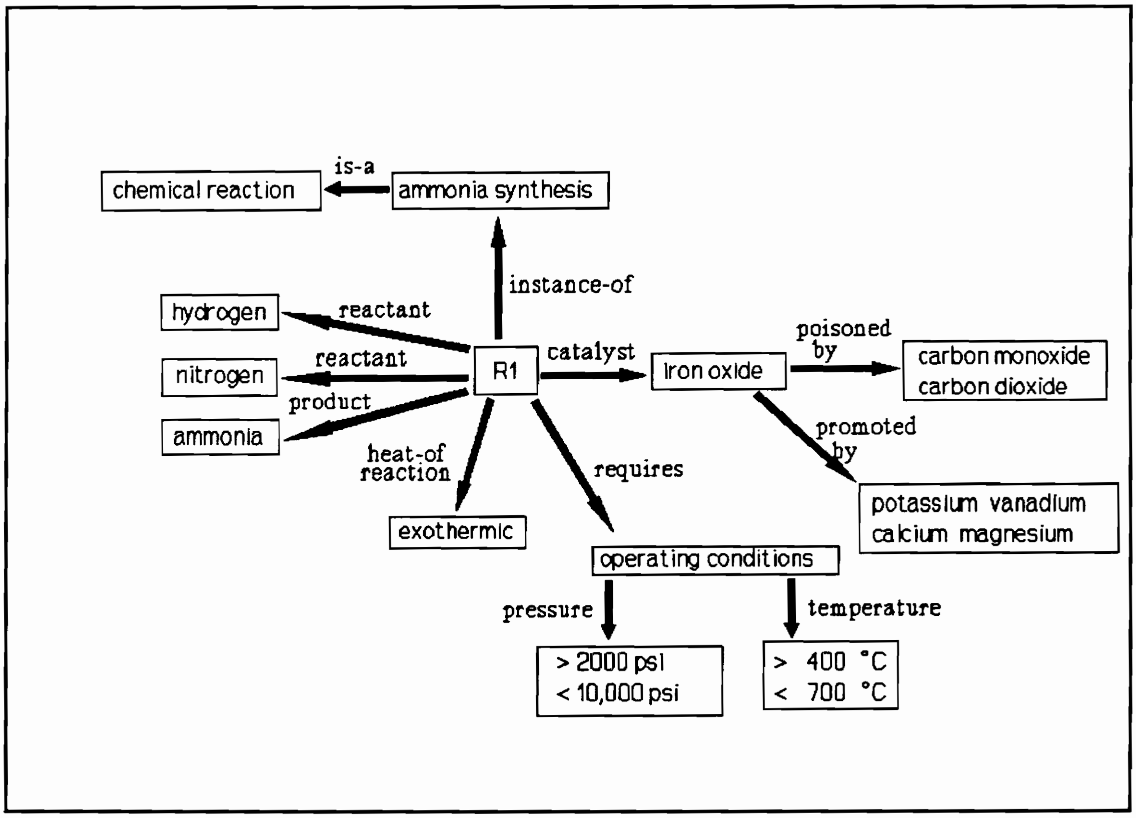


Figure 10.7. Semantic network of the ammonia synthesis reaction.

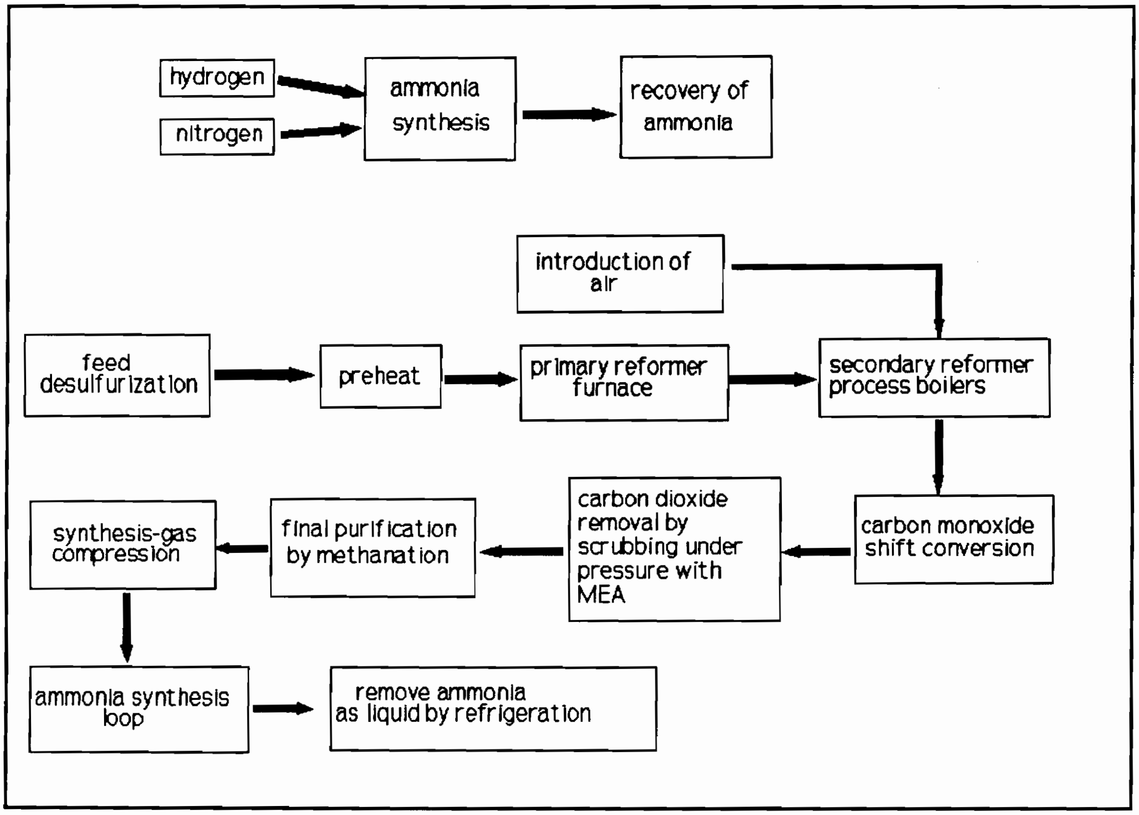

hydrogen
nitrogen
ammonia synthesis
recovery of ammonia
introduction of air
feed desulfurization
preheat
primary reformer furnace
secondary reformer process boilers
carbon monoxide shift conversion
carbon dioxide removal by scrubbing under pressure with MEA
final purification by methanation
synthesis-gas compression
ammonia synthesis loop
remove ammonia as liquid by refrigeration

Figure 10.8. Semantic network of the ammonia process structure.

F. Frame-Based Systems

1. Introduction to Frames

A *frame* is a grouping of properties that a given object, situation, or process has. It is a data structure that organizes the properties of a situation into a hierarchy, and therefore, is closely related to semantic networks. Humans tend to classify and group items to solve problems, and a frame attempts to emulate this tendency. Figure 10.9 offers an example of this kind of classification.

An engineer knows that many different types of pumps exist. He also knows that all pumps have certain generic properties. We can group these properties into a frame:

```
pump (
      class: general,
      unit: _x1
      type: _x2,
      material: _x3,
      capacity: _x4,
      motor: electric,
      head: _x5,
      temp : _x6,
      inlet: _x7,
      outlet: _x8).
```

In this example, the : is used as an operator. The attributes **type**, **material**, **capacity**, etc. are in areas known as a *slot*. A slot is a property location. The above frame has ten slots. The slot is *unfilled* when a variable is present (**_x1**, **_x2**, **_x3**, etc.). The two *filled* slots above tell us the class of the frame (**general**), and the type of motor (**electric**).

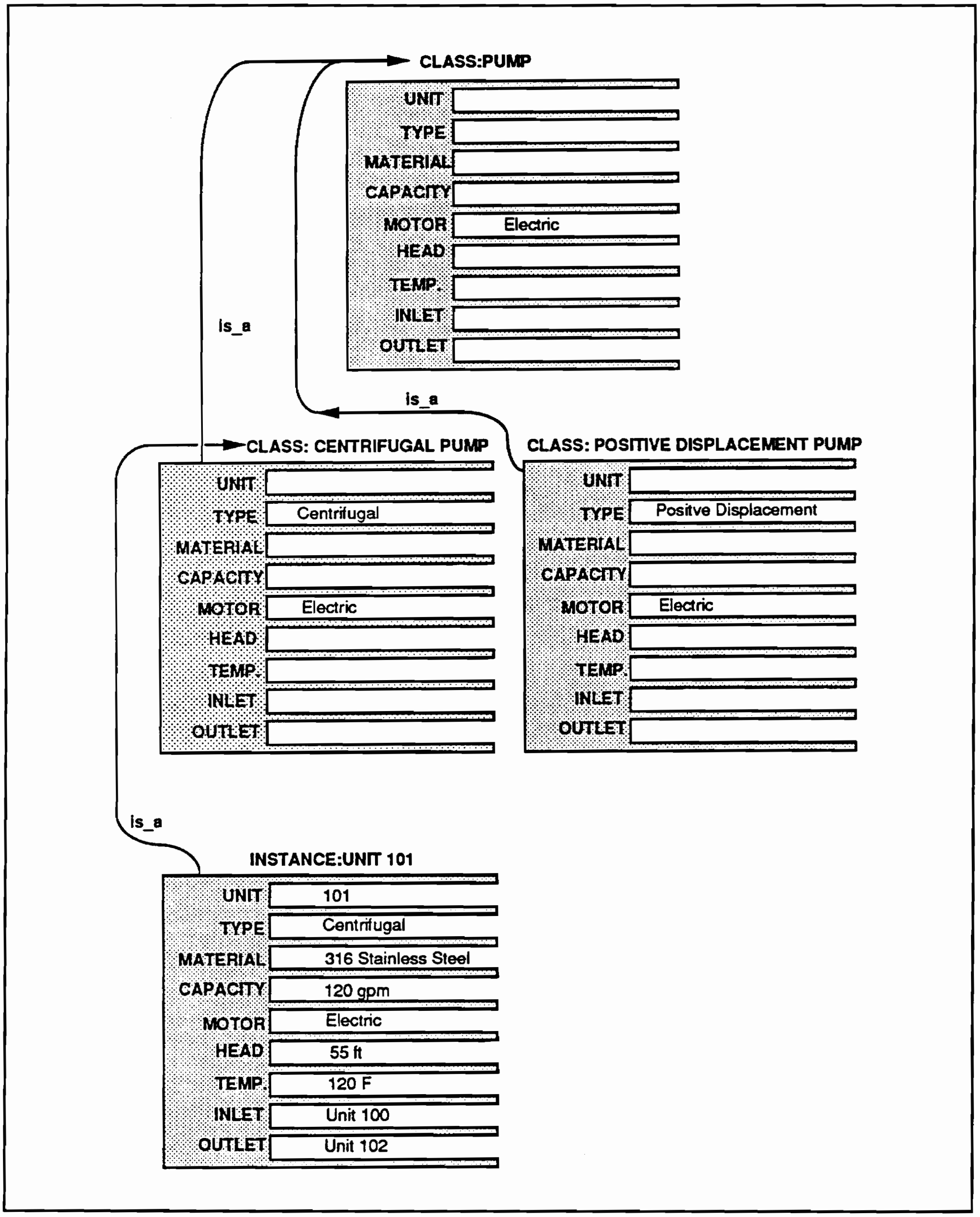


Figure 10.9. Frame representation of a pump.

What are the differences between frames and semantic networks? Frames are more structured. They organize specific information around a specific object, such as the attributes of a pump. When used in a program, the pump frame's structure does not change with program execution. Instead, attributes within the slots change. For example, we may use our pump frame for pump #1 and pump #2. We have two different frames, but each frame maintains the same structure. To distinguish between the frames, we change slot values.

In contrast, semantic networks are more fluid and less structured. A semantic network for pump #1 may be entirely different from that for pump #2.

In addition, frames differ from semantic networks in that there may be attached procedures to take if the frame is accessed. Semantic networks declare relations only, with no attached or implied procedures.

1.1 Inheritance and Default Reasoning

When an engineer looks at pumps in a plant, he sees both positive displacement and centrifugal pumps present. We may write in Prolog

```
isa(centrifugal_pump,pump).
isa(positive_displacement_pump,pump).
```

In addition, the engineer sees that unit 101 is a centrifugal pump:

```
isa(unit_101,centrifugal_pump).
```

What we are doing here is building a hierarchy. Just as in semantic networks, unit 101 *inherits* the properties of a centrifugal pump because it belongs to that *class*. In addition, both centrifugal and positive displacement pumps inherit the properties of pumps because they belong to that *class*. A frame represents a class when it has additional frames (i.e., "children") underneath it. Frames at the top of the hierarchy, i.e., those with children but no "fathers," belong to a *general* class of frames or no class at all. They essentially create their own class for additional frames lower in the hierarchy. Frames that are in the middle of the hierarchy and have additional frames both above it and below it denote a *subclass*. Frames at the bottom of the hierarchy, i.e., those with no children, are called *instances* and do not introduce a new class. This principle is shown in Figure 10.10.

Thus, in Figure 10.9, the pump frame forms a class. The centrifugal pump frame is in the class of pumps, and forms its own sub-class of centrifugal pumps. The unit_101 frame, however, is an instance, since their are no frames below unit_101 in the hierarchy. Properties within the frame are inherited through filled slots. One filled slot in the pump frame is the electric motor. Therefore, we would say that by default, positive displacement and centrifugal pumps have electric motors.

Frame-based systems use a network of nodes connected by relations and organized in a hierarchy, just like semantic networks. As seen in

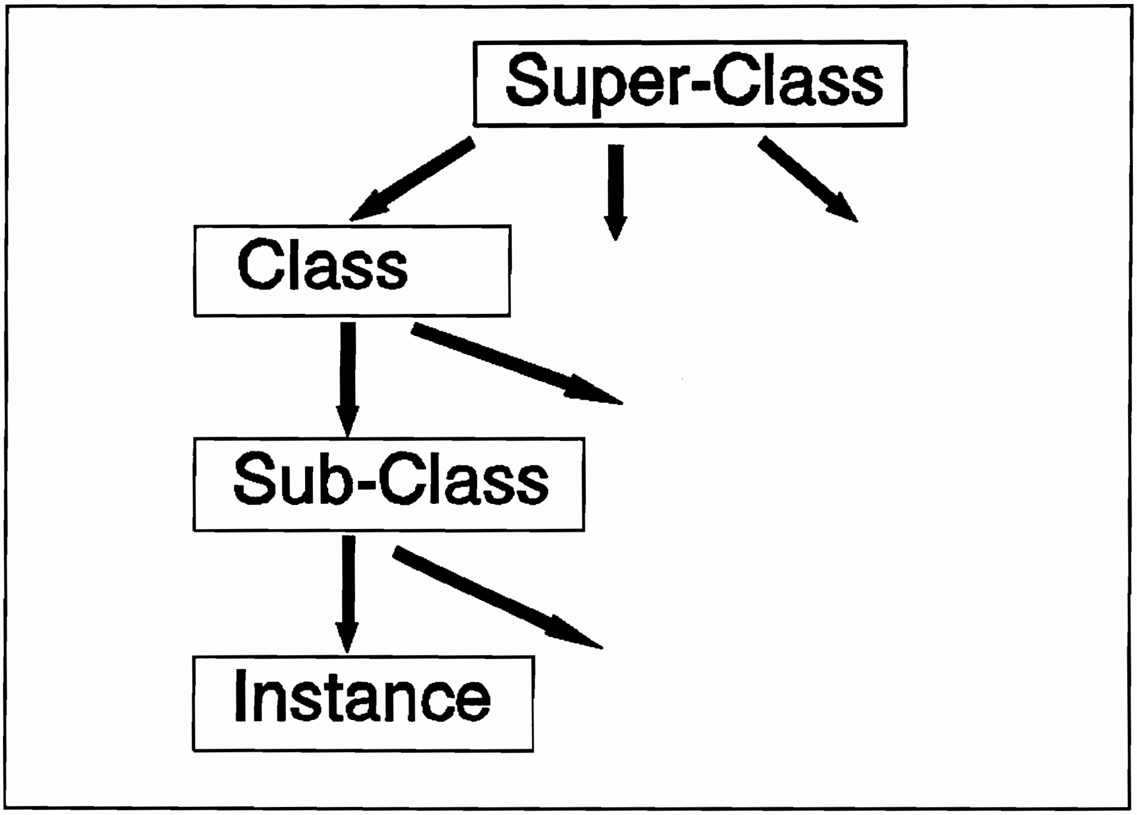


Figure 10.10. Hierarchical inheritance in a frame-based system.

Figures 10.9 and 10, nodes lower in the hierarchy *automatically* inherit the properties of higher-level nodes. For example, a centrifugal pump inherits the properties of the general class of pumps. The isa predicate causes automatic inheritance.

Because of automatic inheritance, frame-based systems support *default reasoning*. Default reasoning gives us the ability to draw conclusions in the absence of information. When no specific information exists, the system uses inheritance to define properties and relations by default. These "default properties" are suitable for use by the expert system, and enable the system to continue its inferences until it reaches definite conclusions.

As an example of default reasoning, let us again consider Figure 10.9. We assume that the user supplies all the information about pump 101 to frame **unit_101**. However, he fails to specify the **type** and **motor** slots, so that when the program begins, these slots are empty. During execution, the program requires information on the **type** and **motor** of **unit_101**. We appear to be trapped; we need information about an object, but that information was not supplied by the user.

In a rule-based expert system using Prolog, the information about **type** and **motor** does not exist. Therefore, the inference chain breaks down and Prolog fails. Prolog may, in fact, respond with an abrupt ***no***.

A frame-based system, in contrast, implements *default reasoning* using inheritance. Since no information is supplied by the user about **type** and **motor**, these values become, respectively, **centrifugal** and **electric**, by inheritance. The slots are filled as shown in Figure 10.9. Now that these slots are filled, the program has its required information and can continue. Default reasoning has enabled the analysis to continue in the absence of specific, user-defined information. Default reasoning gives the system a sense of "intuition."

Frame-based systems employ not only single, direct inheritance, but also *multiple inheritance*. Multiple inheritance happens when a particular frame has two is_a links. For instance:

```
is_a(unit_101,centrifugal_pump).
```

is_a(unit_101,acid_recycle_pump).)

Because **unit_101** is both a centrifugal and an acid recycle pump, it inherits information from more than one source. When this occurs, *local values* for the unit_101 frame override ***default values*** inherited from higher-level frames. Multiple inheritance is shown in Figure 10.11. In this figure, a frame in a sub-class undergoes multiple inheritance from frames in a class.

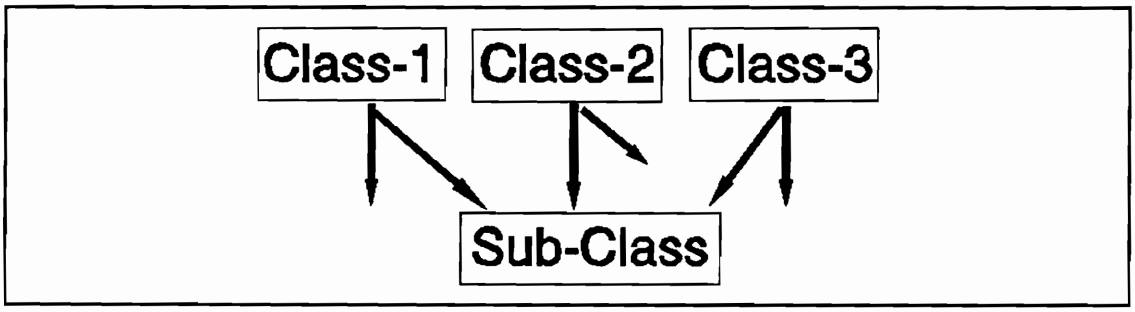


Figure 10.11. Multiple inheritance in a frame-based system.

To summarize, a frame is a grouping of properties that a given object, situation, or process has. Frames are organized into a hierarchy, where lower-level frames can inherit information from higher-level frames. Frames support multiple-inheritance, where a lower-level frame inherits information from two or more "parent frames" because that lower-level frame is connected to *both* parent frames. When performing inheritance with frames, we must always let local values override inherited values. This summary is shown in Figure 10.12.

1.2 Procedural Attachments to Frames

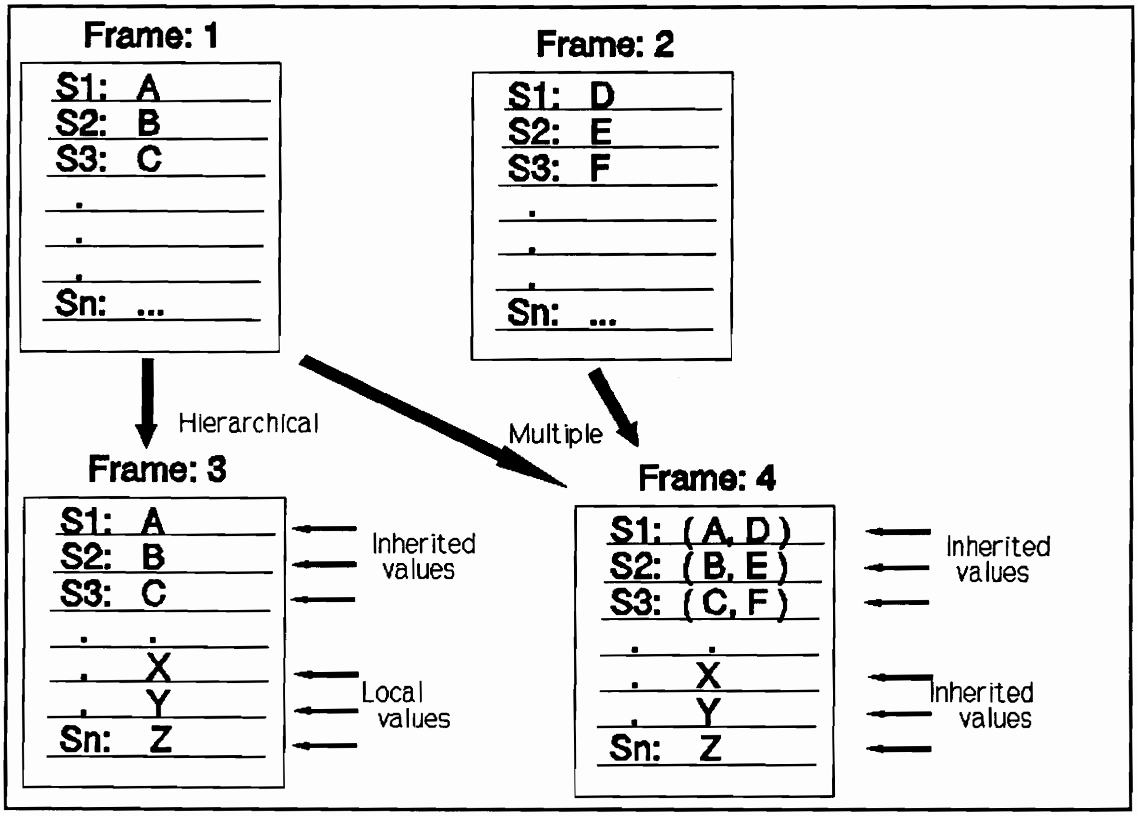


Figure 10.12. A summary of the structure of a frame-based system.

When an AI program processes frames, it manipulates their slots. Depending on the type of manipulation, the system may implement certain procedures. We say that each slot has procedures *attached* to it. These procedures automatically execute based on what we do to the slot. The procedures are:

- *If-added procedure*: executes when new information is placed in the slot.
- *If-needed procedure*: executes when information is needed, but the slot is empty.

- *If-removed procedure*: executes when information is deleted from the slot.

These *procedural attachments* are important in frame-based systems. First, they are a good program-control tool. They monitor the assignments in the slots and take appropriate actions when required.

Secondly, procedural attachments mimic the human ability to take specific actions based on the situation. If that situation changes, reason suggests that certain actions will not solve the problem and therefore should not be tried. The procedural attachments in frame-based systems allow for this situation-specific response. They facilitate powerful problem-solving based on the properties of a situation and associated expectations. Frame-based system, therefore, impart a special property, called *expectation-driven processing*.

As an example of an *if-added* procedure, let us consider again the **unit_101** frame. At the beginning of program execution, the **temperature** slot is empty, but there is an attached *if-added* procedure. We indicate this status by the two Prolog facts:

```
unit_101(temperature,value,_x).
unit_101(temperature,if_added,[include_c,include_k]).
```

During program execution, the **temperature** slot is filled. The fluid temperature is 120 °F, and the slot in Prolog is:

```
      unit_101(temperature,value,[120, f]).
```

However, there is an *if-added* procedure that must be executed if we add information into the temperature slot. Since the value **[120, f]** was just added, we must execute the *if-added* procedure. There are actually *two* attached procedures here, the **include_c** and **include_k** procedures. These procedures say to include slot information not only in °F, but also in °C and in K. We execute these procedures, and at the end of the execution, the slot has been updated to:

```
      unit_101(temperature,value,[ [120,f], [48.89,c], [322.04,k] ]).
      unit_101(temperature,if_added,[include_c,include_k]).
```

Thus, the *if-added* procedure took the 120 °F entry that was added to the slot, calculated conversions between °F and °C, as well as °F and K, and asserted this additional information into the slot. If, later in the program, the temperature is needed in °C or K rather than °F, the slot is ready to supply that information.

We summarize the advantages of frame-based expert system in Table 10.6.

Table 10.6. Advantages of a frame-based system.

- Provides better structure and organization of the knowledge base.

- Promotes expectation-driven problem solving.
- Stores large blocks of related information as a single entity.
- Utilizes automatic inheritance.
- Allows for default reasoning.
- Captures the hierarchal nature of knowledge through inheritance.
- Captures the inter-related nature of knowledge through multiple inheritance.

---

2. Uses of Frame-Based Systems

The most common use of frames is to systematically organize large amounts of information for the expert-system program. In this mode, a combined rule-based and frame-based system is used. The rules are used to reason through the inference chain, and the frames are used to store and classify large blocks of data for quick and easy access. The *pump frame* of Figure 10.9 can be used for this purpose. Also, a number of commercial software packages exist for this type of frame-based implementation. An example is KEE (Knowledge Engineering Environment), which utilizes the language LISP.

Frames can also be used for *situational* analysis. Slots can be filled with process data, and based on this information, frames can be used for:

(1) Pattern Recognition - such as in fault diagnosis;

(2) Data Interpretation- such as in quality control, analytical

chemistry, and process fault diagnosis;

(3) Process prediction- such as in process control.

G. Object-Oriented Programming

1. Introduction

Object-oriented programming (OOP) is related to frame-based systems, with some notable exceptions. OOP was developed to overcome some of the drawbacks realized with frame-based systems.

What are some of the drawbacks of frame-based systems that OOP overcomes? One drawback of the frame-based knowledge is the separation of data and action-oriented computer code. For example, we use relations such as **isa** to tie frames and objects together in an expert system, and this use can create difficulties. When we say "hydrochloric acid **isa** corrosive material" or "unit_101 **isa** pump," each **isa** has a different meaning. The meaning depends on the context in which **isa** is used. We must implement a procedure, and that procedure depends on the objects the **isa** atom is relating. Therefore, to run properly, we must combine the data and the corresponding computer code into a single unit. This combination is the essence of *object-oriented programming.*

Object-oriented programming combines data and computer code together into a single, inseparable "object." This object contains not only a list of properties, but also the *required procedures to manipulate them.* In that sense, object-oriented programs closely resemble frame-based expert systems. However, object-oriented programs go beyond frame-based systems. Frame-based systems define the properties of object classes, but cannot communicate between different classes. Object-oriented programming overcomes this limitation by easily allowing us to relate to different classes.

Object-oriented programs differ from frame-based systems in that *each object can communicate to another object by sending "messages."* This is not true in the frame-based system of Figure 10.9, for instance, where **centrifugal_pump** cannot communicate with **positive_displacement_pump**. In object-oriented programming, any object may communicate directly with any other object, regardless of where each is in the frame hierarchy.

To see how object-oriented systems work, let us consider Figure 10.10. We start an object-oriented program by "calling" an object with a message. Based on the input message, the object then consults its database to see what procedures to undertake. The object may send messages out to other objects. All the while, the frame-based hierarchy of inheritance continues to be enforced.

Objects, then, are self-contained units that possess data in groups (just as in frames) and the ability to take action by themselves.

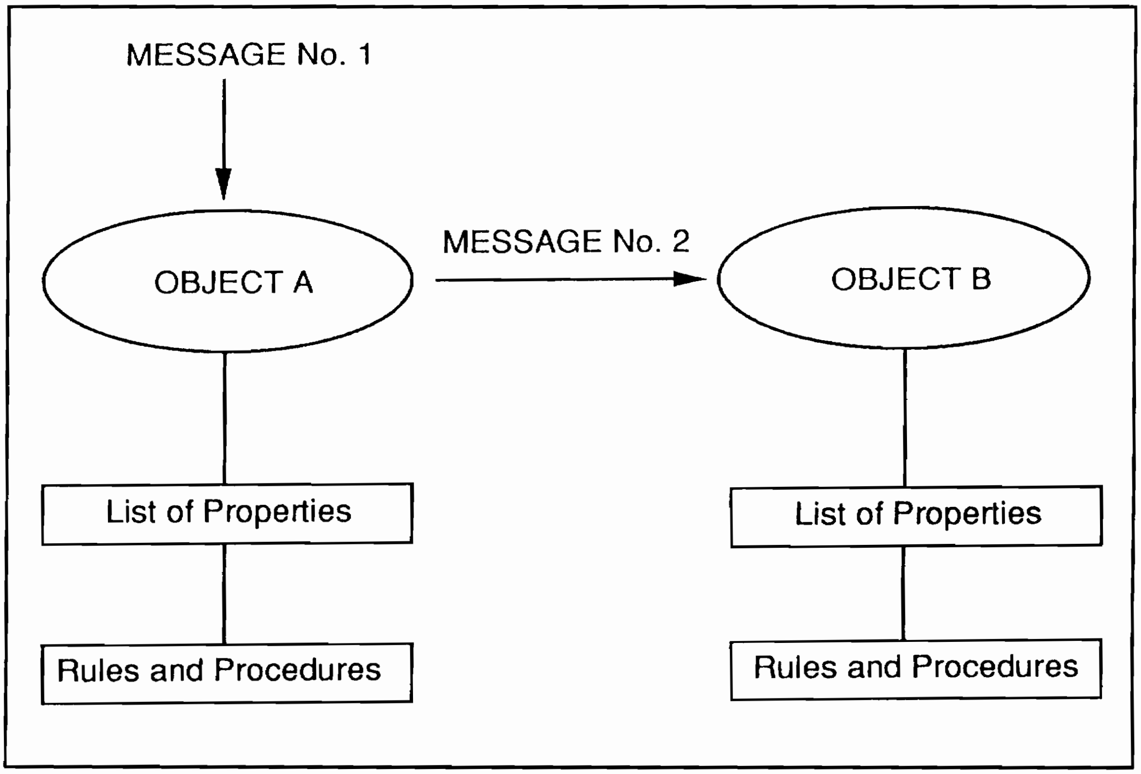


Figure 10.10. Passing messages in object-oriented programming.

2. Structuring an Object-Oriented Program

Ideally, in an object-oriented program, the only data structures that exist are the objects (Ungar and Venkatasubramanian, 1990). All program execution and problem-solving is achieved by calling objects, and having these objects communicate with each other. Recently, however, this has changed somewhat; there have been some examples of OOP where an umbrella control structure (such as a rule-based system) controls to a certain extent how the objects interact (Schnupp et. al., 1989, pp.121-146). We discuss this aspect in more depth in section 11.2D.

Objects have a specific structure. We define four aspects of an object's structure (Schnupp et. al., 1989):

1. The object must have a *name*. This name identifies the frame to call and access.

2. The object must have *slots*. These slots are equivalent to those of frame-based systems. Slots are entries or arguments in the object (frame) that classify information about the object (frame). These slots use the principle of inheritance from higher-level hierarchies, and may also have attached *if-added*, *if-needed*, and *if-removed* procedures.

3. The slot must have an *interpretation*. The interpretation tells us what to do with information in the slot. Is the argument in the slot to be interpreted simply as data? Or is the argument a call for a procedure? Depending on the interpretation, we may take additional action.

4. The object must contain the actual information stored in the slot. This information can be data (if the slot houses data) or a call to a clause in the program (if the slot houses a procedure).

Let us illustrate the properties of an object through an example. Object-oriented programming can be implemented in Prolog or LISP; LISP has some built-in functions that make it more natural for object-oriented programming. Nevertheless, Prolog can do OOP, and we demonstrate the organization of an object-oriented program through Prolog. There are several ways to implement objects in Prolog. One way is to use a Prolog fact with the functor identifying the object:

```
object_name(
      is_a(class),
      attribute_1(interpretation, [argument_11]),
      attribute_2(interpretation,[argument_21,argument_22,argument_22]),
      attribute_3(interpretation, [argument_31,argument_32])).
```

For example, the **centrifugal_pump** frame, shown in Figure 10.9, may be written as:

```
      centrifugal_pump(
            is_a(pump),    % centrifugal_pump belongs to the class of pumps
            unit(value,_x1), % "value" means the argument is to be inter-
            type(value,_x2),  % preted as data rather than procedure.
            material(value,_x3),
            capacity(value,_x4),
            motor(value,[electric]), % data slots are empty except here
            head(value,_x5),
            temp(value,_x6),
            inlet(value,_x7),
            outlet(value,_x8),
            pump_curve(procedure,[GPM,Head])). % a required procedure.
```

The last slot in the object is a procedure the generates the pump curve, and stores the gallons per minute (gpm) in a list instantiated to variable

GPM, with each pump head corresponding to the gpm value stored in list **Head**.

If we look closely at this approach, some negatives are evident. We must know the arity (i.e., number of slots) and the locations of these slots to properly access the information. This requirement can be tedious and error-prone. Another way to implement a frame, and avoid the requirement of knowing the arity and slot locations, is to use a *set of facts* rather than just one fact. The basic framework for the fact is:

```
object_name(Slot,Interpretation,ValueList).
```

Thus, the same **centrifugal_pump** frame shown in Figure 10.9 and discussed above can be represented by the following facts:

```
centrifugal_pump(pump,is_a,[]).   % centrifugal_pump belongs to the
                                  %  class of pumps
centrifugal_pump(unit,value,_x1).
centrifugal_pump(type,value,_x2).
centrifugal_pump(material,value,_x3).
centrifugal_pump(capacity,value,_x4).
centrifugal_pump(motor,value,[electric]). % This slot is filled
centrifugal_pump(head,value,_x5).
centrifugal_pump(temperature,value,_x6).
centrifugal_pump(inlet,value,_x7).
centrifugal_pump(outlet,value,_x8).
centrifugal_pump(pump_curve,procedure,[GPM,Head].
```

Now, we call the attached procedure to generate the pump curve. We implement the clause **execute_procedure**:

```
execute_procedure(Object(Slot,procedure,ValueList)):-
            X =.. [Slot|ValueList],
            call(X).
```

We use the *univ* predicate, =.., which is built-in to Prolog. If the interpretation is not instantiated to **procedure**, the clause fails. Otherwise, the clause calls the relation with the name **Slot**, and that relation instantiates **ValueList**.

To give an example of facts arranged in an object-oriented program, let us consider the frame representation of Figure 10.9, with attached procedures. The Prolog facts grouped in an object-oriented program are shown in Figure 10.14.

3. Properties of Object- Oriented Programs

Stefik and Bobrow (1985) give a good summary of the properties of an object-oriented program (or computer language).

```
pump(general,is_a,[]).
pump(unit,value,_x).
pump(type,value,_x).
pump(material,value,_x).
pump(capacity,value,_x).
pump(motor,value,_x).
pump(head,value,_x).
pump(temperature,value,_x).
pump(inlet,value,_x).
pump(outlet,value,_x).
pump(pump_curve,procedure,[[GPM],[Head]]).

centrifugal_pump(pump,is_a,[]).
centrifugal_pump(unit,value,_x).
centrifugal_pump(type,value,_x).
centrifugal_pump(material,value,_x).
centrifugal_pump(capacity,value,_x).
centrifugal_pump(motor,value,[electric]).
centrifugal_pump(head,value,_x).
centrifugal_pump(temperature,value,_x).
centrifugal_pump(inlet,value,_x).
centrifugal_pump(outlet,value,_x).
centrifugal_pump(pump_curve,procedure,[GPM,Head].

positive_displacement_pump(general,is_a,[]).
positive_displacement_pump(unit,value,_x).
positive_displacement_pump(type,value,_x).
positive_displacement_pump(material,value,_x).
positive_displacement_pump(capacity,value,_x).
positive_displacement_pump(motor,value,_x).
positive_displacement_pump(head,value,_x).
positive_displacement_pump(temperature,value,_x).
positive_displacement_pump(inlet,value,_x).
positive_displacement_pump(outlet,value,_x).
positive_displacement_pump(pump_curve,procedure,[GPM,Head]).

unit_101(centrifugal_pump,isa,[]).
unit_101(unit,value,[101]).
unit_101(type,value,[centrifugal]).
unit_101(material,value,[stainless_steel_316]).
unit_101(capacity,value,[120,gpm]).
unit_101(motor,value,[electric]).
unit_101(head,value,[55,feet]).
unit_101(temperature,value,[120,f]).
unit_101(inlet,value,[unit_100]).
unit_101(outlet,value,[unit_102]).
```

Figure 10.14. Prolog facts grouped into an object-oriented program.

Six essential properties are: ***data abstraction***, ***encapsulation***, ***inheritance***, ***message passing***, ***polymorphism***, and ***modularity***. They are discussed below.

3.1 Data Abstraction

Data abstraction refers to the level of detail that the knowledge representation encompasses when representing a more complicated idea or object. At high levels of abstraction, we are concerned more about the problem "overview" rather than fine details. High levels of abstraction are generally more qualitative, and encompass broader principles and trends. Focusing on the overall picture and not getting engrossed in fine details promotes better understanding of the problem, and prevents us from getting bogged down. On the other hand, lower levels of abstraction may be needed for more in-depth analysis when higher levels fail.

As an example, we consider an engineer who is troubleshooting a reactor. The reactor currently has a low flow rate of feed to it, and in addition, the feed pressure is low. At a *high level* of abstraction, the engineer begins to look for possible origins of this problem. He immediately generates some plausible explanations: 1) the reactor feed pump is malfunctioning, 2) the reactor pre-heater, a shell-and-tube heat exchanger, has an excessively high pressure drop, or 3) there is a leak somewhere in the feed line.

To determine which explanation is correct, the engineer moves to a lower level of abstraction. He traces lines, looking for leaks. He performs material balances where possible. He determines the pressure drops across equipment. He gets increasingly more detailed, and usually more quantitative, until the problem is solved.

Object-oriented programs frequently have the ability to operate at different levels of abstraction. *To define a new data abstraction, we typically need to define a new object.*

3.2 Encapsulation

Encapsulation refers to the scope of objects, i.e., what responsibilities each object has and how each object interacts. Encapsulation defines the grouping of and between objects, and how these objects interface with each other. Encapsulation is related to the *class* that the object is in. Objects within the same class typically have the same responsibilities. *To create a separate encapsulation, we usually create a separate class.* Let us investigate why.

When we design an object-oriented program, we must decide which objects will perform what procedures. This decision is a task in encapsulation. For example, let us consider "utility" procedures that aid the input-output function of the program. We have a procedure called **graph(xlist, ylist)** that plots the function $y = f(x)$, where **xlist** is a list that houses the values for the independent variable, x, and **ylist** is a list that houses the values for the dependent variable, y.

When we incorporate the **graph(xlist, ylist)** procedure into an object-oriented program, we must decide where to put the procedure and the data. There are two routes we can take: 1) the *wide-open* route, where procedures are allowed to access data from *any* object; and 2) the *restricted* route, where procedures can only access data only *from within* the object in which the procedure is found. Thus, no data can be accessed from outside of the object. The route we choose for encapsulation can dramatically affect program performance.

The wide-open route allows for great program flexibility. We can write procedures and access data anywhere. Unfortunately, however, this benefit is not without cost. Program reliability decreases with the wide-open approach. The programs also become much more difficult to read and debug. Program maintenance is higher. For these reasons, many researchers (King, 1990) recommend the restricted approach.

Typically in the restricted approach, each class of objects has the same encapsulation. Each object in a specific class typically possesses similar procedures. Procedures can access data only within their own objects, so for the purpose of continuity, these objects (all in the same class) usually interact with other objects in the same way. That is why we say: *to create a separate encapsulation, we need to create a separate class.*

As an example of encapsulation, let us consider Figure 10.15. We have two objects, **ABSORBER** and **DISTILLATION** under the class of separators. Each object has its own data and procedures. Because of principles of encapsulation, the objects are remarkably similar and perform the same functions. Each object implements a *design equation* (procedure) followed

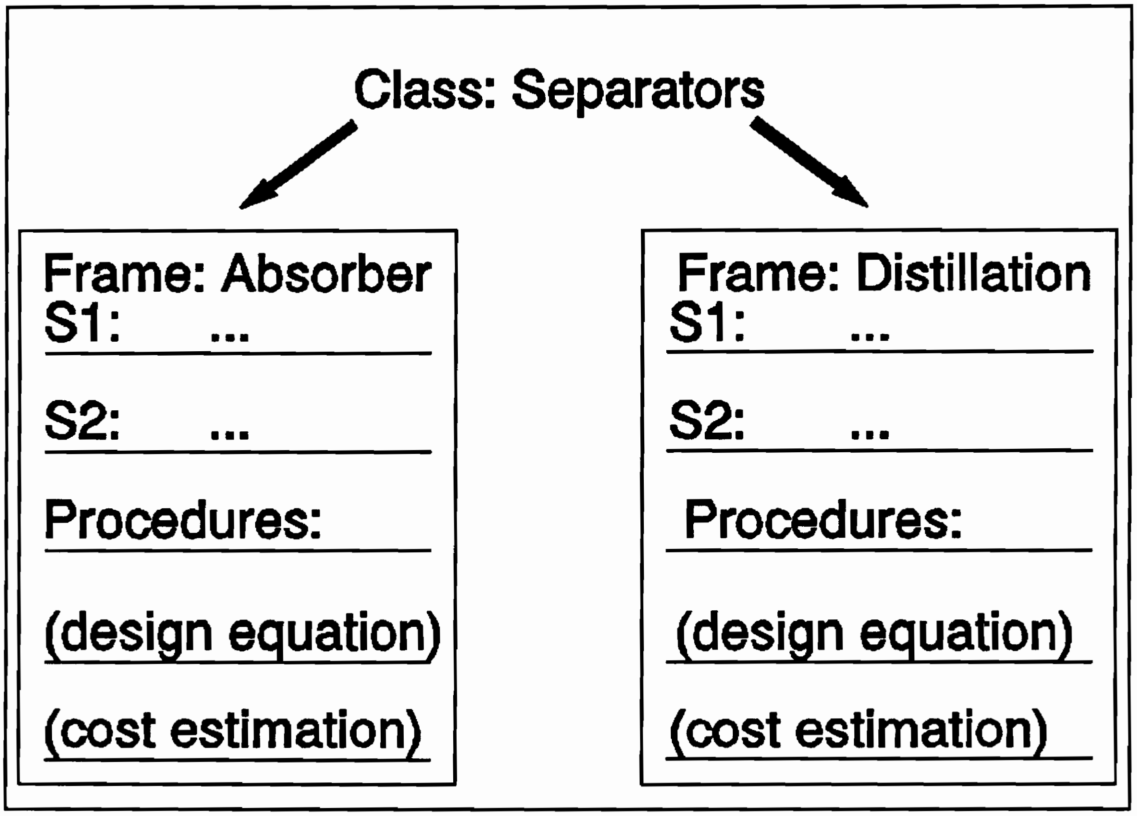


Figure 10.15. Encapsulation in object-oriented programming.

by a *cost estimation* (another procedure). These equations, of course, are unique to the object itself (the **ABSORBER** object implements the design and cost equations for absorption; the **DISTILLATION** object does likewise for distillation). Each object has the same encapsulation, i.e., the same responsibilities and interactions, since they both belong to the same class.

3.3 Inheritance

Inheritance in object-oriented programming is closely related to inheritance in frames. Objects that are lower in the hierarchy inherit slot-properties from objects higher in the hierarchy. However, local values to a specific slot override the default values that may arise from inheritance. As we move down the hierarchy of objects, each object gets increasingly *specialized.*

In addition to the standard hierarchical inheritance, an object can participate in multiple inheritance. Multiple inheritance occurs when an object has two or more parent objects. This principle is shown in Figure 10.16, where *Object 3* participates in hierarchical inheritance only, and *Obect 4* participates in multiple inheritance.

Note, however, that inheritance applies to *data only, and not to procedures*. Procedures are unique and local to each object, and cannot be inherited. This principle is also shown in Figure 10.16.

The reader may wish to compare Figures 10.12 and 10.16 to see the similarities and differences between a frame-based system and an object-oriented program.

Inheritance has a number of benefits. We can define new objects in terms of already existing objects, thereby simplifying expansions of the program. Inheritance provides us with a means of developing more specialized cases from more general ones. When a new case is needed, instead of generating a whole new class of objects, we inherit the essential aspects from an object higher in the hierarchy and add local values to this new object to reflect its uniqueness.

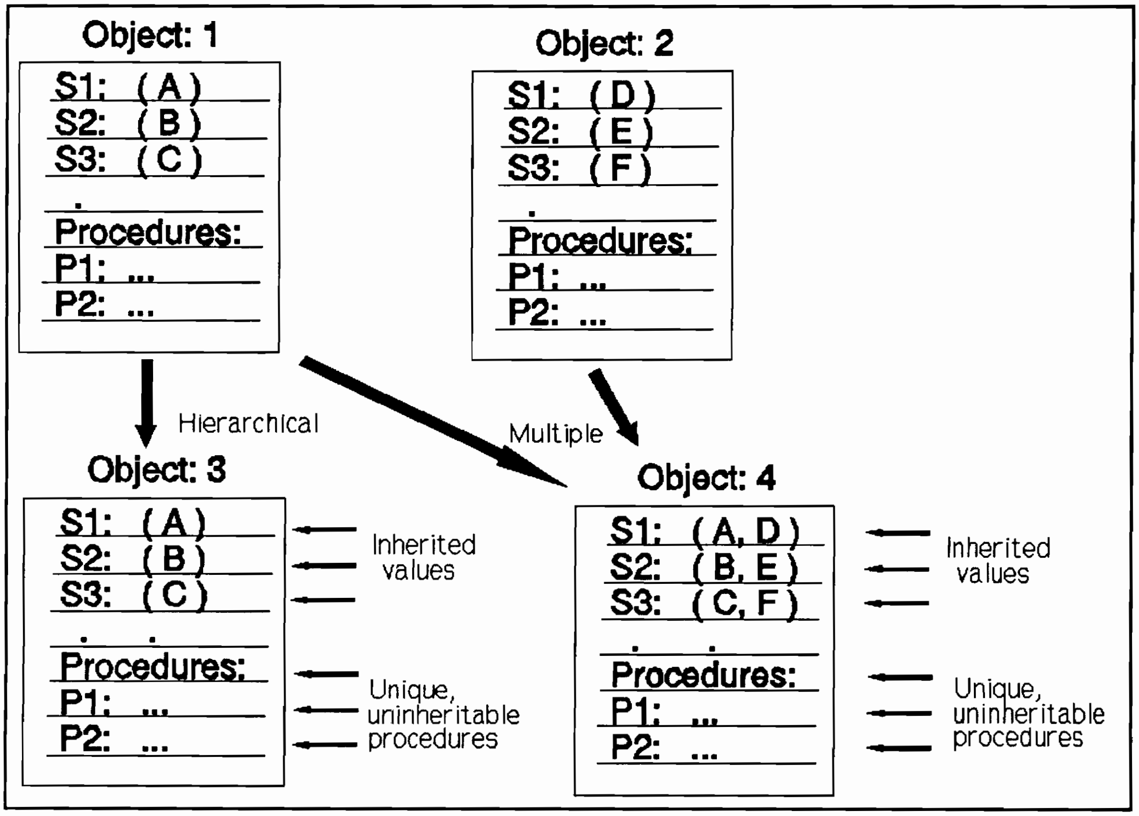


Figure 10.16. Inheritance from objects.

3.4 Message Passing

In contrast to frame-based systems, objects communicate with each other in an OOP environment. Objects communicate and interact with each other by passing *messages*. Ideally, *all* problem-solving and program execution is achieved through message passing. (This requirement has been relaxed somewhat in an effort to speed-up OOP programs. Schnupp et. al. (1989) discuss an OOP environment where program execution and problem-solving are not necessarily restricted to being achieved through message passing.) An object is said to "send a message" when it executes one of its procedures. The procedure typically specifies what operation to perform and what object to call. The actual implementation of the procedure is left up to the object that is called. For example, in our **pump** object of Figure 10.14, the slot:

```
pump(pump_curve,procedure,[GPM,Head]).
```

invokes a procedure to determine the pump curve (it calls the **pump_curve** object), and wants the relation of GPM vs head stored in two lists.

An object that receives a message from another object relies on its own procedures to decide what to do. The receiving object can respond to the request, or do nothing at all. For example, let us consider when the **pump** object calls the **pump_curve** object to generate a pump curve, i.e., GPM vs. head, for a pump. The **pump** object leaves all processing up to the **pump_curve** object. The **pump_curve** object may invoke additional procedures or send out additional messages that the **pump** object has no knowledge of.

If we wish to generate the pump curve in a program written in FORTRAN, we may *directly* call a function or a subroutine. In OOP, however, we achieve the same task *indirectly* by calling an object. Thus, in OOP, data and procedures are *localized* (to objects). This localization can simplify a complex program and make program maintenance easier. In addition, since objects define their own procedures, each object in the same class can be made to respond in a similar way to the same message set (a principle of encapsulation).

3.5 Polymorphism

Polymorphism means "many shapes", and is an appropriate name for this characteristic of object-oriented programs. Polymorphism can be defined as the capability of different classes of objects to respond to the exact same set of messages in a unique way that is most appropriate for those receiving objects. Thus, we may send the exact same message to different objects, and get completely different results.

Another characteristic inherent in polymorphism is the ability of a procedure within an object to respond in varying ways depending on the data. Based on the message received from the sending object and the data contained in the receiving object, that receiving object may invoke different procedures.

Polymorphism is attainable because of *dynamic binding* of variables. "Binding" refers to how and when a variable takes on a value in a computer program. To understand dynamic binding, let us first get a grasp of its opposite: *static binding.* In a traditional, compiled, procedural computer program, the entire program, including all procedures and functions, is linked into a single program. This linking process "locks-in" the variable bindings, and hence, the name static binding. The addresses of all variables, functions, and procedures are identified and allocated a specific location in the program. Thus, in static binding, variable bindings are determined before the program ever executes.

OOP, as well as AI in general, is not this way. Variable bindings are dynamic, and occurs during program execution. Variables are instantiated and uninstantiated as the program runs. When a procedural call is finally made, difference in variable bindings (i.e., data) can direct the object to undertake different procedures.

As an example of polymorphism, let us assume we lose pressure in a saturated-steam header; an object-oriented, fault-diagnosis program monitors the process. This fault-diagnosis program could send a message to *any* object that may be affected, and allows that object to determine effects and take corrective action. Depending on the status of the process, the receiving object may recommend different procedures for different situations. This interaction is polymorphism at work. An object that represents a unit operation relying on the saturated steam may, for instance, prepare for a shutdown, since the steam pressure has been lost.

To demonstrate polymorphism using our steam-pressure example, we may send the following message:

```
steam_header(X,corrective_action,[low_steam_pressure]).
```

This statement contacts *any* object **X**, and tells it to take corrective action (if any is needed) because we lost steam pressure.

3.6 Modularity

Program modularity is natural in object-oriented programs. Because all execution is achieved through objects, the programs are inherently modular. The modular nature of OOP makes this type of programming more convenient. Program maintenance is easier, since we are only concerned about local modules. In addition, program expansion is also easier, since we can expand modularly.

4. Applications of Object-Oriented Programs

How might we use object-oriented programming? One particularly suitable use is in chemical-process fault diagnosis. Consider again the steam-pressure example in the previous section. Assume that the we are running a low-pressure steam unit. This steam operation serves an entire oil refinery by supplying saturated steam to the plant's utility-steam header. We have an object-oriented expert system monitoring plant operations.

Now assume that the steam unit loses pressure. The expert system detects this. With an object-oriented expert system, the steam unit may broadcast to all other control rooms that a pressure loss has occurred. Based on this message, control rooms throughout the refinery can respond accordingly to minimize the problem. They may tell the steam unit that they are switching to a different source of steam. Or, if no alternate sources exist, they may prepare for a shutdown if the pressure loss is expected to be prolonged. Thus, the object-oriented expert system allows us to simultaneously control unrelated (or distantly related) operations.

One better-known prototype research program that takes an object-oriented approach is DESIGN-KIT (Stephanopoulos et. al., 1987). DESIGN-

KIT's agenda is very ambitious. It is a software-support tool developed to aid process-engineering activities such as: flowsheet development, control-loop configuration, and operational analysis. DESIGN-KIT also supports graphic constructions. Some objects included in DESIGN-KIT, along with data included in each object, are:

- *Graph*- includes the object name, graphic functions used to create it, and graphic procedures such as move, expand, and rotate.
- *Processing Unit*- includes the object name, input and output streams, modeling equations, design methodology, constraints, start-up procedures, and potential control loops.
- *Reaction*- includes the object name, reaction mechanism, kinetic-rate expression, catalyst, operating conditions, and yield data.
- *Equipment*- includes the object name, sizing and costing methodologies, costing assumptions, and design and operational constraints.

DESIGN-KIT also has the ability to do equation-oriented simulation and design, forward- and backward-chaining reasoning strategies, and a degree-of-freedom analysis. DESIGN-KIT is written in Common LISP, and is built on IntelliCorp's software package, KEE (Knowledge Engineering Environment).

DESIGN-KIT is only a research prototype, and is focused more on how to *structure* an OOP for process engineering. Consequently, its knowledge base is limited. Implementing a "full-blown" system would be an enormous effort requiring many man-years of time. Nevertheless, by suggesting how to organize an OOP program, and demonstrating its use on a few simple problems, DESIGN-KIT successfully demonstrates the power and flexibility of object-oriented programming.

5. Challenges in Object-Oriented Programming

OOP can be very powerful, but the wise developer will know the limitations and challenges that OOP faces today. We discuss these issues in this section.

5.1 Speed and Efficiency

One of the biggest problems with OOP today is speed (King, 1990). Most systems are simply too slow and inefficient to be practical commercially. One of the main reasons that OOP is slow is *granularity* (Ramamoorthy and Sheu, 1988). What is granularity? In typical OOP systems (such as SMALLTALK), an integer is treated like an object. To add two integers, we must send a message between the two objects. Thus, the system is excessively granular. In simple addition, we desire to just add the numbers together. Unfortunately, we are forced to send a message between two objects of integers, which is clearly less efficient.

Another problem associated with granularity that also affects program speed is system overhead. As we may expect, to solve even a moderate-sized problem, the number of objects in the program is very large. To solve a large, complex problem, the number of object required is astronomical. Because of the very large number of objects, system overhead is high and the program becomes increasingly inefficient.

Another reason for the slowness and inefficiency of object-oriented programs is the lack of good architectural support for the program. Typical programs have thousands of objects and thousands of procedural calls being sent. Data manipulation within the objects as well as implementation of certain procedures are, by necessity, managed by both hardware and software. Reading and writing off a disk is necessary. With limited computing resources, this management task becomes burdensome and loaded with excessive data swapping.

5.2 Systematic Object Management

Object-oriented programs can get very large and complex, and therefore,we need to develop a systematic object-management system. Unfortunately, there is little information on how to systematically organize an object-oriented program, and most programmers are develop their own systems in an ad-hoc fashion. Ramamoorthy and Sheu (1988) give a summary of some issues in object management:

- *Query processing*- query language design, mechanisms to evaluate queries, and performance optimization.
- *Error recovery*- the ability to detect faults and recover when a piece of data is lost or is unable to be calculated.
- *Concurrency control*- the capacity to allow simultaneous accesses

and the ability to control shared objects efficiently.

- *Disk management*- when objects become large, parts of them must be stored in secondary storage. Consequently, efficient disk access is important.

5.3 Clear Programming Methodology

Again, there is little information on effective programming methodologies for object-oriented programs. When programmers develop object-oriented programs, the following issues can be particularly unclear and difficult to resolve (Ramamoorthy and Sheu, 1988):

(1) Deciding whether to make elements of the program objects or not;

(2) Deciding the best way to decompose the problem into classes;

(3) Deciding the best set of messages to pass between objects; and

(4) Deciding which class provides the best procedures to perform desired calculations.

There can be so many objects in the program that these programming-methodology concerns make it very difficult for a programmer to implement an efficient and consistent program. Some programming tools are obviously needed here.

H. Blackboard Systems

1. Introduction to Blackboards

A blackboard system, in and of itself, is not a formal knowledge representation like rule-based and frame-based systems are. Instead, *a blackboard system is an intentional combination of two or more knowledge representations into one single, operating system.*

Why would we want to use a blackboard system? The answer to this question requires an understanding of knowledge representation and problem-solving. Every knowledge-representation scheme has advantages and disadvantages, as we have discussed in sections 10.2A-G. For example, if we want to develop a finite-element analysis program, we better not use a rule-based representation with Prolog. Instead, a procedural, algorithmic representation with a language such as FORTRAN, Pascal, or C is more appropriate.

Applications of traditional rule-based expert systems in engineering are better at in-depth qualitative rather than quantitative reasoning. When complex "number-crunching" is required in these systems, we are out of luck. The knowledge representation simply will not efficiently support extensive floating-point calculations. That is where the motivation for a *blackboard system* comes in. When "number-crunching" is required, why do we not just "shut-down" the rule-based knowledge representation in Prolog, and switch over to a better numerical language such as C? This way, *we effectively integrate the strengths of both languages*, and can perform both complex symbolic computing and complex numerical analysis under one roof. This advantage is one of the central motivations behind blackboard systems.

Figure 10.17 shows a typical blackboard-system architecture. The blackboard has multiple knowledge sources, and these knowledge sources can be declarative or procedural. A control mechanism integrates the various knowledge sources, tells the system when to switch over from one knowledge source to the other, and importantly, keeps track of "data typing."

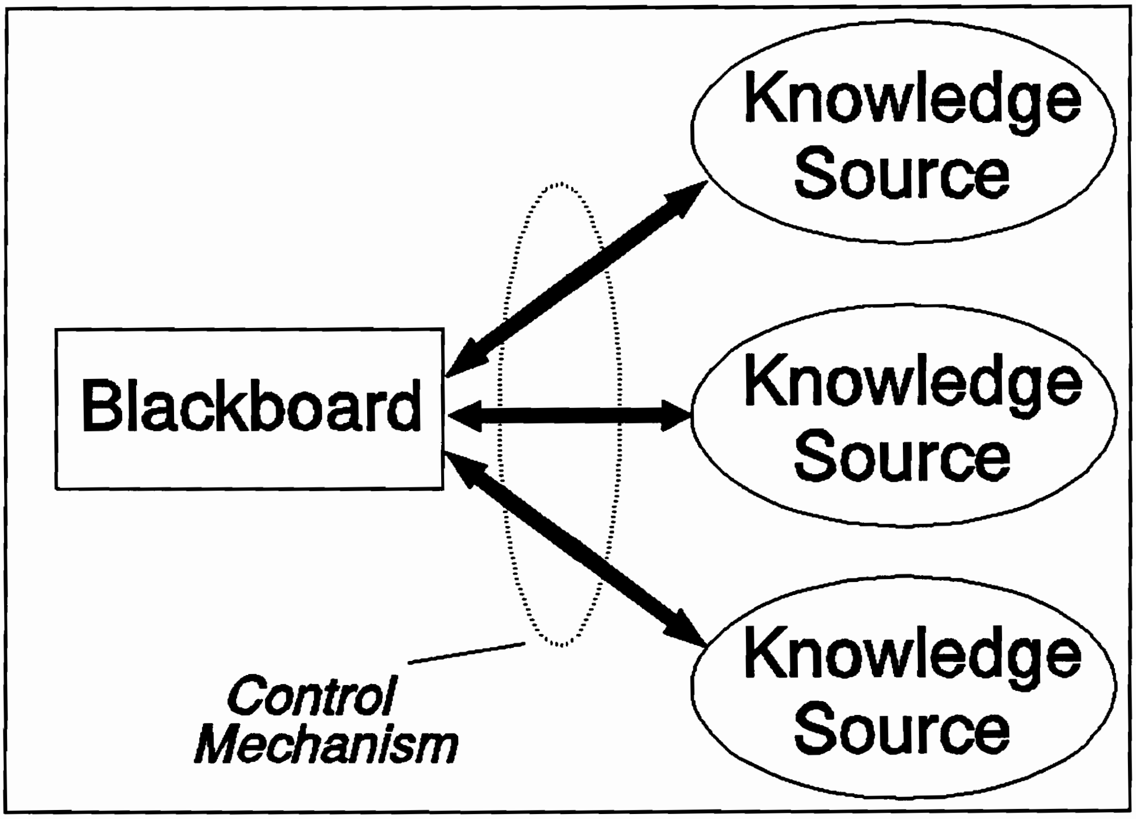


Figure 10.17. The architecture of a blackboard system.

Data type is very important in blackboard systems, especially when we interface between knowledge sources. For example, let us consider the instantiation **X = butane** done in Prolog. In Prolog, this **X** represents an atom. Atoms do not exist in the language C, so if we switch over from Prolog to C and must pass the variable **X**, we may have to change its data type to, for instance, a string. Likewise, in C, we may have an array such as B(I,J). Dimensionalized variables and arrays do not exist in Prolog, so if we wish to pass B(I,J) from C to Prolog, we must convert the variable into a data type that Prolog can use, such as a list. Data type is a critical logistic function that the control mechanism performs. Proper data type is essential if different types of computing are to be integrated successfully.

2. Elements of a Blackboard System

Through blackboard-system development and experience, researchers have identified some important elements of the system that facilitate proper problem-solving. We discuss these elements below.

(1) ***Entries***- These are *intermediate* results, values, or conclusions that are arrived at during program execution. Entries can be any type of information, i.e., data, rules, goals, procedures, or even partial solutions. Typically, *entries* are stored by the blackboard and are reserved for use in a different knowledge source. These entries frequently tell the other knowledge sources what action should be undertaken next.

(2) ***Knowledge Sources***- These are separate, self-contained knowledge

representations that are used for *opportunistic reasoning*. Usually, each knowledge source is chartered with the solving a specific subset of the problem, and the knowledge representation of that knowledge source is specifically geared and tuned to solve that problem subset accurately and efficiently.

Knowledge sources are *separate and independent*. Therefore, they do not communicate to each other. Instead, communication between knowledge sources is controlled through the control mechanism of the blackboard. The control mechanism decides which knowledge source is to execute next, and passes necessary entries to that knowledge source.

Knowledge sources are usually data- or input-driven in nature, and can contain either declarative or procedural knowledge. For example, a knowledge source may implement a rule-based, frame-based, or object-oriented problem-solving approach using a declarative language such as Prolog. On the other hand, we may have another knowledge source that performs a rigorous finite-element calculation using the languages FORTRAN or C. Depending on the type of problem that the knowledge source is trying to solve, we design the appropriate knowledge representation.

3. Control in a Blackboard System

The control mechanism in blackboard systems is critical to the system's success. This control mechanism can be viewed as a "moderator," and determines:

- which knowledge source is to be activated at specific times;
- how long the knowledge source may control the reasoning process; and
- the relative significance of conclusions drawn by the knowledge source.

The control mechanism also has a number of operating responsibilities and possibilities. Under normal operating conditions, the control mechanism:

- must put information into a format that the called knowledge source can handle (i.e., "data typing");
- may itself be partitioned into a hierarchical organization; and
- may allow the user to interrupt the process and change the direction of the problem-solving approach.

4. Applications of Blackboard Systems

When would we use a blackboard system? To answer that question, we must first understand some of the advantages of blackboard systems. These advantages are shown in Table 10.7.

Table 10.7. Advantages of a blackboard system.

- Integrates multiple sources of knowledge.
- Integrates multiple problem-solving approaches.
- Provides a means for opportunistic reasoning.
- Provides the user with a way to non-catastrophically interrupt the program.
- Can potentially implement true parallel processing and problem-solving.
- Provides for multiple levels of abstraction.
- Promotes flexibility of control of the problem-solving process.

Knowing these advantages of blackboard systems, we can then identify what types of problems are particularly well-suited for a blackboard implementation. Table 10.8 describes the characteristics of problems suitable for blackboard implementation.

Table 10.8. Problem characteristics suitable for a blackboard implementation.

- Problem requires the combination of different types of knowledge.
- Problem is of sufficiently high complexity such that opportunistic reasoning is helpful.
- Specific problem-solving sequence may not be fully anticipated *a priori*.
- Solution requires flexibility of control with the user in the problem-solving loop.
- Hierarchical knowledge representation or multiple levels of abstraction are required.
- Complex or combinatorially large search space exists.

One system that has been implemented using a blackboard is called DECADE (Design Expert for CAtalyst DEvelopment), by Banares-Alcantara et. al. (1987, 1988). DECADE chooses a catalyst for the Fisher-Tropsch process (hydrogenation of carbon monoxide). It uses three computer languages and knowledge representations in one package, as summarized in Table 10.9.

Table 10.9. Knowledge representations used in DECADE.

LANGUAGE	KNOWLEDGE REPRESENTATION
OPS5	Rule-based, forward chaining
SRL	Frame-based
LISP	Procedural (functions)

Another application of a blackboard system by Wahnschafft et. al. (1990) is SPLIT (Separation Process Layout by Invention and Testing), a flowsheet-development expert system. SPLIT uses a blackboard that integrates a flowsheet-development package (ASPEN Plus), with a numerical-optimization technique (mixed-integer nonlinear programming).

I. Exercises

10.1 What are some key characteristics of expert systems?

10.2 What kind of problem favors forward chaining over backward chaining? What kind of problem favors backward chaining?

10.3 What are the differences between a frame-based system and an object-oriented program?

10.4 What is a blackboard system? How does it differ from other traditional knowledge representation?

10.5 What problem characteristics make a blackboard system suitable? What are some chemical engineering problems that could be solved using a blackboard?

10.3 CHAPTER SUMMARY

We have introduced artificial intelligence (AI) and have discussed ways of representing knowledge in AI programs. These concepts are summarized below.

- There is no formal definition of AI. Rich and Knight (1991) define AI as "The study of how to make computers do things which, at the moment, people are better." Barr and Feigenbaum (1981, Vol. I, p.3) have proposed the following definition: "Artificial Intelligence is the part of computer science concerned with designing intelligent computer systems, that is, systems that exhibit characteristics we associate with intelligence in human behavior."

- One working definition of AI by Buchanan and Shortliffe (1983) is: "Artificial Intelligence is that branch of computer science dealing with symbolic, non-algorithmic methods of problem-solving."

- Uses of AI include natural language processing, computer vision, robotics, theorem proving, expert systems, learning, and artificial neural networks.

- One major challenge facing AI today is the meshing of quantitative analysis and qualitative reasoning. Another is that knowledge itself is voluminous, very difficult to characterize accurately, and constantly changing.

- *Expert systems* are one of the major applications in AI. Expert systems possess a knowledge base, an inference engine, and a user interface.

- Uses of expert systems include: interpretation, prediction, diagnosis, design, planning, monitoring, debugging, repair instruction, and control.

- In the process industries, expert-system applications include process design, process simulation and optimization, plant layout, decision support, training, process-fault diagnosis, process control, mechanical and structural design, planning, start-up and shutdown analysis, critiquing a design for flexibility, reliability, and safety, monitoring and assessing the origins of process trends, and even automatic programming. (See chapter 16 for a survey.)

- The most-used knowledge representations in expert systems are *logic-based systems*, *frame-based systems*, and *object-oriented programming*.

- Two of the most common logic-based systems are *rule-based* and *fuzzy-logic* systems.

- Another knowledge representation discussed is the *semantic network*. Semantic networks take a graphical approach to knowledge representation, and are a pre-cursor to frame-based systems and object-oriented programming.

- A frame is a grouping of information. Frames have the properties of inheritance, default reasoning, and attached procedures. Attached procedures include *if-added*, *if-needed*, and *if-removed* procedures.

- Some advantages of frame-based systems are that they: provide better structure and organization of the knowledge base, promote expectation-driven problem-solving, store large blocks of related information as a single entity, utilize automatic inheritance, allow for default reasoning, capture the hierarchal nature of knowledge through inheritance, and capture the interrelated nature of knowledge through multiple inheritance.

- An object-oriented program combines data and procedures together into a single, inseparable unit called an *object*. OOP is related to frame-based systems, with the exception that objects communicate with each other by sending messages, while frames do not.

- Some essential properties of OOP are *data abstraction*, *encapsulation*, *inheritance*, *message passing*, *polymorphism*, and *modularity*.

- Some needs in OOP include improvements in speed and efficiency, systematic object management, and a need for a clear programming methodology.

- A unique approach to AI is *blackboard systems*. A blackboard system combines multiple knowledge sources under one control structure. It uses opportunistic reasoning, and attempts to integrate the advantages of different problem-solving approaches.

A LOOK AHEAD

We now have a firm grasp of artificial intelligence (AI), and what AI encompasses. We have a feel for how to represent knowledge, and have identified some common knowledge-representation schemes used in AI today. In the next chapter, we shall investigate how Prolog fits into the realm of AI. We shall first focus on techniques of *search*, and discuss how to implement search procedures in Prolog. We then move to the topic of knowledge representation, and describe in detail how to implement some of the representations discussed in this chapter (in Prolog). We then introduce some additional problem-solving techniques in Prolog.

REFERENCES

Banares-Alcantara, R., A.W. Westerberg, E.I. Ko, and M.D. Rychener, "DECADE- A Hybrid Expert System for Catalyst Selection - I. Expert System Consideration," *Comput. Chem. Eng.*, 11, 265 (1987).

Banares-Alcantara, R., A.W. Westerberg, E.I. Ko, and M.D. Rychener, "DECADE- A Hybrid Expert System for Catalyst Selection - II. Final Architecture and Results," *Comput. Chem. Eng.*, 12, 923 (1988).

Barr, A. and E. Feigenbaum, *The Handbook of Artificial Intelligence*, Addison-Wesley, Reading, MA, Volumes I, II, and III, (1981).

Bratko, Ivan, *Prolog Programming for Artificial Intelligence*, second edition, pp. 331-358, Addison-Wesley, Reading, MA (1990).

Buchanan, B.G. and E.A. Feigenbaum, "DENDRAL and Meta-DENDRAL: Their Applications Dimension," *Artificial Intelligence*, 11, 5 (1978).

Buchanan, B.G. and E.H. Shortliffe, *Rule-Based Expert Systems*, Addison-Wesley, Reading, MA (1983).

Davis, J.F. and M.S. Gandikota, "Rule-Based Systems in Chemical Engineering," *in Artificial Intelligence in Process Systems Engineering, Volume II*," G. Stephanopoulos and J.F. Davis (eds.), CACHE Monograph Series, CACHE Corp., Austin, TX (1990).

Ernst, G.W. and A. Newell, *GPS: A Case Study in Generality and Problem Solving*, Academic Press, New York, NY (1969).

Hayes-Roth, F., Waterman, D., and Lenat, D., *Building Expert Systems*, Addison-Wesley, Reading, MA (1983).

King, J., "A Beginner's Guide to OOP: Object-Oriented Programming," *Proc. of the ASEE Annual Conference*, pp. 952-955, Toronto, Canada, May (1990).

Mueller, R.A. and R.L. Page, *Symbolic Computing with LISP and Prolog*, John Wiley & Sons, New York, NY (1988).

Newell, A. and H.A. Simon, "GPS, a Program that Simulates Human Thought," in *Computers and Thought*, E. Feigenbaum and J. Feldman (Eds.), McGraw-Hill, New York, NY (1963).

Quillian, R., "Semantic Memory," in *Semantic Information Processing*, M. Minsky (Ed.), MIT Press, Cambridge, MA (1968).

Ramamoorthy, C.V. and P.C. Sheu, "Object-Oriented Systems," *IEEE Expert*, pp. 9-15, Fall (1988).

Rich, E. and K. Knight, *Artificial Intelligence*, McGraw Hill, New York, NY (1991).

Rowe, Neil C., *Artificial Intelligence Through Prolog*, pp. 307-348, Prentice-Hall, Englewood Cliffs, NJ (1988).

Schalkoff, Robert J., *Artificial Intelligence: An Engineering Approach*, McGraw-Hill, New York, NY (1990).

Schnupp, P., C.T. Nguyen Huu and L.W. Bernhard, *Expert Systems Lab Course*, pp.121-146, Springer-Verlag, New York, NY (1989).

Stefik, M. and D.G. Bobrow "Object-Oriented Programming: Themes and Variations," *AI Magazine*, 6, 40 (1985).

Stephanopoulos, George, J. Johnston, T. Kriticos, R. Lakshmanan, M. Mavrovouniotis and C. Siletti, "DESIGN-KIT: An Object-Oriented Environment for Process Engineering," *Comput. Chem. Eng.*, 11, 655 (1987).

Stephanopoulos, George, "Brief Overview of AI and Its Role in Process

Engineering," in *Artificial Intelligence in Process Systems Engineering*, CACHE Monograph Series, Vol. I, Stephanopoulos, G. and J.F. Davis (Eds.), CACHE Corp., Austin, Texas (1990).

Taylor, W.A., *What Every Engineer Should Know About AI*, MIT Press, Cambridge, MA (1988).

Ungar, L.H. and V. Venkatasubramanian, "Knowledge Representation," in *Artificial Intelligence in Process Systems Engineering*, CACHE Monograph Series, Vol. III, Stephanopoulos, G. and J.F. Davis (Eds.), CACHE Corp., Austin Texas (1990).

Venkatasubramanian, V. and S.H. Rich, "An Object-Oriented Two-Tier Architecture for Integrating Compiled and Deep-Level Knowledge for Process Diagnosis," *Comput. Chem Eng.*, 12 903 (1988).

Waterman, D., *A Guide to Expert Systems*, Addison-Wesley, Reading, MA (1986).

11

PROLOG IN ARTIFICIAL INTELLIGENCE

In this chapter, we introduce Prolog applications to artificial intelligence (AI). We first describe aspects of *search*. In AI, search is the process of systematically analyzing and processing the knowledge. After discussing systematic search strategies, we move on to techniques of knowledge representation in artificial intelligence. We introduce Prolog techniques of knowledge representation, particularly in the areas of rule-based, frame-based, and object-oriented systems. Finally, we discuss other problem-solving strategies, such as AND-OR strategies, constraint satisfaction, generate-and-test, and means-end analysis.

11.1 SEARCH STRATEGIES IN PROLOG

A. General Definition and Description of Search Strategies

Search is the process of applying inferences and moving through the *state space* until we find an acceptable answer. The state space is the collection of all possible situations or configurations. In most engineering problems, the state space is very large. For example, in an engineering-design expert system, the state space for potential design solutions can include as many as 10^{250} different configurations. We need to find the configuration that is economically optimal.

Consider Figure 11.1. Let us apply evolutionary design techniques; we have an initial design **A** and we want to consider other economically more attractive options.

In Figure 11.1, we are at the design plan **A** and want to get to plan **H**. In this diagram, each letter represents a different design, and is called a *node*. Node **A** is the *starting node*, and node **H** is the *goal node*. We could travel arbitrarily through the state space, sometimes looping (e.g., taking the route A → B → F → C → A). Eventually, we may finally get from **A** to **H**. Fortunately, our state space is small, and choosing our path arbitrarily may not create much trouble. Choosing a path arbitrarily like

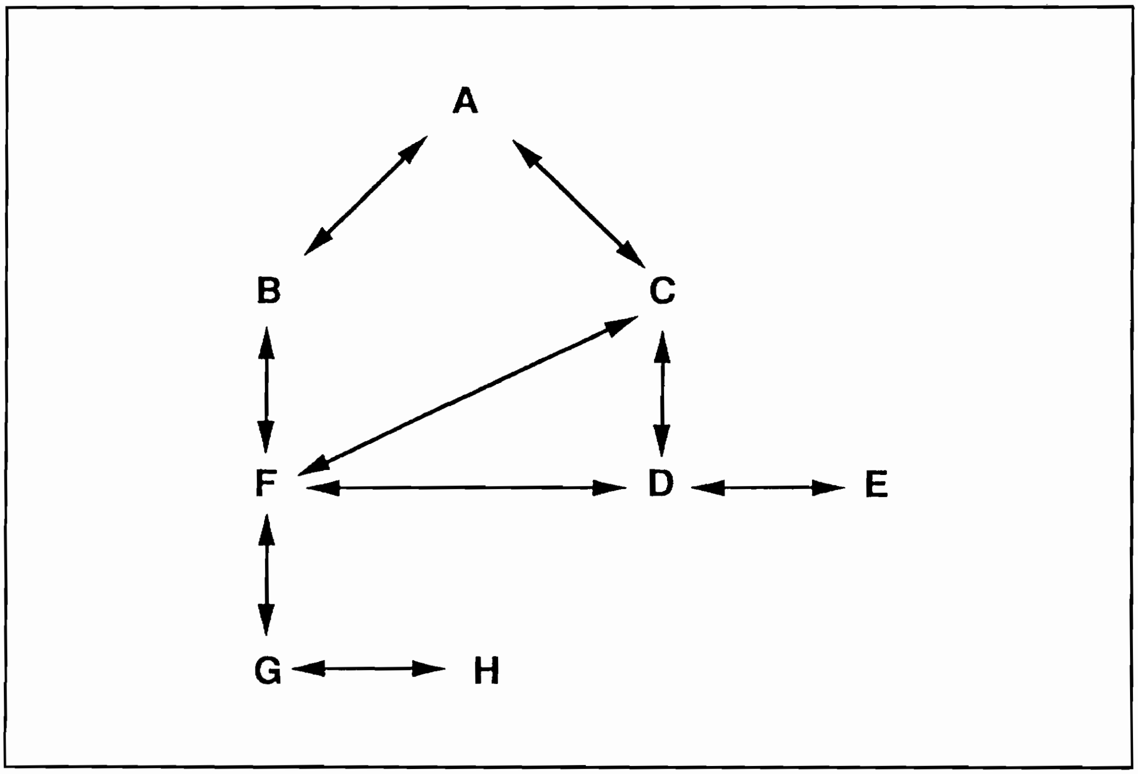


Figure 11.1. The concept of state space.

this is called *blind search*.

As mentioned, however, most AI problems in science and engineering have huge state spaces. No search trees representing the state space can be drawn. Some state spaces are so large and complex that fully assessing their size is practically impossible. Even for very simple problems (such as stacking four or five blocks in a desired fashion), the state space can have as many as 10^{150} to 10^{250} nodes.

The purpose of search is to find the goal node *efficiently* and *systematically*. In Prolog, we apply logical inferences to develop a search path or inference chain that eventually leads us to the final goal. A number of systematic methods of searching the state space exist. These search methods are ***domain-independent***; that is, they employ tools that can be applied universally to all AI problems regardless of the specific task or domain at hand. The following sections discuss the implementation of these tools in Prolog.

B. Depth-first search

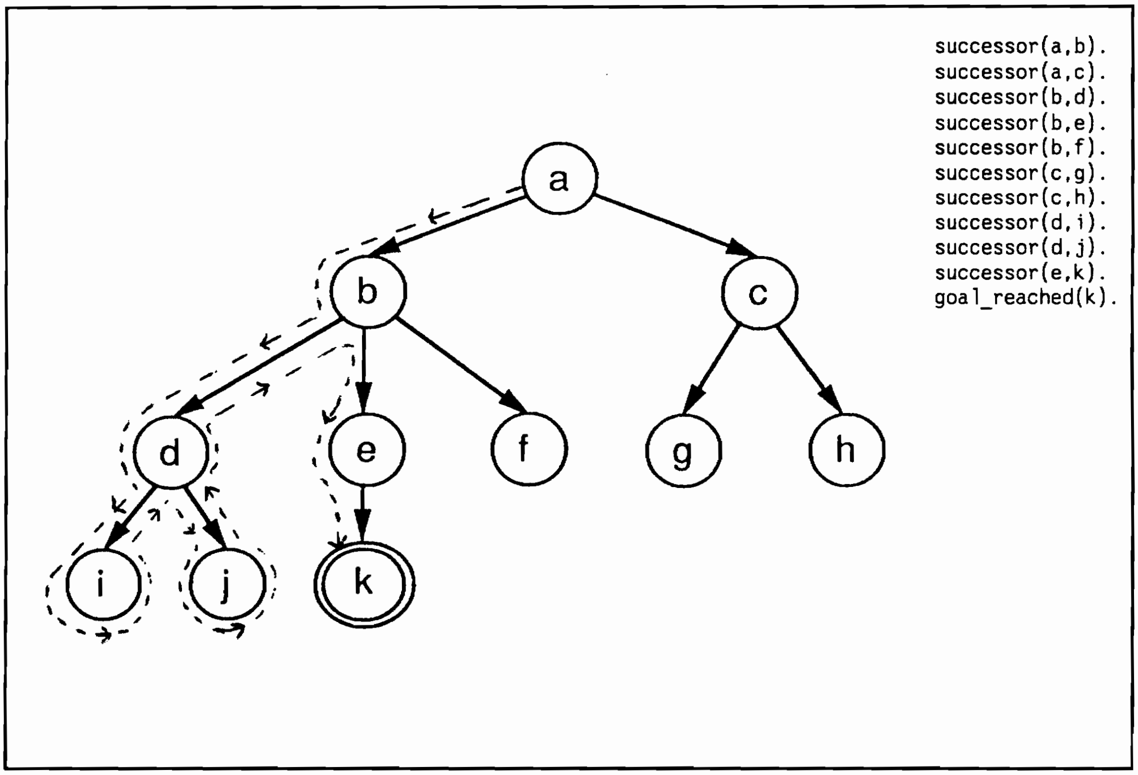
1. Understanding Depth-First Search

A depth-first search favors rapid penetration into the state space. Let us consider Figure 11.2. We have a diagram of a top-down state space, consisting of nodes **a** through **k**. These nodes could represent a number of different items, such as pumping stations, design configurations, and even competing schedule plans. Regardless of what the nodes represent, we can only move from one node to the next if the subsequent node is a *successor* to the preceding node. We control this restriction with the **successor** fact:

```
successor(a,b).
```

This statement says, "The successor to node **a** is node **b**." When we are at a particular node, the **successor** clause fully specifies what nodes we may move to.

When searching top-down, as shown in Figure 11.2, a depth-first search always favors the "left-hand side" of the tree. Our starting node is a and our goal node is **k**. A depth-first search expands to the deepest-node first. One node is "deeper" than another if it is farther away from the starting node. Thus nodes **d** and **e** are deeper than **b** and **c**.



```
successor(a,b).
successor(a,c).
successor(b,d).
successor(b,e).
successor(b,f).
successor(c,g).
successor(c,h).
successor(d,i).
successor(d,j).
successor(e,k).
goal_reached(k).
```

Figure 11.2. A depth-first search.

A depth-first search always tries a node's successor as its first choice. It will only try an alternate path when a node fails and has no remaining successor. In Figure 11.2, the search keeps going deeper and deeper until it hits node i, which fails. The system then backtracks to node **d**. This node still has a successor node, **j**, so the system tries that.

Node **j** fails too. Since node **d** has no other successor, the system backtracks to node **b**. Going down the search tree along node **b**, the system again favors successors. From node **b**, the system expands first to node **e**, and then node **k**. At node **k**, the system has successfully achieved its goal, and stops executing.

2. Depth-First Search in Prolog

Because Prolog's built-in inference and unification (i.e., matching) mechanism naturally operates on a depth-first search method, we can easily implement a depth-first search. For example, let us develop a clause **depth_first(N,L)**, where N is the current node and L is the list of nodes required to travel from the starting node to the goal node. The depth-first procedure, with comments, is as follows:

```
/***********************************************************************
      The first argument in depth_first is the current node we are
      at. The second argument is the tail of the pathway, i.e., the
      list of nodes the system has expanded to. If the search has
      reached the goal node, Prolog will instantiate the tail of the
      path list to the goal node. Prolog will then stop, since the
      search has found the desired location.
***********************************************************************/
```

```
      depth_first(Node,[Node]):-
                  goal_reached(Node).

/***********************************************************************
      If the previous rule fails, we have not yet reached the goal
      node. Therefore, let us expand to a successor and add the node
      we are expanding from onto the accumulating path list.
***********************************************************************/

      depth_first(Node,[Node|ListofNodes]):-
                  successor(Node,NewNode),
                  depth_first(NewNode,ListofNodes).
```

If we combine this with the relations shown in Figure 11.2, i.e.

```
      successor(a,b).
      successor(a,c).
      successor(b,d).
      successor(b,e).
      successor(b,f).
      successor(c,g).
      successor(c,h).
      successor(d,i).
      successor(d,j).
```

```
successor(e,k).

goal_reached(k).
```

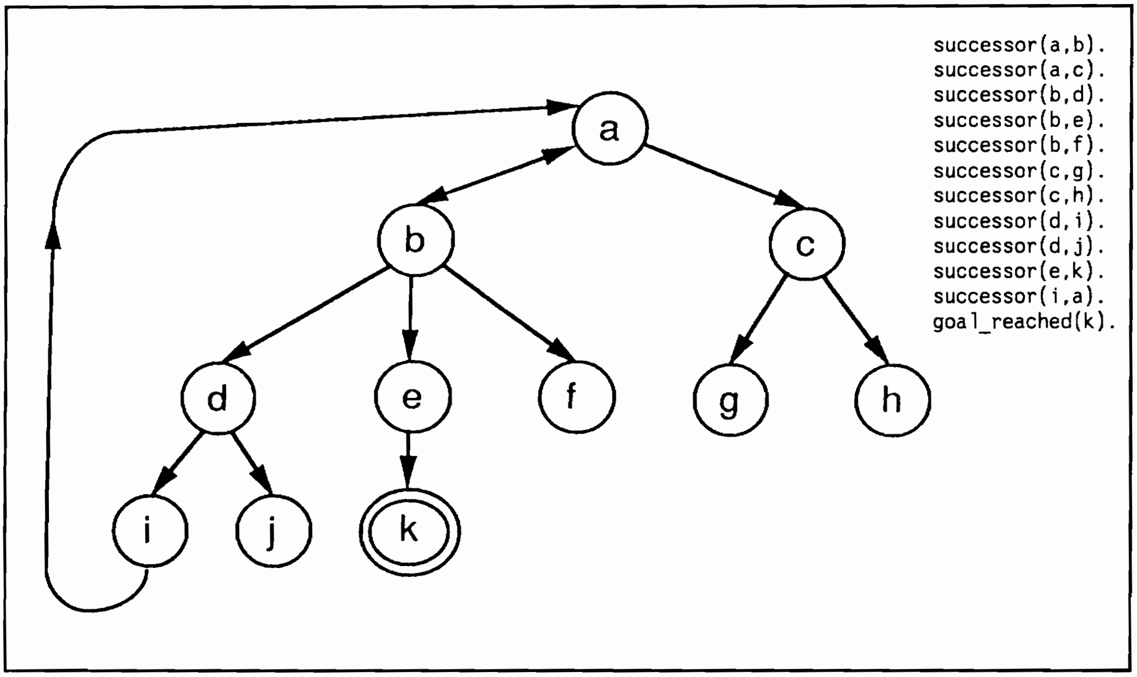
then the following dialogue results:

```
?- depth_first(a,Path).
Path = [a,b,e,k]
?- depth_first(b,Path).
Path = [b,e,k]
?- depth_first(c,Path).
no
```

3. A Problem with Depth-First Search: Circular State Space

Depth-first search strategies work well with many problems. They drive deep into the search space, and thus can find solutions quickly. However, they can run into trouble. One problem type that poses difficulties with depth-first search is a *circular state space.*

Consider the diagram and Prolog relations shown in Figure 11.3. Node a is the starting node and **k** is the goal node. With this state space, we have a "back door" where we can expand from node i to node **a**. Following a depth-first search, Prolog drives down from node a through node **b** and **d**, until it gets to node i. A depth-first search always favors a node's



```
successor(a,b).
successor(a,c).
successor(b,d).
successor(b,e).
successor(b,f).
successor(c,g).
successor(c,h).
successor(d,i).
successor(d,j).
successor(e,k).
successor(i,a).
goal_reached(k).
```

Figure 11.3. A circular state space.

successor. Therefore, Prolog expands back to node a, since it is one step "farther" into the tree. The result? No progress occurs, and the program never gets to goal node k. Unknowingly, and seemingly naively, Prolog goes through the *a-b-d-i* loop forever.

This problem can be prevented in depth-first searches by putting in a *loop-detection mechanism* that assesses whether or not the search has reached that node before. The entire program is:

```
/*************************************************************************

     The first argument in depth_first_help is the current node
     that we are at. The second argument is the accumulator, i.e.,
     the accumulated list of nodes that we have expanded to. The
     third argument is the final path answer. To run the
     depth_first procedure, we call depth_first_help and initialize
     the accumulator with the starting node.
***********************************************************************/

     depth_first(Node,Ans):-
               depth_first_help(Node,[Node],Ans).

/***********************************************************************
     If the goal node has been reached, then the path answer list
     is the accumulated list.
***********************************************************************/

     depth_first_help(Node,Ans,Ans):-
               goal_reached(Node).

/***********************************************************************
     If we get to this point, we know that we have not yet reached
     the goal node, since the previous rule failed. We need to
     expand immediately to a successor. If the successor is not a
     member of the accumulator, then it is a new node. We add this
     new node to the list of nodes that we have expanded to, and
     call the depth_first_help procedure again.
          If the successor is a member of the accumulator, we have
     already been to that node and the not(member(NewNode,ListofNodes))
     goal fails. We then backtrack to alternate successors. If no
     alternate successors exist, then that branch of the tree is a dead
     end and we backtrack to previous depth_first_help clauses.
************************************************************************/

     depth_first_help(Node,ListofNodes,Ans):-
                 successor(Node,NewNode),
                 not(member(NewNode,ListofNodes)),
                 depth_first_help(NewNode,[NewNode|ListofNodes],Ans).

     successor(a,b).
     successor(a,c).
     successor(b,d).
     successor(b,e).
     successor(b,f).
     successor(c,g).
     successor(c,h).
     successor(d,i).
     successor(d,j).
     successor(e,k).
     successor(i,a).

     goal_reached(k).
```

```
member(X,[X|_]).
member(X,[_|T]):-
        member(X,T).
```

In the above program, the **not** and **member** relations prevent an infinite loop if a circular state space exists. Note that here, we add the new node onto the accumulator at the *end* of the clause with the *recursive call*:

```
. . ., depth_first_help( NewNode,[NewNode|ListofNodes],Ans).
```

In the program without loop detection, we add the new node onto the accumulator at the *beginning* of the clause with the *initial call*:

```
depth_first(Node,[Node|ListofNodes]):- . . .
```

The result? Our loop-detection program works, but we see an interesting twist. We ask the question:

```
?- depth_first(a,Path).
```

Instead of going into an *a-b-d-i* infinite loop, Prolog gives the answer:

Path = [k,e,b,a]

The "twist" is that our pathway is reversed. Instead of giving ***Path* = *[a,b,e,k]*** like we saw in the program without loop detection, we get ***Path* = *[k,e,b,a]***. This is not a major problem, but can be an inconvenience. There is a way to write the **depth_first** procedure with loop detection such that the answer for **Path** is not reversed. That procedure will be investigated in the problems in section 11.1F.

4. Limitations of Depth-First Search

A depth-first search can solve a problem quickly and efficiently. Because the search favors the "left" side of the tree, however, it arbitrarily imposes a preference on what node to try next. For this reason, a depth-first search may not expand to a favorable portion of the tree, and therefore can give a non-optimal result.

To demonstrate, let us consider the diagram shown in Figure 11.4, where **a** is the starting node and **k** is the goal node. The optimal path is simply **[a,k]**. But when we implement a depth-first search, we get the following result:

```
?- depth_first(a,Path).
Path = [a,b,e,i,k]
```

Prolog's answer is clearly not the optimal path. The result illustrates an inherent weakness of the depth-first search: it does not guarantee the

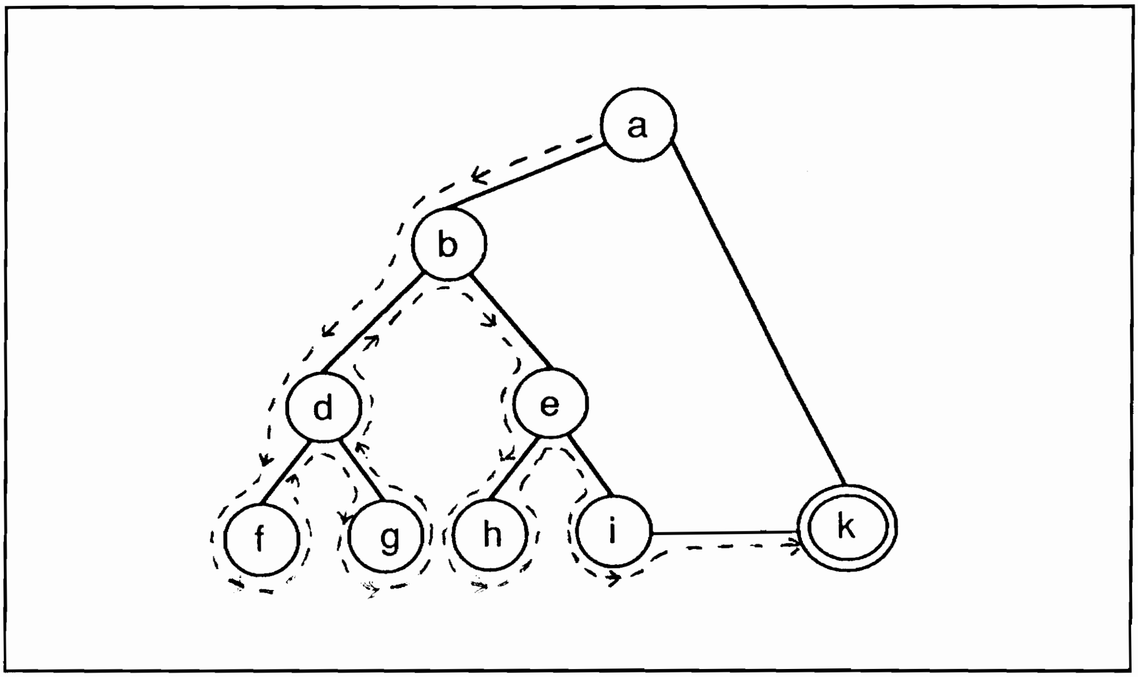


Figure 11.4. An inefficient depth-first search.

optimal result.

5. Summary of Depth-First Search

To summarize, a depth-first search:

- favors expansion to the deepest node first;
- is *efficient*, has the ability to arrive at the answer quickly; and
- is *non-optimal*, gives no guarantee that the result is optimal.

We discuss how to overcome the non-optimality deficiency in the following section.

C. Breadth-First Search

1. Description of Breadth-First Search

A breadth-first search chooses all nodes closest to the starting node before going any deeper. Instead of favoring the left side of the tree, a breadth-first search expands equally to all immediate successor nodes before going deeper. As shown in Figure 11.5, what results is a broader path than seen in depth-first search.

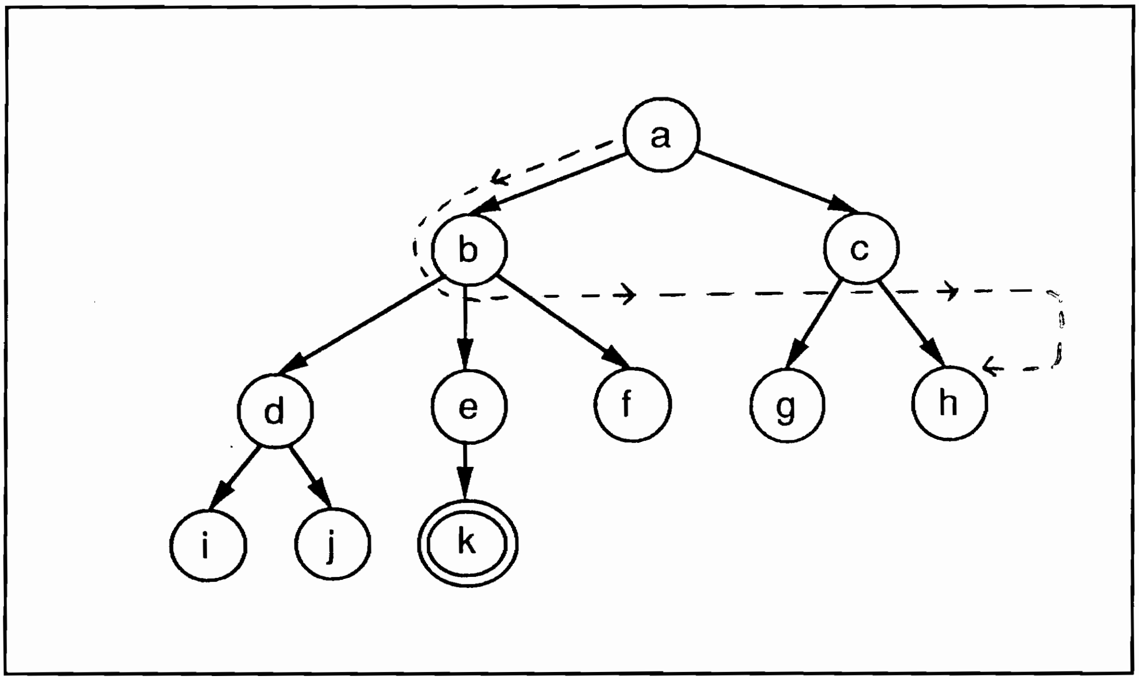


Figure 11.5. A breadth-first search.

2. Breadth-First Search in Prolog

Implementing a breadth-first search in Prolog is more complex than implementing a depth-first search, because we need to continuously build and maintain a *set* of candidate nodes. This *candidate set* is a list of lists, representing the set of potential paths from the starting node to the goal node. Recall that in a depth-first search, we are only concerned about the current node we are at, not a set of candidates that may reach the goal.

When Prolog runs a breadth-first search, the candidate set of potential paths grows. When we finally hit the goal node, we choose the optimal candidate set. This technique ensures the optimal result.

We initiate the search with a single-element candidate set, the starting node:

```
[ [StartNode] ]
```

A breadth-first search continuously builds and maintains the candidate set, expanding as close to **StartNode** as possible. Therefore, in Figure 11.5, **StartNode** is

```
[ [a] ]
```

and the first expansion goes to

```
[ [b,a], [c,a] ]
```

Now we analyze the first candidate path **[b,a]** from the set. We generate extensions to this path in a breadth-first mode; that is, we expand to all adjacent nodes before going deeper. From node **b**, we expand to nodes **d**, **e**, and **f**. The result is:

[[d,b,a], [e,b,a], [f,b,a]]

We now append these new paths onto the current set to give:

[[c,a], [d,b,a], [e,b,a], [f,b,a]].

Since the goal node **k** has not been hit, we continue. To expand in a breadth-first fashion, we return to **[c,a]** from the set and generate its new paths:

[[g,c,a], [h,c,a]]

We append these onto the end of the current set to give:

[[d,b,a], [e,b,a], [f,b,a] [g,c,a], [h,c,a]]

With this list, we have completed one "layer" of expansion from node **a**; we have expanded to nodes **b** and **c** and have identified their successors. We have not yet reached goal node **k**, however. Therefore, we must keep expanding in a breadth-first mode. We move down to the next "layer," nodes **d**, **e**, and **f**, to give:

```
      [ [i,d,b,a], [j,d,b,a], [k,e,b,a], [f,b,a], [g,c,a], [h,c,a] ]
```

The search detects goal node **k** in **[k,e,b,a]**. Prolog reports this as the answer, and the breadth-first search is complete.

Bratko (1990) has implemented an efficient breadth-first search in Prolog, and his implementation will be used here. He gives the following code:

```
      breadth_first(StartNode,ListofNodes):-
            breadth_first_help([ [StartNode] ],ListofNodes).
      breadth_first_help([ [Node|Path]|_ ], [Node|Path]):-
            goal_reached(Node).
      breadth_first_help([ [Node|Path]|Paths], ListofNodes):-
            bagof([NewNode,Node|Path],
                  (successor(Node,NewNode),
                   not(member(NewNode,[Node|Path]))), NewPaths),
% Newpaths = acyclic extensions of [Node|Path]
            append(Paths,NewPaths,Paths1),
            breadth_first_help(Paths1,ListofNodes);
            breadth_first_help(Paths,ListofNodes). % Use this if Node has
                                                   % no successors
```

One advantage of a breadth-first search is that it guarantees that we find the optimal (shortest) path between the starting node and the goal node. Look again at the state space in Figure 11.4, where **a** is the starting node and **k** is the goal node. We get the following dialogue:

```
?- breadth_first(a,Path).
Path = [k,a]
```

The breadth-first search gives the optimal answer. The result is clearly a better solution than the depth-first result of **[a,b,e,i,k]**.

3. Limitations of Breadth-First Search

Although breadth-first searches give the optimal answer, they have an associated cost. Breadth-first searches take a long time to execute. We may be guaranteed the optimal path, but we also may wait a very long time for it. Breadth-first searches are *combinatorially explosive or prohibitive.*

As a breadth-first search progresses, the candidate set grows exponentially. The search in Figure 11.5 starts out with the candidate set of **[[a]]**. After expanding to one layer, the set grows to **[[d,b,a], [e,b,a], [f,b,a] [g,c,a], [h,c,a]]**. At this rate, if we expand to seven layers, the result is a candidate set with over 800,000 elements. Clearly, the search gets combinatorially explosive, and the computer soon runs out of memory.

In a complex scientific or engineering expert system with a very large state space, we cannot use a breadth-first search. A computer will run out of memory before it completes the search. For instance, with a state space of 10^{140} nodes, the combinatorial complexity is so high that no computer can solve the problem in a breadth-first manner. We need a more efficient technique.

4. Summary of Breadth-First Search

To summarize, a breath-first search:

- favors expansion to all successor nodes equally before going any deeper;
- is *inefficient*, usually unable to arrive at the answer quickly; and
- is *optimal*, guarantees that the result is optimal.

We see two extremes: depth-first search gives us efficiency, but is non-optimal. Breadth-first search gives us the optimal result, but is inefficient. The next section discusses a way to mesh these two together in an attempt to attain both an efficient and near-optimal search.

D. Best-First Search

A best-first search attempts to achieve both *efficiency* and *optimality*. It is a refinement of a breadth-first search. A breadth-first search:

- maintains a set of candidate paths, and
- expands to all nodes closest to the starting node before going deeper.

By contrast, a best-first search:

- also maintains a set of candidate paths;
- but computes a *heuristic estimate* for each candidate; and
- expands to the best candidate according to the estimate.

A best-first search is an attempt to:

(1) *get the optimal solution*- by evaluating all nodes in the next layer; and

(2) *be efficient and avoid combinatorial explosion*- by choosing only that node in the next layer that is heuristically the best candidate.

We can implement the heuristic estimation in many ways. The method chosen depends on the problem. We discuss briefly here the principle of heuristic estimation that apply to virtually all problems. Depending on the specific problem, however, this principle may change somewhat.

Suppose we want to get from starting node **s** to the goal node **e** as shown in Figure 11.6. Let us assume that we are on our path and are currently at node **n**. We designate the heuristic estimate, *f(n)*, as a function that estimates the "total difficulty" of node **n**. We represent *f(n)* as

$$f(n) = g(n) + h(n)$$

Here, *g(n)* is the difficulty (a cost function) from starting node s to node n, and *h(n)* is the difficulty (another cost function) from node n to goal node e.

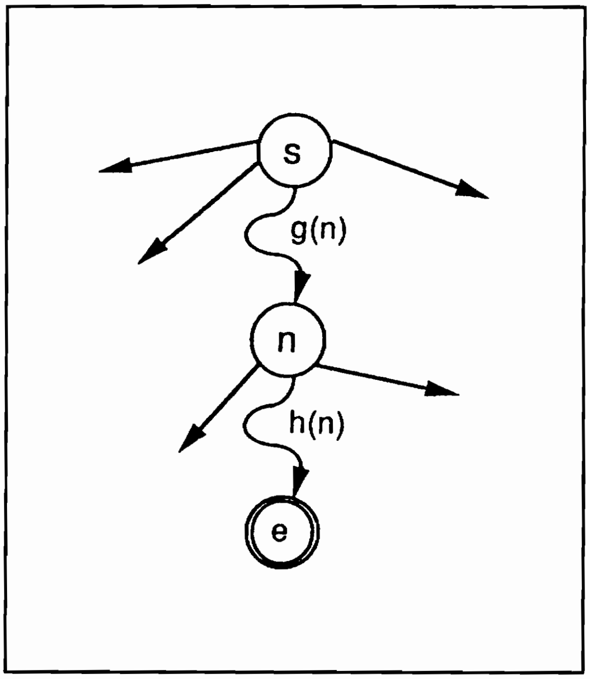


Figure 11.6. A best-first search.

Since we have already travelled the path from s to n, we know what concrete steps we have taken. Therefore, calculating the cost function *g(n)* is not difficult, assuming that we know the explicit functional form of *g(n)*. We can estimate the value based on the nodes that we have already visited.

Calculating *h(n)*, however, is much more difficult. We do not know what path we will take from n to e. Therefore, evaluating *h(n)* is often a heuristic guess. The reliability and efficiency of the search hinges on the quality of the heuristic guess. Since these guesses are frequently arbitrary, the search may not be reliable and consistent.

Computer scientists have spent a tremendous amount of effort in finding ways to calculate *h(n)* and implement best-first searches. Some have used statistics; others have used inexact reasoning theory. But in complex scientific and engineering AI problems, few of these approaches have yielded breakthroughs. Estimates of *h(n)* and the accompanying best-first search can be just as arbitrary as a depth-first search.

When developing an AI application, we must be cautious about implementing complex search mechanisms involving numerical heuristic estimates or statistics. They can require much time and yield little gain. In the first edition of *Artificial Intelligence*, Elaine Rich (1983) gives two good guidelines for the use of probabilistic reasoning:

- *Avoid statistical representations when a better analysis of the problem would make them unnecessary or less important.*

- *If statistical reasoning is necessary to handle real-world randomness or genuine lack of complete information, then perform the statistical manipulations in small increments that correspond to logical steps in the reasoning rather than in one large operation* *that fails to reflect the structure of the problem.*

A notable expert system that successfully uses statistical reasoning is MYCIN. This system diagnoses infectious blood diseases and recommends treatment. In this case, statistical reasoning is essential, since both microbiology and medical diagnostics involve uncertainty and operate on incomplete information.

E. Summary

Table 11.1 contrasts the pros and cons of depth-first, breadth-first, and best-first searches.

Table 11.1. Depth-first, breadth-first, and best-first searches.

SEARCH	EFFICIENCY	OPTIMALITY	COMMENTS
depth-first	very efficient	solution may be non-optimal	Arbitrary. Expands to next lower layer; results are usually non-optimal. Also, difficulties arise with a circular state space.
breadth-first	inefficient	optimal solution is guaranteed	Expands to all adjacent nodes before going any deeper; optimality is guaranteed. Suffers from combinatorial explosion, even in moderately-sized problems.

best-first	efficient	near-optimal solution	Attempts to deliver both efficient and near-optimal results. Performance hinges on the quality of the heuristic estimate, which can be arbitrary.

To summarize, all artificial intelligence applications require search. A good search technique is essential to the performance of an expert system. However, in most scientific and engineering applications of expert systems, *our focus should be on the development of good knowledge representation over complex search techniques.* Time is better spent attempting to understand the problem at hand rather than developing sophisticated heuristic searches.

F. Practice Problems

11.1.1 In what kind of problem space is a depth-first search better than a breadth-first search?

11.1.2 Under what circumstances would a best-first search be worse than a breadth-first search?

11.1.3 Write the Prolog code **depth_first_limit(Node,Answer,DepthLimit)**. This relation performs a depth-first search, starting at node **Node**. The relation instantiates **Answer** to a list of nodes representing the path from the starting node to the goal node. Importantly, the **depth_first_limit** relation differs from a standard depth-first search in that it will go no deeper in the search than the specified **DepthLimit**. If no answer is found within this maximum depth, the relation fails.

11.1.4 Develop the relation **depth_first_lc(Node,Answer,DepthLimit)**. This relation behaves exactly like **depth_first_limit** in problem 11.1.3, except that it *also* prevents looping through a circular state space.

11.1.5 The **depth_first** search procedure with loop detection in section 11.1B3 inconveniently gave the answer **Path** as the *reverse* of the actual path we take to reach the goal node. For instance, if start at node a and travel to node **k**, our path is **a** → **b** → **e** → **k**. The procedure in section 11.1B3 gives the answer ***Path*** = ***[k,e,b,a]***. Correct the procedure such that the answer **Path** = **[a,b,e,k]** results. Do not use the **reverse** relation.

11.2 KNOWLEDGE REPRESENTATION IN PROLOG

A. Rule-Based Knowledge

Rule-based knowledge is the most common type of knowledge representation used in Prolog-based expert systems and AI programs. A rule-based approach must include: 1) *rule-based knowledge*, i.e., facts, rules, and heuristics specific to the problem and written in the form of Prolog facts and rules; and 2) an *inference mechanism* that uses the knowledge base to create an inference chain, ultimately leading to conclusions and desired problem-solving. Typically, this inference mechanism is either backward chaining or forward chaining.

Rule-based systems are a "natural" in Prolog. We can represent rule-based knowledge by writing Prolog rules, and utilize Prolog's built-in backward-chaining, depth-first inference mechanism. For example, if we are designing a fault-diagnosis expert system, we may write rules to determine why the reactor conversion is dropping off for an exothermic reaction in a fixed-bed reactor. Let us consider the following problems and their evident symptoms:

- *Poor regeneration*- evidenced by carbon residue on the catalyst.
- *Poisoned catalyst*- evidenced by two potential sets of symptoms:
  (1) heavy metals appear on the catalyst under normal operating flow rates; or

(2) halogenated organics appear on the catalyst, also under normal operating flow rates.

- *Channeling*- evidenced by a low pressure drop through the reactor under normal operating flow rates.
- *Plugging*- evidenced by both a high pressure drop and low flow rate through the reactor, under a normal feed pressure.

This knowledge can be cast in the form of Prolog rules:

```
poor_regeneration:-
      on_catalyst(carbon_residue).
poisoned_catalyst:-
      on_catalyst(heavy_metals),
      flow_rate(normal).
poisoned_catalyst:-
      on_catalyst(halogenated_organics),
      flow_rate(normal).
channeling:-
      pressure_drop(low),
      flow_rate(normal).
plugging:-
      pressure_drop(high),
      flow_rate(low),
      feed_pressure(normal).
```

Given these facts in the database:

```
on_catalyst(carbon_residue).
on_catalyst(heavy_metals).
flow_rate(normal).
pressure_drop(high).
feed_pressure(high).
```

We ask:

```
?- poisoned_catalyst.
```

and Prolog responds ***yes***.

We ask:

```
?- channeling.
```

and Prolog responds ***no***. We use rules written in Prolog and rely on Prolog's backward chaining, depth-first inference strategy to answer our questions.

Backward chaining starts with a hypothesis (a goal) and attempts to prove that the hypothesis is true. It is a top-down reasoning strategy that proposes a conclusion, and then analyzes the data to validate the proposal's truth. Backward chaining is *goal-driven*, and the rules can be viewed as "THEN-IF"-type rules. Prolog's built-in inference mechanism utilizes backward chaining.

Forward chaining, on the other hand, starts with the data and attempts to determine which hypotheses are true. It is a bottom-up reasoning strategy that collects all of the data, and then "fires" certain rules if specific data-conditions are met. As the program executes and rules fire, more and more conclusions (facts) are derived. In essence, forward chaining derives new facts from a collection of data. Thus, forward chaining is *data-driven*, and the rules can be viewed as "IF-THEN"-type rules.

Implementing a backward-chaining program is almost trivial in Prolog, since Prolog naturally relies on backward chaining. We can, however, also implement forward-chaining program in Prolog. Rule-based programs using forward chaining are only slightly more complex than the backward-chaining programs. The reader may refer to Bratko (1990, pp.337-342) for a good forward-chaining rule interpreter in Prolog; we use a modified version of that interpreter here.

To do forward-chaining in Prolog, we rely upon the rules:

```
if Condition then Conclusion.
if Conditionl and Condition2 then Conclusion.
if Conditionl or Condition2 then Conclusion.
```

The forward-chaining program is shown in Figure 11.7. We see that we may have a single **Condition** to generate a conclusion, or we may have **Condition1** and **Condition2** linked by an AND or an OR. The interpreter starts with what is already known (stated in the **fact(P)** relation) and derives all new conclusions that can be derived from the known facts. These new conclusions are then placed into the database using the **assert** procedure. Once the fact is asserted into the database, it can be used by the forward-chaining program to derive new conclusions. We again analyze why the conversion is falling off in our fixed-bed reactor, as described earlier in this section. We use the same rules as those in the backward-chaining example, but write the rules in the **if Condition then Conclusion** format.

We may now watch the forward-chaining program work. With the program as written in Figure 11.7, we get the following dialogue:

```
?- forward.

Derived: poor_regeneration
Derived: poisoned_catalyst
No more facts can be derived
yes
```

```
:- op( 800, fx, if).     % prefix operator with precedence = 800
:- op( 700, xfx, then).  % infix operator with precedence = 700
:- op( 300, xfy, or).    % infix operator with precedence = 300
:- op( 200, xfy, and).   % infix operator with precedence = 200


forward:-
      new_derived_fact(P),
      !,                                   % A new fact has been derived
      write('Derived: '), write(P), nl,    % Write what this new fact is
      assert(fact(P)),
      forward ;                          % Continue with forward chaining
      write('No more facts can be derived').

new_derived_fact(Conclusion):-
      if Condition then Conclusion,  % A potential rule exists
      not(fact(Conclusion)),        % Rule's conclusion is not yet a fact
      composed_fact(Condition).      % Is the condition true?

new_derived_fact(Conclusion):-
      if Condition1 and Condition2 then Conclusion,
      not(fact(Conclusion)),
      composed_fact(Condition1 and Condition2).

new_derived_fact(Conclusion):-
      if Condition1 or Condition2 then Conclusion,
      not(fact(Conclusion)),
      composed_fact(Condition1 or Condition2).

composed_fact(Condition):-
        call(Condition).                             % A simple fact

composed_fact(Condition1 and Condition2):-
      composed_fact(Condition1),
      composed_fact(Condition2).           % Both conjuncts are true

composed_fact(Condition1 or Condition2):-
      composed_fact(Condition1)
      ;
      composed_fact(Condition2).           % One disjunct is true

if
   fact(on_catalyst(carbon_residue))
then
   poor_regeneration.

if
```

```
   fact(on_catalyst(heavy_metals))
and
   fact(flow_rate(normal))
then
   poisoned_catalyst.

if
     fact(on_catalyst(halogenated_organics))
and
     fact(flow_rate(normal))
then
     poisoned_catalyst.

if
     fact(pressure_drop(low))
and
     fact(flow_rate(normal))
then
     channeling.

if
     fact(pressure_drop(high))
and
     fact(flow_rate(low))
and
     fact(feed_pressure(normal))
then
     plugging.

fact(on_catalyst(carbon_residue)).
fact(on_catalyst(heavy_metals)).
fact(flow_rate(normal)).
fact(pressure_drop(high)).
fact(feed_pressure(high)).
```

Figure 11.7. A forward-chaining interpreter in Prolog.

B. Semantic-Network Knowledge

Semantic networks were introduced in section 10.2E. We now discuss how to implement the principle of *inheritance* using Prolog. Lets us consider the semantic network shown in Figure 11.8. This network is for heat-transfer equipment. Because H-1 is a direct-fired heater, then, by inheritance, we conclude that it runs on LPG (liquified petroleum gas), operates continuously, is made of steel, operates 7000 hours per year, and is used for energy transfer.

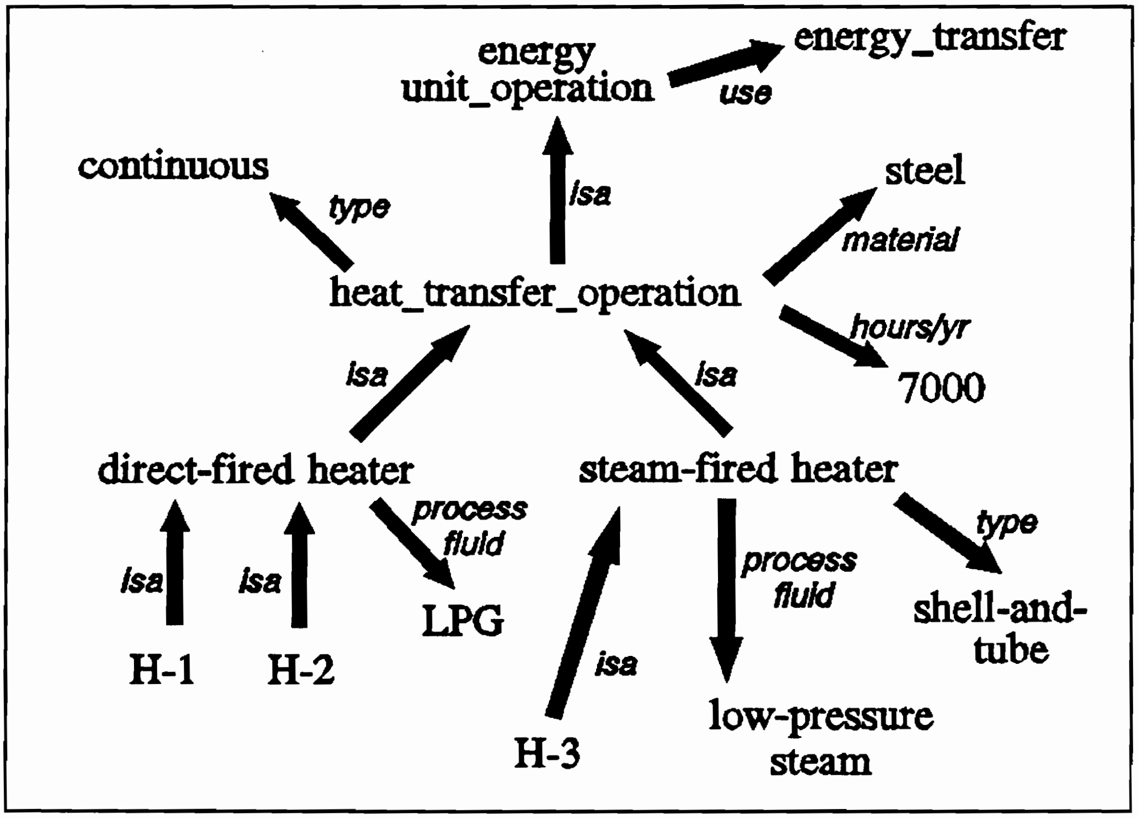


Figure 11.8. A semantic network for heat-transfer equipment.

We implement the principle of inheritance. Inheritance works as follows:

(1) if the information is available within the current node, do not inherit, and let the local value supersede the default value.

(2) If the information is not available within the current node,

then implement inheritance automatically, and obtain the default value from a higher-level node connected by the isa link.

Table 11.2 shows the Prolog code for the semantic network, along with the procedures for achieving automatic inheritance.

Table 11.2. Automatic inheritance in a semantic network.

```
/* Fact is not a variable and is available within the current node.    *
 * Therefore, do not inherit, since local values supersede all others  */
     fact(Fact):-
          nonvar(Fact),
          call(Fact).
/* Fact is a variable and is not available within the current node.    *
 * Therefore, perform inheritance, and climb the isa hierarchy.        */
     fact(Fact):-
          Fact =.. [Relation, Argument1, Argument2],
          isa(Argument1, Default_Argument),
          HigherFact =.. [Relation, Default_Argument, Argument2],
          fact(HigherFact).

     isa(heat_transfer_operation, energy_unit_operation).
     isa(direct_fired_heater,heat_transfer_operation).
     isa(steam_fired_heater,heat_transfer_operation).
     isa(h_1,direct_fired_heater).
     isa(h_2,direct_fired_heater).
     isa(h_3,steam_fired_heater).
     type(heat_transfer_operation,continuous).
     type(steam_fired_heater,shell_and_tube).
     process_fluid(steam_fired_heater,low_pressure_steam).
     process_fluid(direct_fired_heater,lpg).
     material(heat_transfer_operation,steel).
     hours_per_year(heat_transfer_operation, 7000).
     use(energy_unit_operation,energy_transfer).
```

We now ask the program some questions and utilize automatic inheritance. We ask, what is the use of equipment **h_1**?

```
?- fact(use(h_1,X)).
X = energy_transfer;
no
```

There is only one solution to this question. Since **h_1** is ultimately an **energy_unit_operation**, and the use of **energy_unit_operation** is **energy_transfer**, by inheritance, the use of **h_1** is **energy_transfer**.
We may ask additional questions and see how the semantic network responds:

```
?- fact(type(h_3,X)).
X = shell_and_tube ;
X = continuous ;
no
```

Here, by inheritance, **h_3** is both a **shell_and_tube** type of heat exchanger as well as a **continuous** type of heat transfer operation.

C. Frame-Based Knowledge

As discussed in sections 10.2F and G, there are many different ways to implement frames in Prolog. We implement the frame using a set of Prolog facts in the form:

frame_name(*Slot*,*Interpretation*, *ValueList*).

Four frames are shown in Figure 11.9. The frame at the top of the hierarchy is the **pump** frame. Next in the hierarchy are the **centrifugal_pump** and **positive_displacement_pump**. At the bottom of the hierarchy is the **unit_101** frame.

With this frame format, we need some *primitives* for manipulating frames. These primitives must be able to manipulate *any* frame in the program, and in that sense, are general. Primitives are "utility" operations, used to process frames in a frame-based system. Schnupp et. al. (1989, pp. 126-143) give a good summary of "typical" primitives for frame-based systems written in Prolog. To get a broader understanding of frames in Prolog, we recommend their work. Here, we discuss below some useful primitives for processing frames.

inherit(*Frame*, *Slot*, *Interpretation*, *ValueList*).

This relation implements automatic inheritance to determine an instantiation for *ValueList*. This procedure is closely related to the inheritance procedure introduced under semantic networks in section 10.2B.

```
pump(general,is_a,[]).
pump(unit,value,_x).
pump(type,value,_x).
pump(material,value,_x).
pump(capacity,value,_x).
pump(motor,value,_x).
pump(head,value,_x).
pump(temperature,value,_x).
pump(inlet,value,_x).
pump(outlet,value,_x).
pump(pump_curve,procedure,[[GPM],[Head]]).

centrifugal_pump(pump,is_a,[]).
centrifugal_pump(unit,value,_x).
centrifugal_pump(type,value,_x).
centrifugal_pump(material,value,_x).
centrifugal_pump(capacity,value,_x).
centrifugal_pump(motor,value,[electric]).
centrifugal_pump(head,value,_x).
centrifugal_pump(temperature,value,_x).
centrifugal_pump(inlet,value,_x).
centrifugal_pump(outlet,value,_x).
centrifugal_pump(pump_curve,procedure,[GPM,Head].

positive_displacement_pump(general,is_a,[]).
positive_displacement_pump(unit,value,_x).
positive_displacement_pump(type,value,_x).
positive_displacement_pump(material,value,_x).
positive_displacement_pump(capacity,value,_x).
positive_displacement_pump(motor,value,_x).
positive_displacement_pump(head,value,_x).
positive_displacement_pump(temperature,value,_x).
positive_displacement_pump(inlet,value,_x).
positive_displacement_pump(outlet,value,_x).
positive_displacement_pump(pump_curve,procedure,[GPM,Head]).

unit_101(centrifugal_pump,isa,[]).
unit_101(unit,value,[101]).
unit_101(type,value,[centrifugal]).
unit_101(material,value,[stainless_steel_316]).
unit_101(capacity,value,[120,gpm]).
unit_101(motor,value,[electric]).
unit_101(head,value,[55,feet]).
unit_101(temperature,value,[120,f]).
unit_101(inlet,value,[unit_100]).
unit_101(outlet,value,[unit_102]).
```

Figure 11.9. Prolog facts grouped into a frame-based system.

When the **inherit** clause is called, both *Frame* and *Slot* must be instantiated, or the clause will fail. Likewise, *Interpretation* must be instantiated to the atom **value**, or the clause will fail (we are only allowed to inherit values, not procedures). If *ValueList* is available within the current frame, it will be instantiated to the matching term in the database. If *ValueList* is not available within the current frame, the **inherit** procedure will climb the **isa** hierarchy in search of a default value. The Prolog code is shown below.

```
% ValueList is available within the current node. Therefore, do not
% inherit, since local values supersede all others

inherit(Frame, Slot, value, ValueList):-
     nonvar(Frame),nonvar(Slot),
     X =.. [Frame, Slot, value, ValueList],
     call(X),
     nonvar(ValueList).

%  ValueList is not available within the current frame.
%  Therefore, perform inheritance, and climb the isa hierarchy.

inherit(Frame, Slot, value, ValueList):-
       nonvar(Frame),nonvar(Slot),
       X =.. [Frame, HigherFrame, isa, _],
       call(X),
       inherit(HigherFrame, Slot, value, ValueList).
```

if_procedure(*Frame*, *Slot*, *Interpretation*, *ProcedureList*).

This procedure implements the *if_added*, *if_needed*, and *if_removed* procedural attachments to the specified *Frame* and *Slot*. There are no restrictions placed on how the *Interpretation* is instantiated -- that issue is dictated by the attached procedures themselves. Usually, however, the interpretation is one of the three forms: **if_added**, **if_needed**, or **if_removed**. Both *Frame* and *Slot* must be instantiated or the procedure will fail. If *ProcedureList* contains multiple procedures, all attached procedures will be implemented. The **if_procedure** itself does not rely on any supporting primitives, but frequently the attached procedures do (in particular, they frequently rely on the **inherit** primitive). Finally, **if_procedure** is deterministic and cannot be backtracked into. The Prolog code is simple and is shown below.

```
if_procedure(_,_,_,[]):- !.
if_procedure(Frame,Slot,Interpretation,[Proc|Rest]):-
      nonvar(Frame), nonvar(Slot),
      execute(Frame,Slot,Interpretation,Proc),
      if_procedure(Frame,Slot,Interpretation,Rest).

execute(Frame,Slot,Interpretation,Proc):-
     X =.. [Proc,Frame,Slot,Interpretation],
     call(X).
```

This **if_procedure** relation simply executes attached procedures and relies upon those procedures to execute correctly. To better understand how the **if_procedure** clause interacts with attached procedures, let us consider an example. The frame-based system is shown in Figure 11.10.

```
pump(general,is_a,[]).
pump(unit,value,_x).
pump(type,value,_x).
pump(material,value,_x).
pump(capacity,value,_x).
pump(motor,value,_x).
pump(head,value,_x).
pump(temperature,value,[120,f]).
pump(inlet,value,_x).
pump(outlet,value,_x).
pump(temperature,if_added,[include_c,include_k]). % Procedural Attachment

include_c(pump,temperature,if_added):-
      pump(temperature,value,[Temp,f]),
      TempC is (Temp - 32)*5/9,
      assertz(pump(temperature,value,[TempC,c])).
include_k(pump,temperature,if_added):-
      pump(temperature,value,[Temp,f]),
      TempK is 273.15+(Temp - 32)*5/9,
      assertz(pump(temperature,value,[TempK,k])).

centrifugal_pump(pump,is_a,[]).
centrifugal_pump(unit,value,_x).
centrifugal_pump(type,value,_x).
centrifugal_pump(material,value,_x).
centrifugal_pump(capacity,value,_x).
centrifugal_pump(motor,value,[electric]).
centrifugal_pump(head,value,_x).
centrifugal_pump(temperature,value,_x).
centrifugal_pump(inlet,value,_x).
centrifugal_pump(outlet,value,_x).
centrifugal_pump(temperature,if_needed,[get_temperature]). % Procedural
                                                           % Attachment
get_temperature(centrifugal_pump,temperature,if_needed):-
      inherit(centrifugal_pump,temperature,value,X),
      (retract(centrifugal_pump(temperature,value,_));true),
      assertz(centrifugal_pump(temperature,value,X)).

if_procedure(_,_,_,[]):- !.
if_procedure(Frame,Slot,Interpretation,[Proc|Rest]):-
      nonvar(Frame), nonvar(Slot),
      execute(Frame,Slot,Interpretation,Proc),
      if_procedure(Frame,Slot,Interpretation,Rest).

execute(Frame,Slot,Interpretation,Proc):-
      X =.. [Proc,Frame,Slot,Interpretation], call(X).
```

Figure 11.10. A frame-based system with attached procedures.

There are two procedural attachments in this database: **if_added** and **if_needed**. The **if_added** procedure belongs to the **pump** frame and is identified by:

```
pump(temperature,if_added,[include_c,include_k]).
```

This procedure says that if we add the temperature of the fluid to the **pump** frame, we call the procedures **include_c** and **include_k**. These two procedures convert the temperature from °F to °C and to K, and assert these facts into the frame.

The **if_needed** procedure is attached to the **centrifugal_pump** frame, and is identified by the fact:

```
centrifugal_pump(temperature,if_needed,[get_temperature]).
```

This procedure says that if we need the fluid temperature for the **centrifugal_pump** frame, we call the **get_temperature** relation. This relation uses inheritance to find the temperature.

find(*Frame*, *Slot*, *Interpretation*, *Value*).

This primitive finds and unifies the specified *Frame* with the given *Slot*, *Interpretation*, and *Value*. *Frame* must be instantiated and act as an input term, or the clause will fail. The remaining arguments (*Slot*, *Interpretation*, and *Value*) can be instantiated or uninstantiated. Thus, when the clause is called, the remaining arguments can act as either input or output terms. The clause can respond in numerous ways, depending on the nature of *Value*:

- If *Value* is free variable or a list, then the specified *Frame*, *Slot*, *Interpretation*, and *Value* is unified with a matching fact in the database. If a matching slot is found and that slot is empty, automatic inheritance will be invoked to fill the slot from a higher-level frame.

- If *Value* is a constant or a structure, then the clause succeeds only if *Value* is a member of the *ValueList* associated with the specified *Frame*, *Slot*,and *Interpretation*. *Value* is treated as a single-data object even if it is a list or structure.

The Prolog code is shown below. It requires some supporting relations, such as **inherit** (defined previously), and utility relations such as **member**, **flatten**, and **append**.

```
find(Frame, Slot, Interpretation, Value):-
      ( var(Value) ;
        is_list(Value) ),
      X =.. [Frame, Slot, Interpretation, Value],
      ( call(X)  ;
        inherit(Frame, Slot, Interpretation, Value)).
```

```
find(Frame, Slot, Interpretation, Value):-
      nonvar(Value),
      X =.. [Frame, Slot, Interpretation, ValueList],
      call(X),
      ( member(Value,ValueList)
        ;
        flatten(ValueList, FlatValueList),
        member(Value, FlatValueList)
      ).

is_list(X):- var(X), !, fail.
is_list([_|_]):- !.
is_list([]).

member(X,[X|_]).
member(X,[_|T]):- member(X,T).

flatten([H|T],FL):-
     flatten(H,FH),
     flatten(T,FT),
     append(FH,FT,FL).
flatten([],[]).
flatten(X,[X]).

append([],L,L).
append([H|T1],L2,[H|T3]):-
     append(T1,L2,T3).
```

delete_frame(*Frame*).

This procedure eliminates the entire *Frame* from the database. The argument *Frame* must be instantiated, or the clause will fail. If a matching frame is found, it is eliminated. If no matching frame is found, the procedure succeeds and does nothing. If the *Frame* is at the top of the hierarchy (i.e., is of the **general** class), it will not be deleted and the user will be told why. If the *Frame* has an **if_removed** procedure, the **if_removed** procedure will be activated. The **delete_frame** clause is deterministic and cannot be backtracked into. The Prolog code is shown below.

```
delete_frame(Frame):-
     nonvar(Frame),
     X =.. [Frame, general, _, _],
     call(X),
     nl,write(' The frame is at the top of the'),nl,
     write(' hierarchy and cannot be deleted'), !.

delete_frame(Frame):-
     nonvar(Frame),
     ( Y =.. [Frame, Slot, Interpretation, List],
       retract(Y),
       Interpretation = if_removed,      % activate if-removed procedure
       if_procedure(Frame,Slot,Interpretation,List),
       fail ;
       true), !.

delete_frame(Frame):- nonvar(Frame), !.
```

delete_frame(*Frame*,*Slot*).

This procedure eliminates the corresponding *Slot* from the desired *Frame* in the database. Both arguments *Frame* and *Slot* must be instantiated, or the clause will fail. If a matching frame and slot is found, it is eliminated. If no matching frame and slot is found, the procedure succeeds and does nothing. If the slot has more than one interpretation, then all facts that match the frame and slot, regardless of the interpretation, are removed. If any **if_removed** procedures exist for the slot, then these **if_removed** procedures are activated. The **delete_frame** procedure is deterministic and cannot be backtracked into. The Prolog code is shown below.

```
delete_frame(Frame,Slot):-
     nonvar(Frame),nonvar(Slot),
     ( Y =.. [Frame, Slot, Interpretation, List],
       retract(Y),
       Interpretation = if_removed,        % execute if-removed procedure
       if_procedure(Frame,Slot,Interpretation,List),
       fail ;
       true), !.

delete_frame(Frame,Slot):-
       nonvar(Frame), nonvar(Slot), !.
```

delete_frame(*Frame*,*Slot*,*Interpretation*).

This procedure eliminates all facts with the corresponding *Slot* from the desired *Frame* and the given *Interpretation* in the database. All of the arguments (*Frame*, *Slot*, and *Interpretation*) must be instantiated, or the clause will fail. If no frame with the matching slot and interpretation is found, the procedure succeeds and does nothing. If the specific slot also has **if_removed** procedure, then that procedure is activated. The procedure is deterministic and cannot be backtracked into. The Prolog code is shown below.

```
delete_frame(Frame,Slot,Interpretation):-
     nonvar(Frame),nonvar(Slot),nonvar(Interpretation),
     ( Y =.. [Frame, Slot, Interpretation, List],
       retract(Y),
       fail ;
       true),
     ( Z =.. [Frame, Slot, if_removed, ProcedureList],
       call(Z),                              % execute if-removed procedure
       if_procedure(Frame, Slot, if_removed, ProcedureList);
       true), !.

delete_frame(Frame,Slot,Interpretation):-
     nonvar(Frame),nonvar(Slot),nonvar(Interpretation), !.
```

delete_frame(*Frame*,*Slot*,*Interpretation*,*Value*).

This procedure eliminates the corresponding *Value* from *ValueList* in the *Slot* from the desired *Frame* with the specific *Interpretation* given. All of the arguments (*Frame*, *Slot*, *Interpretation*, and *Value*) must be instantiated, or the clause will fail. If no frame is found with the matching slot, interpretation, and value, the procedure succeeds and does nothing. If the specific slot has an **if_removed** procedure, then that procedure is activated. The term *Value* is treated as a single item to be deleted, even if it is a list or a structured object. Thus, the term *Value* will not be decomposed in an attempt to match an item in *ValueList*. This procedure is deterministic and cannot be backtracked into. The Prolog code is shown below.

```
delete_frame(Frame,Slot,Interpretation,Value):-
     nonvar(Frame),nonvar(Slot),
     nonvar(Interpretation),nonvar(Value),
     X =.. [Frame, Slot, Interpretation, ValueList],
     call(X),
     delete(Value,ValueList,NewValueList),
     retract(X),
     Y =.. [Frame, Slot, Interpretation, NewValueList],
     assert(Y),
     ( Z =.. [Frame, Slot, if_removed, ProcedureList],
       call(Z),                           % execute if-removed procedure
       if_procedure(Frame, Slot, if_removed, ProcedureList);
       true),!.

delete_frame(Frame,Slot,Interpretation,Value):-
     nonvar(Frame),nonvar(Slot),
     nonvar(Interpretation),nonvar(Value), !.

delete(X, [X|Tail], Tail).
delete(X, [Y|Tail1], [Y|Tail2]):-
     delete(X,Tail1,Tail2).
```

add(*Frame*,*Slot*,*Interpretation*,*Value*).

This procedure adds the corresponding *Value* onto the end of *ValueList* in the *Slot* from the desired *Frame* with the specific *Interpretation* given.

Note that Value is added onto the end of *ValueList* even if *Value* is already a member of the list. All of the arguments (*Frame*, *Slot*, *Interpretation*, and *Value*) must be instantiated, or the clause will fail. If a frame is found that matches *Frame*, but there is no matching slot and interpretation, an additional slot with its interpretation is created, and the *ValueList* becomes *[Value]*. If no matching frame is found, the procedure succeeds and does nothing. This procedure is deterministic and cannot be backtracked into. The Prolog code is shown below.

```
add(Frame,Slot,Interpretation,Value):-        % Slot does exist; use it
     nonvar(Frame),nonvar(Slot),
     nonvar(Interpretation), nonvar(Value),
     X =.. [Frame, Slot, Interpretation, ValueList],
     call(X),
     append(ValueList, [Value], NewValueList),
     retract(X),
     Y =.. [Frame, Slot, Interpretation, NewValueList],
     assertz(Y),           % implement if_added procedure if it exists
     ( Interpretation = value,
       Z =.. [Frame, Slot, if_added, ProcedureList],
       call(Z),
       if_procedure(Frame,Slot,if_added,ProcedureList)
     ;
     true).

add(Frame,Slot,Interpretation,Value):-  % Slot does not exist; create it
     nonvar(Frame),nonvar(Slot),
     nonvar(Interpretation),nonvar(Value),
     X =.. [Frame, Slot, Interpretation, [Value]],
     assertz(X).
```

modify(*Frame,Slot,Interpretation,NewList*).

This procedure modifies the specified *Frame*, *Slot*, and *Interpretation* by eliminating the old *ValueList* and substituting it with *NewList*. All four arguments of the **modify** clause (i.e., *Frame*, *Slot*, *Interpretation*, and *NewList*) must be instantiated or the clause will fail. If the specified *Frame*, *Slot*, and *Interpretation* do not exist, a corresponding slot will be created. If any **if_added** or an **if_removed** procedures are attached, they will be activated. The Prolog code is shown below.

```
modify(Frame,Slot,Interpretation,NewList):-   % Slot does exist; use it
     nonvar(Frame),nonvar(Slot),
     nonvar(Interpretation), nonvar(NewList),
     X =.. [Frame, Slot, Interpretation, ValueList],
     call(X),
     retract(X),
     Y =.. [Frame, Slot, Interpretation, NewList],
     assertz(Y),
     call_procedures(Frame,Slot). % call if_added or if_removed
                                  % procedures if they exist
modify(Frame,Slot,Interpretation,NewList):-   % Slot does not exist
     nonvar(Frame),nonvar(Slot),
     nonvar(Interpretation),nonvar(NewList),
     X =.. [Frame, Slot, Interpretation, NewList],
     assertz(X).

call_procedures(Frame,Slot):-
     Z =.. [Frame, Slot, if_added, ProcedureList],
     call(Z),
     if_procedure(Frame,Slot,if_added,ProcedureList),
     fail.

call_procedures(Frame,Slot):-
     Z =.. [Frame, Slot, if_removed, ProcedureList],
     call(Z),
     if_procedure(Frame,Slot,if_added,ProcedureList),
     fail.

call_procedures(_,_).
```

To summarize, we have introduced the following primitives for processing frames in a frame-based system written in Prolog:

- **inherit(** *Frame*, *Slot*, *Interpretation*, *ValueList*).
- **if_procedure(** *Frame*, *Slot*, *Interpretation*, *ProcedureList*).
- **find(** *Frame*, *Slot*, *Interpretation*, *Value*).
- **delete_frame(** *Frame*).
- **delete_frame(** *Frame*, *Slot*).
- **delete_frame(** *Frame*, *Slot*, *Interpretation*).
- **delete_frame(** *Frame*, *Slot*, *Interpretation*, *Value*).
- **add(** *Frame*, *Slot*, *Interpretation*, *Value*).
- **modify(** *Frame*, *Slot*, *Interpretation*, *NewList*).

There are other primitives that may be useful in a frame-based system, such as displaying frames and facilitating input-output functions. Many of these primitives are specific both to the version of Prolog we are using and to the available hardware. Nevertheless, the primitives defined above make up the "core" of a frame-based system written in Prolog.

D. Object-Oriented Programming

In a "true" object-oriented program, all program execution is performed under the umbrella of individual objects. Objects communicate with each other, and there are no procedures executed that do not "belong" to an object.

Converting from a frame-based system to an object-oriented program is not difficult in Prolog. We use the exact same primitives described in section 11.2C for frames. We recall that in frame-based systems written in Prolog, a frame is a grouping of related facts in the format:

frame_name(*Slot*, *Interpretation*, *ValueList*).

If *Interpretation* = *value*, then that slot holds data for the program to use. On the other hand, if *Interpretation* is instantiated to *if_added*, *if_needed*, or *if_removed*, then that slot holds an attached procedure that we execute if specific conditions are met.

An object-oriented program contains procedures that are *always* executed whenever the frame is called. What distinguishes an object-oriented program from a frame-based system is the existence of facts in the form:

```
frame_name(Slot, procedure, ProcedureList).
```

Thus when the *Interpretation* = *procedure*, the *ProcedureList* is always executed whenever the frame is called. Let us assume that we have an object with the following structure:

```
unit_101(centrifugal_pump,isa,[]).
unit_101(unit,value,[101]).
unit_101(type,value,[centrifugal]).
unit_101(material,value,[stainless_steel_316]).
unit_101(capacity,value,[120,gpm]).
unit_101(motor,value,[electric]).
unit_101(head,value,[55,feet]).
unit_101(temperature,value,[120,f]).
unit_101(inlet,value,[unit_100]).
unit_101(outlet,value,[unit_102]).
unit_101(temperature,if_added,[maintain]).
unit_101(pump_curve,procedure,[GPM,Head]).
unit_101(flow_rate,procedure,[[unit_100,flow], [unit_102, flow]]).
```

What makes this set of facts an *object* instead of a *frame* is the presence of the last two facts, **unit_101(pump_curve,procedure,[GPM,Head])**, and **unit_101(flow_rate,procedure,[[unit_100,flow], [unit_102, flow]])**. The *Interpretation* = *procedure* tells us to always execute this procedure whenever the **unit_101** object is accessed.

How do we execute these procedures every time? The "originating" object that calls and accesses the **unit_101** object must test to see if there is a slot within the **unit_101** object where the *Interpretation* = *procedure*. If the procedure exists, it must be called.

To understand this concept further, let us assume that we are using an object-oriented program for fault-diagnosis of a distillation column. We have **unit_100** (the reboiler), **unit_101** (the bottoms pump), and **unit_102**, the bottoms storage tank, as shown in Figure 11.11.

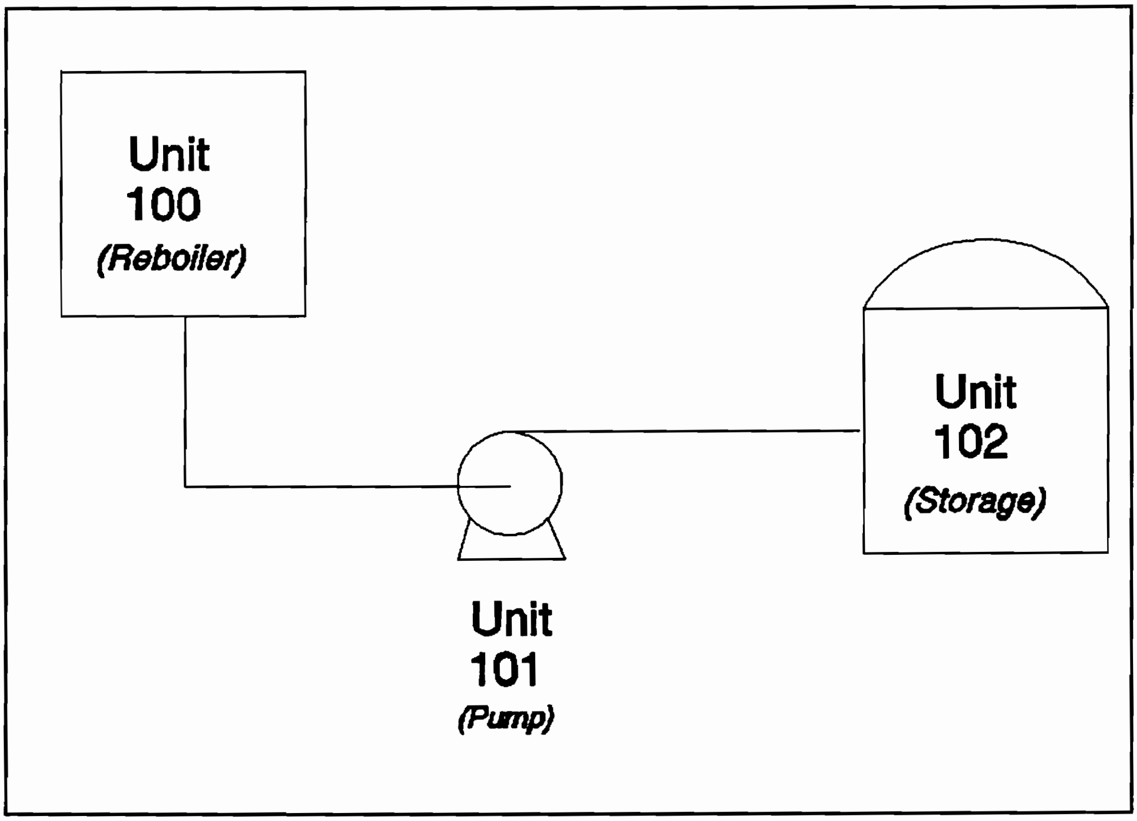


Figure 11.11. A flow diagram of a reboiler, pump, and storage tank.

Each unit number represents an object. Thus, in this program, we need a minimum of three objects: **unit_100**, **unit_101**, and **unit_100**. The **unit_101** object is the same as shown previously.

Let us assume that the plant-control system detects an abnormally low flow rate from the reboiler. This action, in-turn, causes the on-line fault-diagnosis system to implement a procedure using the clause **start**. The start clause is defined below, and uses primitives described in frame-based knowledge representation of section 11.2C:

```
start(Unit, Slot, Interpretation, ValueList, Flow):-
      Flow = low,
      find(Unit, Slot, Interpretation, Value), % object accessed
      X =.. [Unit, Slot, procedure, ProcedureList],
      ( call(X),  % an essential procedure exists
      if_procedure(Unit, Slot, procedure, ProcedureList) % execute the
      ) ; true.                                          % procedure
```

Given our frame-based primitives, this clause shows how easy it is to implement an object-oriented program. We access the object instantiated to **Unit** using the **find** primitive. We then use the built-in *univ* (= ..) and **call** predicates to determine if the object that we access has an essential procedure that must be executed. If that essential procedure exists, we use the **if_procedure** primitive to execute it.

E. Practice Problems

1. Figure 11.7 shows a forward-chaining rule interpreter written in Prolog. Develop and write an explicit backward-chaining rule interpreter in Prolog.

2. Table 11.2 shows the Prolog code for automatic inheritance in a semantic network. This procedure climbs the **isa** hierarchy to inherit a value from a higher-level node. One principle of semantic networks that this program does not implement is *multiple inheritance*. If a lower-level node is connected to *two or more* higher-level nodes, then that lower-level node inherits *all* of the properties. Write the Prolog code that performs multiple inheritance in a semantic network.

3. Develop the primitive **display_frame(*Frame*)** for a frame-based system. This primitive displays every *Interpretation* and *ValueList* for each *Slot* in the *Frame*. The display is made to the current output stream. The term *Frame* must be instantiated; if it is uninstantiated, the primitive tells the user and then succeeds and does nothing.

4. Develop the primitive **display_chain(*Frame*)** for a frame-based system. The primitive displays, in an ascending order, the hierarchy of frames from the given *Frame* up to the top of the hierarchy. Recall that the isa link defines the hierarchy. The frame at the top of the hierarchy belongs

to the **general** class. The term *Frame* must be instantiated; if it is uninstantiated, the primitive tells the user and then succeeds and does nothing.

5. The primitive **delete_frame(***Frame***)** has a problem, and we must fix it. What if one or more lower-level frames rely on the *Frame* that we delete via the **isa** slot for inheritance? If we delete *Frame*, the inheritance chain may break down. Adjust the **delete_frame(** *Frame* **)** clause such that it will test to see if any lower-level frames rely on it. If indeed these lower-level frames exist, then inform the user and do not delete the *Frame*.

11.3 OTHER PROBLEM-SOLVING STRATEGIES

A. AND-OR Strategies

AND-OR graphs are a way of representing problems that facilitates problem-solving. They are useful for problems that can be broken into mutually exclusive sub-problems. They can be very helpful for scientific and engineering applications of Prolog, since

(1) many scientific and engineering problems involve mutually exclusive sub-problems; and

(2) based on their training, most scientists and engineers are familiar with a "divide-and-conquer" approach to problem solving.

Consider an oil-pipeline network shown in Figure 11.12. All oil from the field originates at pumping station **a**. Our goal is to get oil from station **a** to ocean-going tankers off the coastline at terminal station **t**. To do this, however, the oil needs to pass from country I to country II. Country II charges country I for use of its oil-pipeline network, and keeps track of the amount pumped through at border stations **g** and **h**.

From an AND-OR graph standpoint, we know that to get from starting node **a** to goal node **t**, we *must* pass through either node **g** or **h**. We divide the problem into two sub-problems:

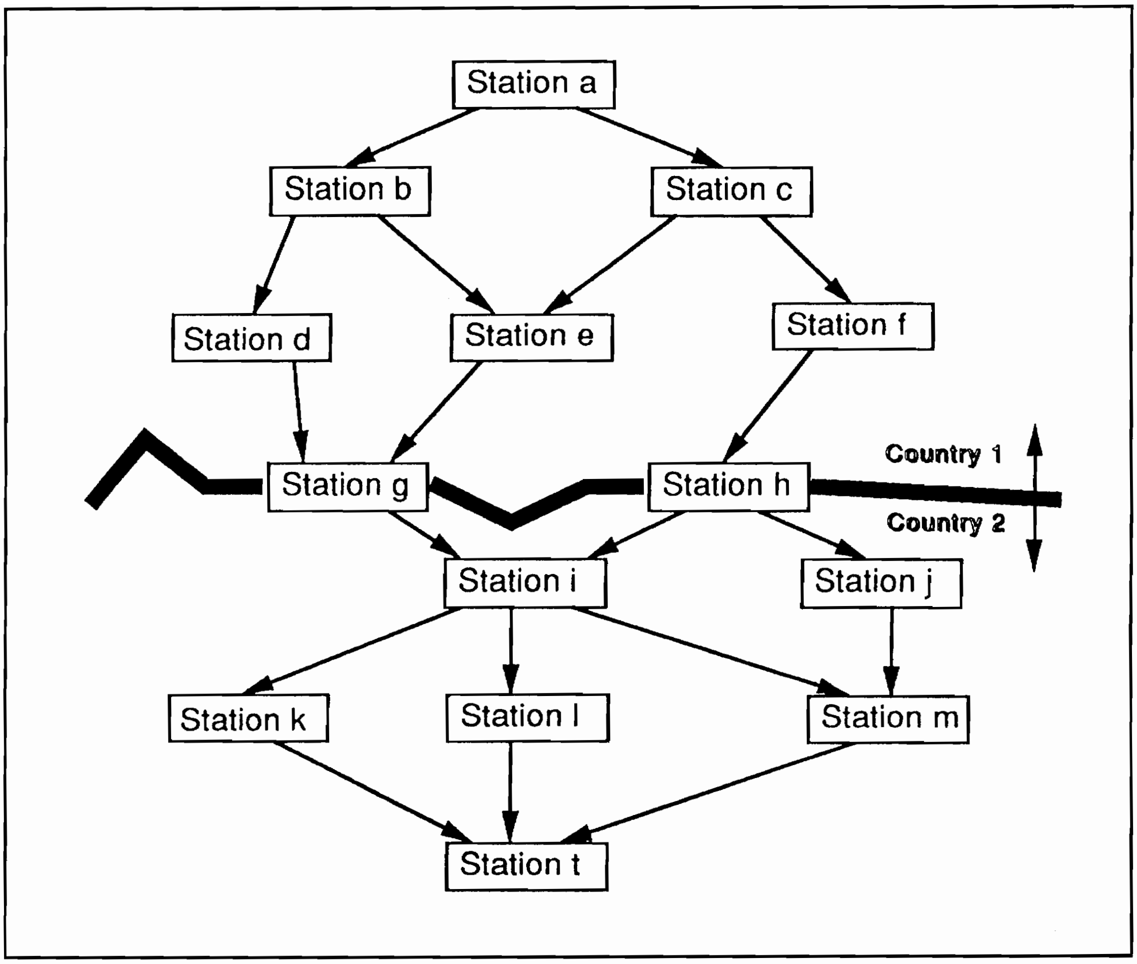


Figure 11.12. An oil-pipeline network.

(1) station a to station **t** via station **g**

OR

(2) station a to station **t** via station **h**

Importantly, these two problems are mutually exclusive (i.e., there is an *or* between them), and can be solved independently. To solve each of them, we:

(1) move from station a to station **t** via station **g**, passing through:

1A) station **a** to station **g**, *and*

1B) station **g** to station **t**

(2) move from station a to station **t** via station **h**, passing through:

2A) station **a** to station **h**, *and*

2B) station **h** to station **t**.

We are well on way to decomposing the problem. The AND-OR graph of Figure 11.13 shows the steps we have taken so far.

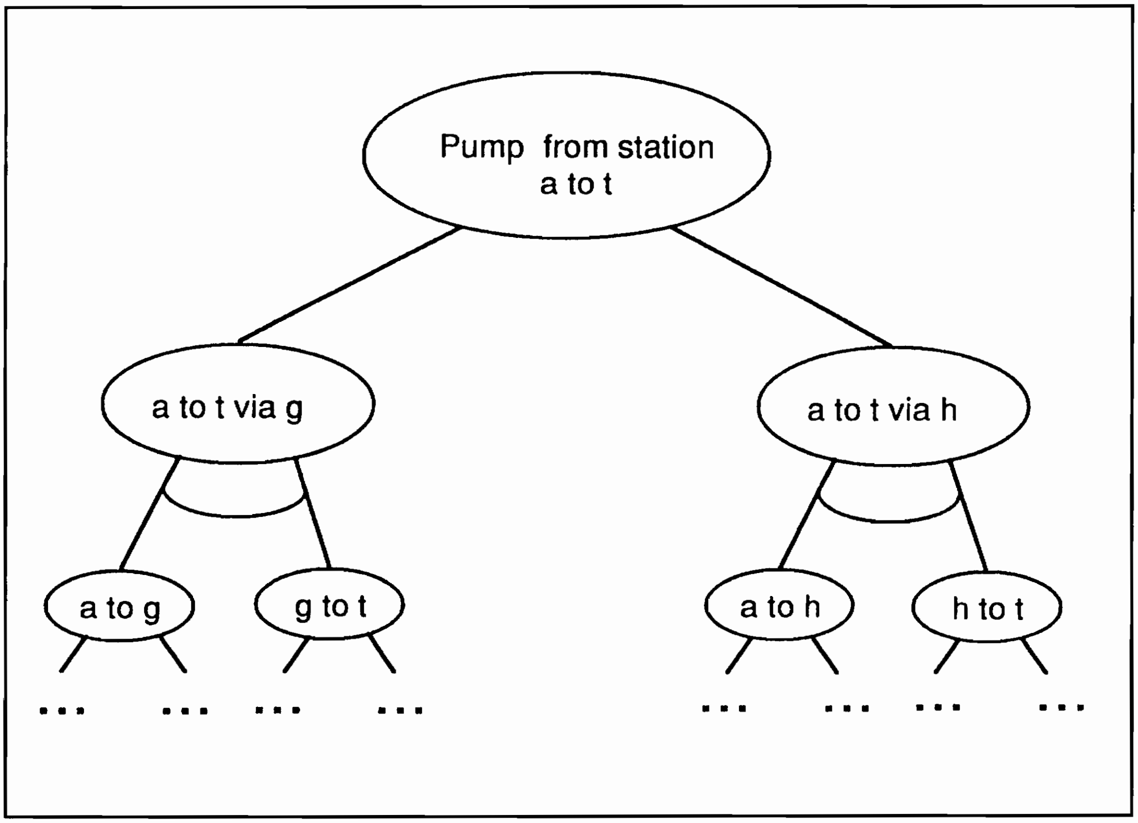
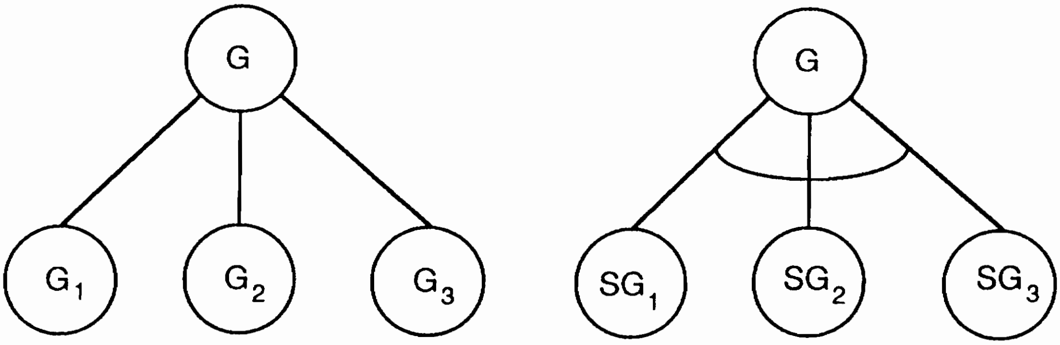


Figure 11.13. The AND-OR graph for the oil-pipeline network.

When we perform a state space analysis of Figure 11.13, we see that a node can issue AND-related arcs and OR-related arcs. Figure 11.14 shows an OR node and an AND node. Graphically, the OR node has unconnected goals. If we satisfy *any* goal $\mathbf{G}_i$ in the OR node, then goal **G** is true. The AND node has connected sub-goals that *all* must be satisfied for goal **G** to be true.



OR NODE

"To solve G, solve goal G_1, G_2, or G_3."

AND NODE

"To solve G, solve sub-goals SG_1 and SG_2 and SG_3."

Figure 11.14. AND-OR representations.

Figure 11.15 shows the complete **g** side of the tree for our oil pumping network.

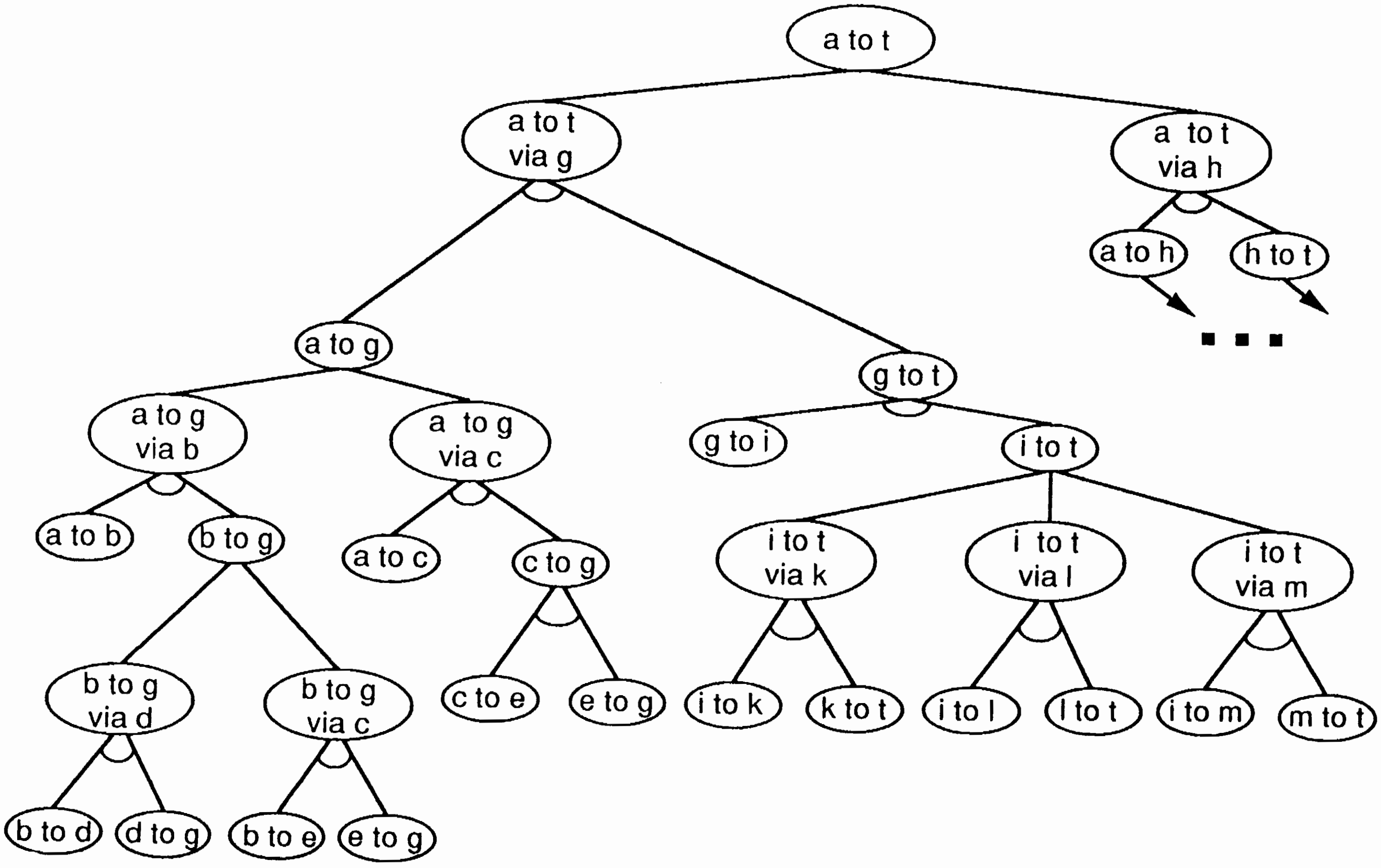


Figure 11.15. The station g side of the oil pipeline network.

Why do we use AND-OR strategies? There are three primary reasons:

1) scientific and engineering problem-solving can frequently be "boiled down" to an AND-OR representation;
2) the AND-OR representation is natural to the logical problem-solving used by Prolog; and
3) it is easier to solve a problem if it can be broken up into sub-problems.

We again look at the g side of the tree in Figure 11.15, and write the following Prolog code for an AND-OR graph:

```
/*******************************************************************
      a to t is an OR node with a to t via g and a to t via h as
      successors.
            Note that 'to' and 'via' are operators, i.e.
      a to t              =    to(a,t)
      a to t via g        =    via(to(a,t),g)
 ********************************************************************/

      a to t :- (a to t via g);
                (a to t via h).


/*********************************************************************
   a to t via g is an AND node with a to g and g to t
   as successors
 *********************************************************************/
```

```
     a to t via g:- a to g,
                    g to t.

/************************************************************************
     The side of the tree from a to t via h is not in the diagram
     and we put it here as a Prolog fact for completeness.
     Actually, the program needs a set of rules for the entire h
     side of the tree.
 ************************************************************************/

     a to t via h.

     a to g:-   (a to g via b);
                (a to g via c).                 % an OR node

     a to g via b:-    a to b,
                       b to g.                  % an AND node

     a to g via c:-    a to c,
                       c to g.                  % an AND node

     b to g:-   (b to g via d);
                (b to g via e).                 % an OR node

     c to g:-   c to e,
                e to g.                         % an AND node

     b to g via d:-    b to d,
                       d to g.                  % an AND node

     b to g via e:-    b to e,
                       e to g.                  % an AND node
```

```
      g to t:-   g to i,
                 i to t.%                       % an AND node

      i to t:-   (i to t via k);
                 (i to t via l);
                 (i to t via m).                % an OR node

      i to t via k:-    i to k,
                        k to t.                 % an AND node

      i to t via l:-    i to l,
                        l to t.                 % an AND node

      i to t via m:-    i to m,
                        m to t.                 % an AND node

/***********************************************************************
      Supporting facts about the pipeline network for the g side of
      the tree only.
***********************************************************************/

      a to b.           a to c.                 b to d.
      b to e.           c to e.                 d to g.
      e to g.           g to i.                 i to k.
      i to l.           i to m.                 k to t.
      l to t.           m to t.
```

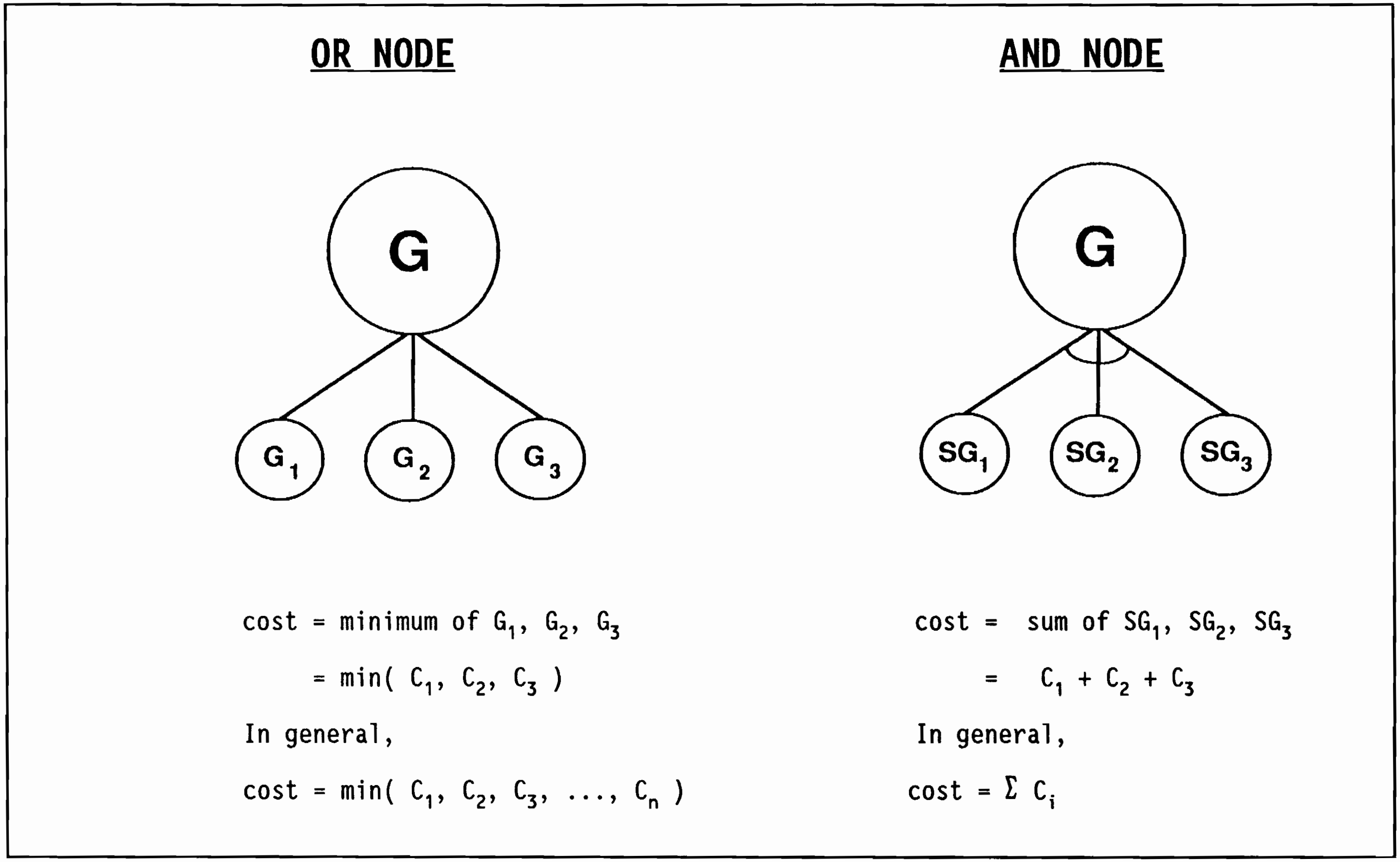
The above program is very declarative and easy to understand. The AND-OR representation is "natural" in Prolog. This simplicity is its key advantage.

Another advantage of AND-OR graphs is that they can be used to implement a best-first search. Here, we have a systematic way of calculating g(n) and expanding to the next node, assuming that *h(n)* is zero or has been calculated. Referring to Figure 11.16, we see that the costs for OR and AND nodes are:

$$\text{OR node cost} = \min(C_1, C_2, C_3, \ldots, C_n)$$

$$\text{AND node cost} = \Sigma\ C_i$$

These calculations provide a systematic method for calculating *g(n)*.



OR NODE

cost = minimum of G_1, G_2, G_3

= min(C_1, C_2, C_3)

In general,

cost = min(C_1, C_2, C_3, ..., C_n)

AND NODE

cost = sum of SG_1, SG_2, SG_3

= $C_1 + C_2 + C_3$

In general,

cost = ΣC_i

Figure 11.16. The g(n) cost for expanding from AND and OR nodes.

B. Constraint Satisfaction

Constraint satisfaction is a *pruning* or *filtering* technique that eliminates certain potential solutions within the state space because they do not satisfy a specific set of constraints. This technique is particularly useful in computer vision and in expert systems targeted towards fault diagnosis or engineering design.

Design problems, for instance, almost always have externally imposed constraints. Typical ones include:

- *Specifications*- what the design is supposed to accomplish.
- *Cost*- what financial targets the design must meet.
- *Materials*- what material size or type is required; what environment the material will be exposed to.
- *Feasibility*- what designs are feasible based on the laws of chemistry and physics (e.g., conservation of mass, energy, momentum; obedience to the laws of thermodynamics).

In constraint satisfaction, the system eliminates a node from the search space if it does not meet all of the constraints required of the solution.

Constraint satisfaction may be used in *any* type of search mode, (depth-first, heuristically driven best-first, etc.). We use constraint satisfaction on a node-by-node basis. In a highly constrained problem, constraint satisfaction may reduce the search space down to only one node-- the solution. But for the majority of AI applications in science and engineering, constraint satisfaction functions as a pruning tool. This approach can reduce the state space by as much as 80-99%. After pruning the state space, we then utilize another search technique to find the final answer.

If we have a Prolog program using constraint satisfaction, we must watch how constraints change as the program executes. Typically, a program generates partial solutions as it progresses through the analysis. Therefore, as the solution develops, the problem inherently becomes more and more constrained. In Prolog, these growing constraints manifest themselves by variables becoming increasingly instantiated. After the analysis is complete, a final solution may be fully constrained; it may have zero degree of freedom. To properly implement constraint satisfaction, then, we must watch how constraints change as the solution develops.

Implementing constraint satisfaction requires us to not only take into account *externally* imposed constraints, but *internal* ones as well. The internal constraints tend to grow as the solution to the problem develops. As an example, let us consider an expert system that allocates raw materials to various units in an oil refinery in an attempt to maintain a schedule and maximize profits. If the program allocates all of the isobutane to the alkylation unit, no isobutane is available to any other unit. This internal constraint *develops* when we decide to allocate all the isobutane.

The general form of the constraint-satisfaction procedure applied on a node-by-node basis is:

(1) Expand to a new node in the search space.

(2) Apply all existing constraint rules to this node, and generate any new constraints that result from the node expansion.

(3) If the constraint application results in a contradiction, eliminate this node from the search space and backtrack to step one.

(4) If no contradiction develops, prune the state space by eliminating all nodes that do not fulfill the current set of constraints, including those generated in step two. Start over in step one with the newly pruned state space.

Steps two and four above are critical to the success of a constraint-satisfaction technique. In step two, the challenge is to generate proper constraint rules based on the partial solution. Step four enforces consistency. A "chain-reaction" occurs when a new constraint is generated. The program *retroactively* removes partial solutions that previously may have been acceptable under a less constrained situation.

The general format for a Prolog code performing constraint satisfaction is:

```
constraint_satisfaction:-
      write('The following nodes are under consideration:'),nl,
      listing(node),   % gives users listing of all nodes, i.e.,
      fail.            % the original state space before pruning.


constraint_satisfaction:-
      node(X),                   % expand to new node.
      apply_constraints(X),      % retract node if constraint violation.
      generate_new_constraints(X), % build constraints caused by new node.
      prune(X),      % retract previous nodes that were
       fail.         %     acceptable but now violate the
                     %     constraints.
constraint_satisfaction:-
      write('The following nodes remain:'),nl,
      listing(node).                  % show state space after pruning
```

Several alternate ways to apply constraint satisfaction exist. Instead of using constraint satisfaction as a pruning tool, we may use it as a state-space generator. Such programs *generate* a set of partial solutions based on their conformity to the current set of constraints. With very large state spaces, generating the set of partial solutions can be more efficient than pruning a large number of unacceptable nodes. After generating the set of partial solutions, the program usually reports them to the user as a *possibilities list*, that is, a list of potential solutions that do not violate the constraints.

We can also use constraint satisfaction to prune *classes* of nodes rather than individual nodes themselves. This technique proves particularly useful in frame-based systems. Pruning a specific class of nodes is more efficient than pruning individually.

C. Generate-and-test

1. Classic Generate-and-Test

Formally, a generate-and-test strategy operates as:

```
solve(X):-
     generate(X),
     test(X).
```

In this technique, a *generator* develops a set of possible solutions and releases them to the *tester*. The *tester* evaluates the set. If it finds an acceptable solution, it reports the result. If it does not find an acceptable solution, the *tester* fails and backtracks to the *generator* in search of more alternatives. Figure 11.17 shows this process.

A true generate-and-test search technique has a nondeterministic generator. The simplest form uses a depth-first search that generates options and sends them to the tester. If the tester rejects these options,

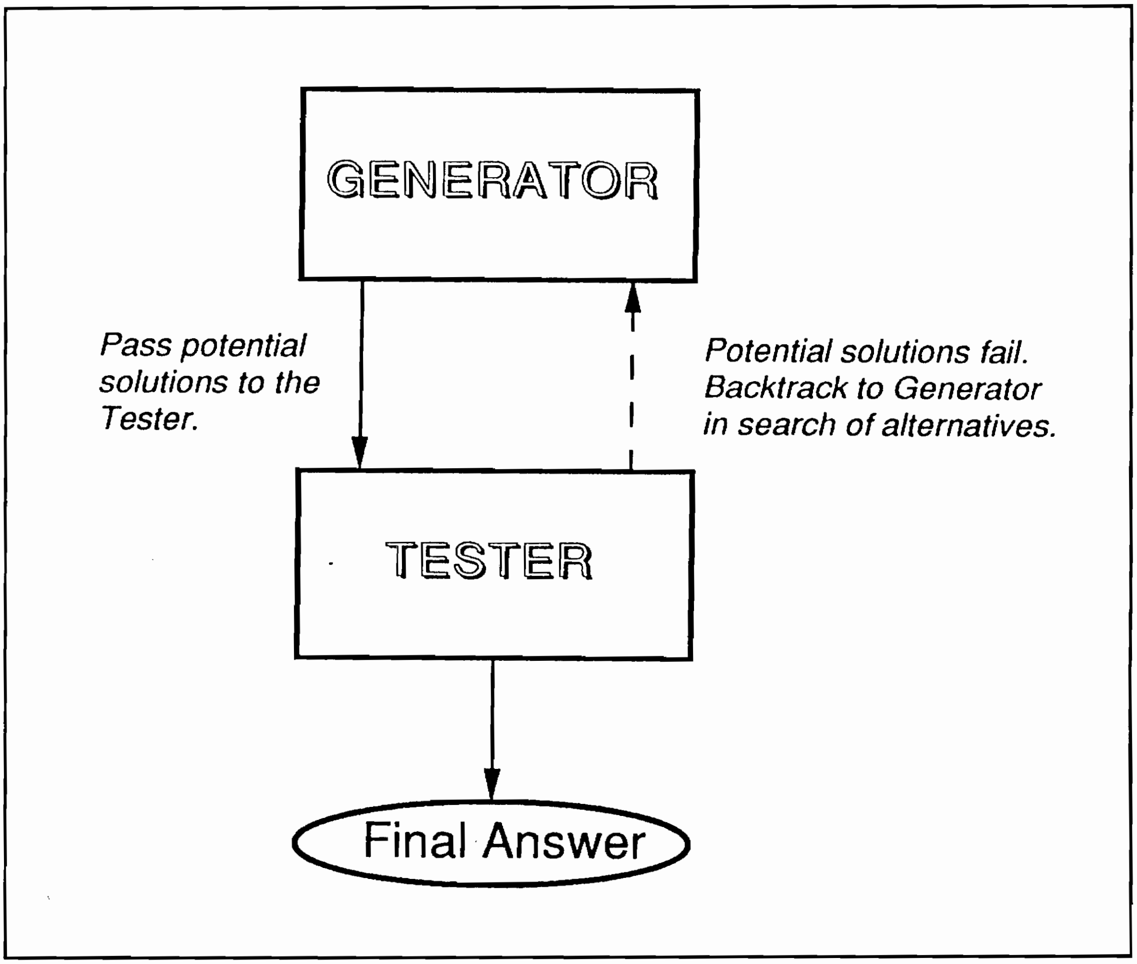

GENERATOR
Pass potential solutions to the Tester.
Potential solutions fail. Backtrack to Generator in search of alternatives.
TESTER
Final Answer

Figure 11.17. Generate-and-test strategy.

the program backtracks and picks up where the depth-first search left off.

Using a depth-first search with the generate-and-test technique, we will eventually find the answer to the problem. Unfortunately, however, the generate-and-test strategy may be inefficient, since time wasted backtracking between the generator and the tester.

To optimize the generate-and-test technique and cut down on useless backtracking, a common approach is to improve the generator. We make the generator do more analysis before releasing the options to the tester. Eventually, the generator may be "pushed" so far into testing that there is no real distinction between the generator and the tester. A single procedure results that has no formal backtracking between the generator and the tester. This effect occurs in highly optimized generate-and-test programs, where there is no clear delineation between the generator and the tester.

2. Plan-Generate-Test

Another way to optimize an inefficient generate-and-test strategy is to use *plan-generate-test*. The very successful expert system DENDRAL uses this method. DENDRAL takes in mass-spectroscopic and nuclear magnetic-resonance data, then enters a *plan* stage, using techniques of constraint satisfaction. The *plan* stage develops a restricted state space that does not contradict the data. The generate-and-test stages pick up where the planner leaves off and explore this confined search space. Since it considers only a limited number of structures, the program finds the answer efficiently. Figure 11.18 represents the *plan-generate-test strategy*; we shall illustrate its use in section 14.3B in the development of an expert system for flowsheet synthesis in multicomponent separations.

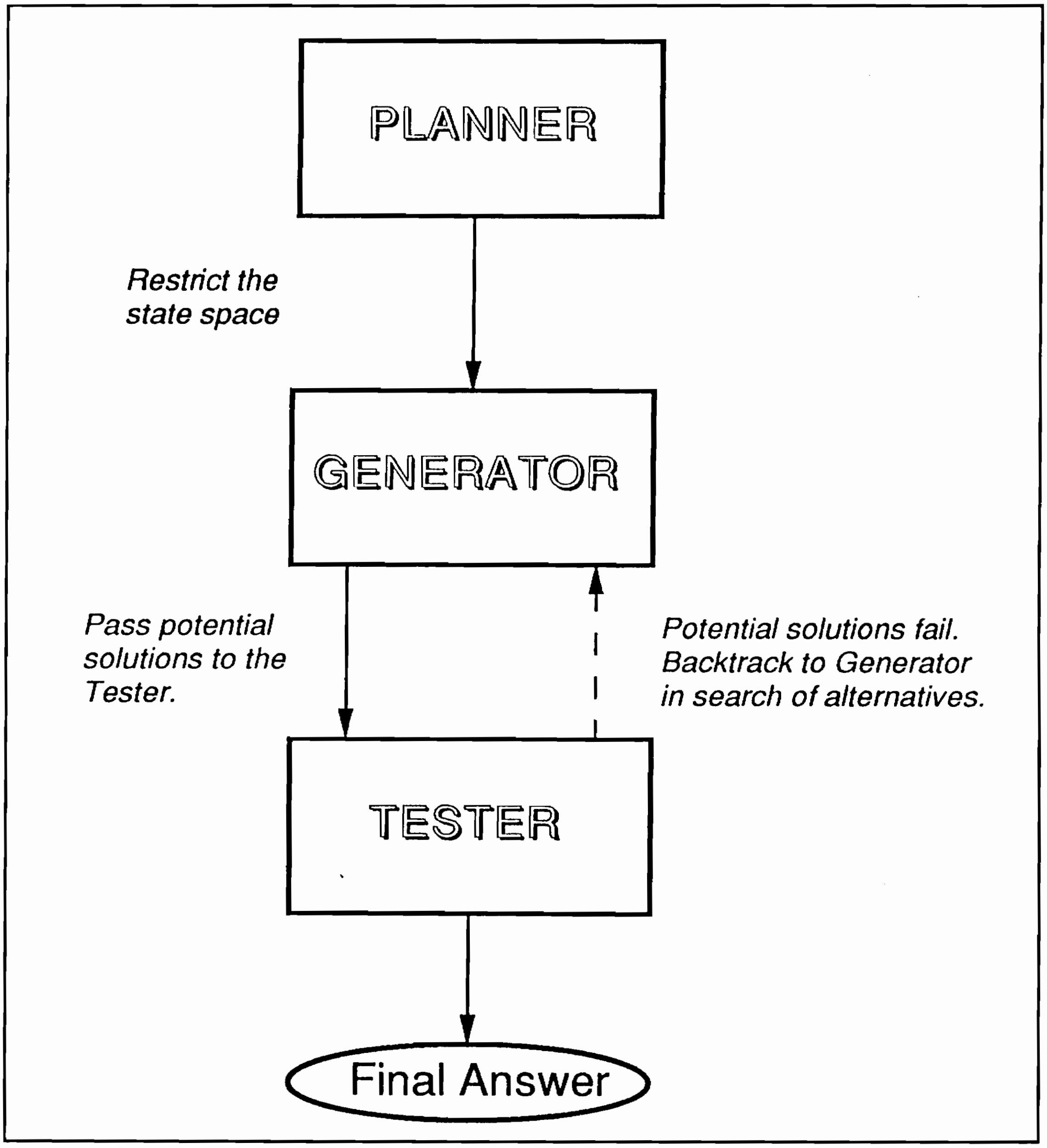

PLANNER
Restrict the state space
GENERATOR
Pass potential solutions to the Tester.
Potential solutions fail. Backtrack to Generator in search of alternatives.
TESTER
Final Answer

Figure 11.18. Plan-generate-test strategy.

3. Hill-Climbing

Still another variation of the generate-and-test technique is *hill-climbing*. In a pure generate-and-test mode, the tester replies *yes* or *no*. When the tester rejects a potential solution, the program simply backtracks to the generator without feeding any information back to the generator. In *hill-climbing*, the tester does more than just answer yes or no. It also provides *guidance* as a form of feedback to the generator.

The guidance may be a heuristic estimate of how close the generated possibilities are to the real solution. It may also be a simple feedback about what search space to focus on next. The guidance tools we develop when using hill-climbing depend on the characteristics of the problem we are trying to solve.

Hill-climbing has been used with strict numerical estimates of how close the program is to the solution. This technique is prone to the same difficulties seen in standard numerical methods used in procedural programs. That is, we can experience:

(1) *Local Maxima*- these lock us onto local hill but prevent us from finding the global solution;

(2) *Instability*- this is caused by a rapidly changing values of the heuristic estimate; and

(3) *Wandering*- this can occur on a *plateau* where the heuristic estimate is relatively constant and we cannot determine the best direction to go.

Identifying and correcting these three problems requires sophisticated numerical analysis. Because AI programs are geared for symbolic computing and are not very efficient with numerical analysis, most AI programs find it particularly difficult to characterize and control these problems when they occur.

D. Means-End Analysis

1. Understanding Means-End Analysis

Means-end analysis was first introduced in a program called the General Problem Solver, also known as GPS (Newell and Simon, 1963; Ernst and Newell, 1968). GPS was mentioned in section 10.1B where we discussed the historical developments of AI.

Means-end analysis works by identifying differences from the current state that we are in and the desired goal state. It then uses operators to reduce this difference. After executing the operator, we may not necessarily arrive at the goal state, but we will hopefully be one step closer. The state that we move into that is hopefully one step closer is called the *intermediate state*. When we apply the operator and move to the intermediate state, we have reduced the problem to a sub-problem of getting from the intermediate state to the goal state. We hope that this sub-problem is easier to solve. Importantly, we can recursively implement this process of assessing where we are and where we want to be, and then applying operators. Figure 11.19 shows a typical means-end analysis.

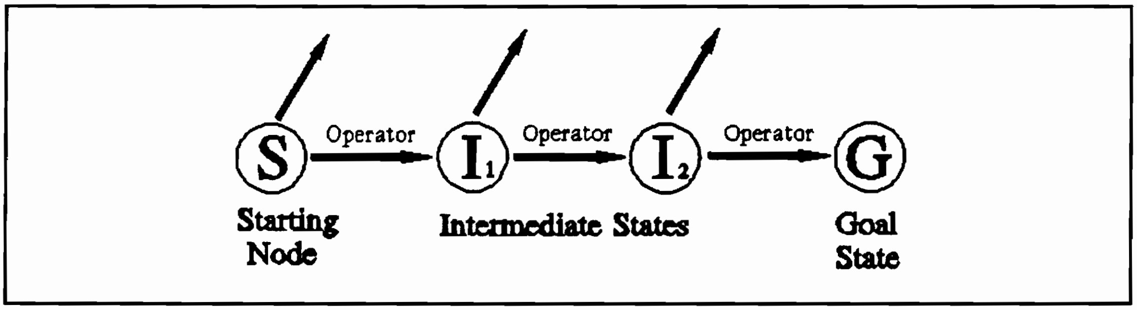


Figure 11.19. An application of means-end analysis.

Means-end analysis can be further complicated by the lack of an available operator. Perhaps an operator exists that can potentially close the difference between the current state and the goal state, but that operator cannot be applied on the current state as it now stands. What is needed is a second operator to move us into yet another intermediate state such that we can apply the first operator. Again, we can recursively implement this process.

To understand means-end analysis, let us apply the technique to chemical process synthesis. Let us that assume we have a mixture of n-hexane and sodium chloride at a temperature (T) of 25°C and a pressure (P) of 101.3 MPa. These composition, T, and P are at starting state 1. We desire a pure benzene stream at 20-30°C as our goal state. We apply the means-end analysis as shown in Figure 11.20. Through analysis of available operators, we wish to apply the following operator:

<u>Operator One:</u> React pure C_6H_{14} in a cyclodehydrogenation reaction to yield benzene, cyclohexane, and hydrogen. The T = 120°C and P = 101.3 MPa for the reactor feed.

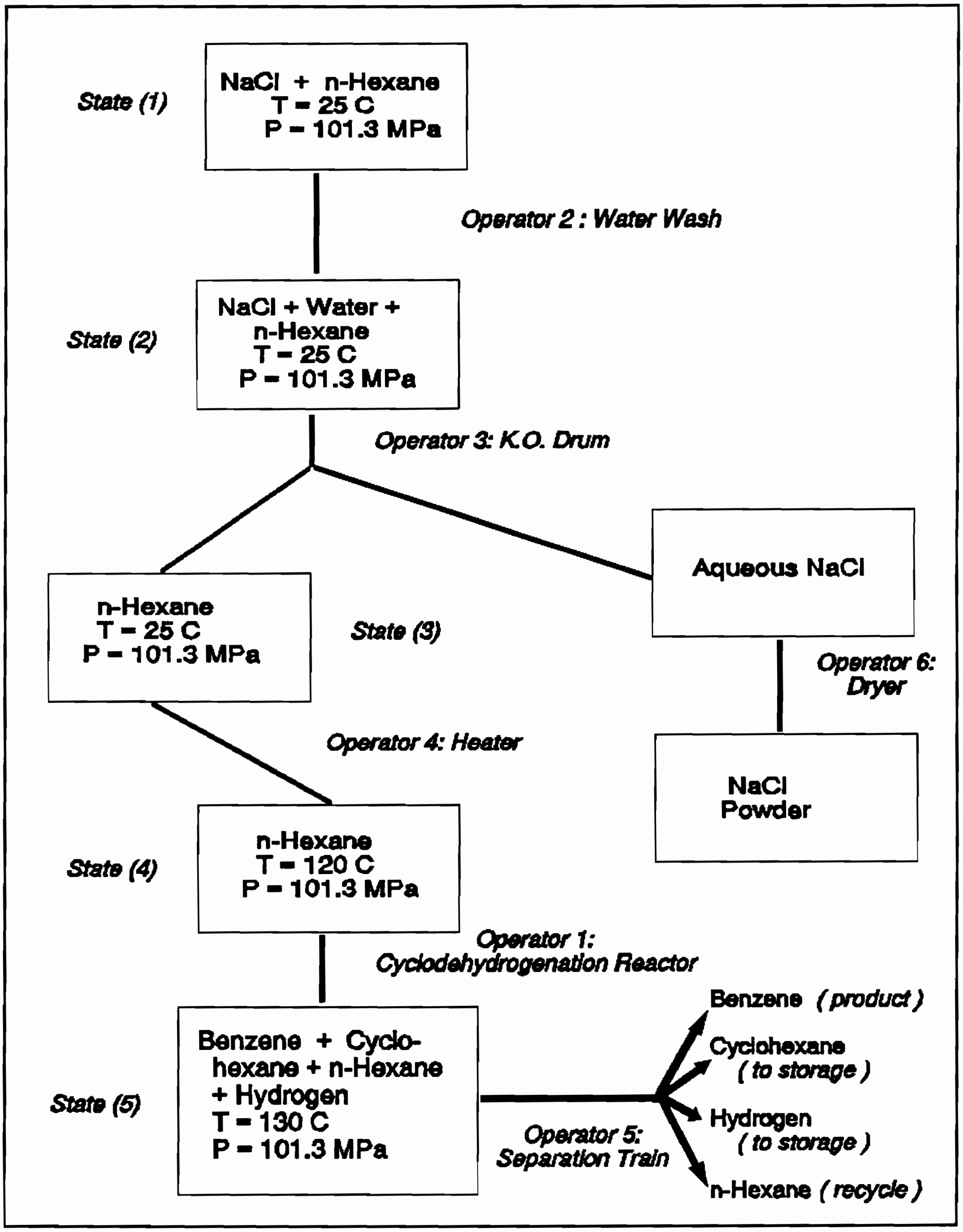


Figure 11.20. A means-end analysis for process synthesis.

Operator one, however, cannot be applied, because of the presence of NaCl in the stream. Therefore, we apply operator two:

> *__Operator Two:__ Remove inorganic salts from organic solvents with a water wash.*

We mix water with the mixture of n-hexane and sodium chloride, and now we are at state 2, an intermediate state. We apply operator three:

> *__Operator Three:__ Separate immiscible liquids with a settling vessel (knockout drum).*

Since the mixture forms two immiscible phases, we apply this operator to give two streams, aqueous NaCl and pure n-hexane. This operator gives us pure n-hexane (state 3), but we still cannot apply operator one. Why? Our feed temperature is too low. We apply operator four:

> *__Operator Four:__ Raise the temperature of the stream with a heater.*

We apply operator four to yield state 4, where we have pure n-hexane at the T and P required for operator one. We execute operator one, the cyclodehydrogenation reactor, to give us state 5. In state 5, we have a mixture of n-hexane (unreacted feed), cyclohexane (an intermediate species), hydrogen (a by-product), and benzene (the desired product). We apply operator five:

> *Operator Five:* *Separate mixtures of hydrogen and hydrocarbons with a separation train of absorption followed by stripping and ordinary distillation.*

The application of this operator yields hydrogen (to storage), n-hexane (recycle), cyclohexane (to storage), and benzene (product). To close, we still need to do something with our aqueous NaCl. We apply operator six:

> *Operator Six:* *Spray-dry aqueous inorganic salts to yield the corresponding inorganic crystal.*

We apply this operator to yield NaCl powder. Thus, we have applied means-end analysis to successfully go from a mixture of n-hexane and sodium chloride to yield benzene. In the process, we generated NaCl powder, as well as hydrogen and cyclohexane.

2. Comments on Means-End Analysis

Means-end analysis, in theory, is very powerful. It allows us to systematically apply operators and move from our current state eventually to the goal state. In practice, however, means-end analysis has been difficult to implement. Why? The operators. Historically, means-end analysis has used *general* operators. These general operators lack specificity. Therefore, in practical systems, it is impossible to move from the current state to the goal state without the presence of contextual, problem-specific operators. As shown in Figure 11.19, our operators are indeed very specific, and thus, this means-end analysis is much more practical.

Some means-end analysis programs have been written with very specific operators (Bratko, 1990, pp. 405-409). To a certain extent, these systems closely resemble rule-based expert systems with a means-end problem-solving approach. We develop a set of plans to get from where we are to where we want to be, and these plans are encapsulated in a rule-based framework. A typical means-end approach is as follows:

Step 1: Identify a set of potential goals that leads from the current state to the desired state. Call this set of goals **[List_of_goals]**.

Step 2: If all the goals are true, execute them and move to the desired state, and stop.

Step 3: For all goals in this set **[List_of_goals]** that fail, select one goal, called **goal_1**.

Step 4: Find a state, **intermediate_state_1**, such that **goal_1** is true.

Step 5: Execute an operator that moves us from the current state to **intermediate_state_1**.

Step 6: Apply **goal_1** to **intermediate_state_1**, thereby moving us to **intermediate_state_2**. Thus, **intermediate_state_2** is our new "current state." Recursively repeat steps 1-6 until the problem is solved.

E. Practice Problems

11.3.1 Consider Figure 11.12 (oil-pipeline network) and Figure 11.15 (AND-OR graph of the **a to t via g** side of the tree). Draw the AND-OR graph for the **a to t via h** side of the tree. Write all the relations in a Prolog program. Use comments to identify AND nodes and OR nodes.

11.3.2 Consider an expert system that is using probabilistic reasoning. We have the following set of rules in the system:

```
w:- x,y,z.
w:- a,b.
w:- c.
```

In addition, we know that:

x is true with a probability of 0.9
y is true with a probability of 0.7
z is true with a probability of 0.85
a is true with a probability of 0.8

b is true with a probability of 0.1

c is true with a probability of 0.9

Order both the sub-goals within each clause, and clauses themselves such that this system executes in the most efficient (fastest) way possible. Justify your decisions. The probabilities of **x**, **y**, **z**, **a**, **b**, and **c** are all independent of each other.

11.3.3 Suppose that we are synthesizing a chemical process flowsheet. We have an expert system using constraint satisfaction to help us. We give the expert system physical properties, plant specifications, and material and energy constraints. We ask the expert system for a set of possible configurations that obey all constraints. What do we conclude if the expert system determines that:

a. No possible configuration exists to the process synthesis problem?

b. Over one million configurations exist to the process synthesis problem?

11.3.4 Hokies Chemical Corporation is trying to schedule its batch manufacturing operations for the upcoming week. Hokies Chemical has five batch reactors, rl, r2, r3, r4, and r5, each with different capabilities:

- r1: can produce products a, b, and c
- r2: can produce product c only
- r3: can produce products c, d, and e.
- r4: can produce products a, b, c, and d
- r5: can produce products c, d and e

Hokies Chemical's work-week consists of five days (Monday through Friday), with three eight-hour shifts per day. Each batch reactor requires a full shift to make a product. For the upcoming work-week, Hokies Chemical needs to produce:

- 9 batches of product a
- 7 batches of product b
- 23 batches of product c
- 9 batches of product d
- 9 batches of product e

In addition, there are certain constraints on the operations:

- c1: If a reactor runs for three straight shifts, it must be shut down on the next shift for maintenance. Maintenance takes one full shift to complete.

- c2: If a reactor makes product a, then the following batch in that same reactor cannot be product b (cross-contamination problems).

- c3: If a reactor makes product c, then the following batch in that same reactor cannot be product d (cross-contamination problems).

Using constraint satisfaction, develop the Prolog program that will give Hokies Chemical possible schedules that satisfy all constraints. This program should be able to generate alternate schedules upon backtracking (i.e., if we enter the semicolon ; after the program reports a potential solution, the program will successfully generate an alternate schedule).

11.3.5 Develop a program that achieves all the objectives of the program in problem 11.3.4. This time, however, use a pure generate-and-test strategy to solve the problem. Make sure to allow for correct backtracking between the generator and the tester.

11.4 CHAPTER SUMMARY

- The three basic "domain-independent" search strategies are *depth-first*, *breadth-first*, and *best-first* searches. The best-first search is *heuristically* driven.

- Expert systems tend to work better if knowledge representation is emphasized over search techniques.

- The majority of expert systems represent knowledge in three major ways. We may have *rule-based*, *frame-based*, or *object-oriented* representations.

- Under a rule-based knowledge representation, we may use *forward chaining* or *backward chaining*. In addition, we may use logic-based approaches such as fuzzy logic.

- A precursor to a frame-based knowledge representation is the *semantic network*. An important concept of semantic networks is the principle of *inheritance*, where lower-level nodes inherit properties from higher-level nodes via the **isa** link.

- A typical frame-based knowledge representation in Prolog will rely on many "utility" procedures for processing the frames. Some of the utility procedures that we introduced include :

- **inherit(** *Frame*, *Slot*, *Interpretation*, *ValueList*)**.**
- **if_procedure(** *Frame*, *Slot*, *Interpretation*, *ProcedureList*)**.**
- **find(** *Frame*, *Slot*, *Interpretation*, *Value*)**.**
- **delete_frame(** *Frame*)**.**
- **delete_frame(** *Frame*, *Slot*)**.**
- **delete_frame(** *Frame*, *Slot*, *Interpretation*)**.**
- **delete_frame(** *Frame*, *Slot*, *Interpretation*, *Value*)**.**
- **add(** *Frame*, *Slot*, *Interpretation*, *Value*)**.**
- **modify(** *Frame*, *Slot*, *Interpretation*, *NewList*)**.**

- Implementing an object-oriented program is relatively simple once we have a frame-based system operational. Whenever a frame is accessed, we simply check to see if the slot interpretation to be equal to **procedure**. If it is, we execute the procedure.

- Other search strategies include *AND-OR analysis*, *constraint satisfaction*, *generate-and-test*, and *means-end analysis*.

  - *AND-OR analysis* operates using problem decomposition into AND nodes and OR nodes. Its main advantages are its ease of understanding, and its ability to reduce large problems down to tolerable sub-problems.

- *Constraint satisfaction* operates by pruning or filtering the state space of potential solutions that do not satisfy current constraints.

- *Generate-and-test* operates by using a *generator* to develop a set of possible solutions, and a *tester* to assess if any of these candidates are acceptable. If no answer is found, the system backtracks to the generator in search of alternatives.

  - One variation of generate-and-test that is targeted to improve efficiency is *plan-generate-test*, which uses a *planner* to restrict the search space. The generate-and-test stages then explore this restricted state space.

  - Another variation of generate-and test is *hill-climbing*, where the tester does not simply answer yes or no, but provides guidance as a form of feedback to the generator.

- *Means-end analysis* operates by applying specific operators in an attempt to move from our current state to the goal state. When we apply the operator and move to the intermediate state, we reduce the problem to a sub-problem of getting from the intermediate state to the goal state. We hope that this sub-problem is easier to solve. Importantly, we can recursively implement this process of assessing where we are and where we want to be, and then applying operators.

A LOOK AHEAD

We now have a firm grasp of both Prolog and artificial intelligence (AI). We see how Prolog fits into the world of AI. With this understanding, we close Part One of the book, and move on to Part Two. Part Two is a case study of EXSEP, the EXpert system for SEParation synthesis. We shall see knowledge representation and Prolog in action to solve the complex problem of process synthesis and multicomponent separations.

REFERENCES

Barr, A. and E. A. Feigenbaum, *The Handbook of Artificial Intelligence*, Volumes I, II and III, William Kaufman Inc., Los Altos, CA (1981).

Barr, A., P. R. Cohen and E. A. Feigenbaum, *The Handbook of Artificial Intelligence*, Volumes IV, Addison-Wesley, Reading MA (1989).

Bratko, Ivan, *Prolog Programming for Artificial Intelligence*, pp. 255-315, 406-409, Addison-Wesley, Reading, MA (1990).

Ernst, G.W. and A. Newell, *GPS: A Case Study in Generality and Problem Solving*, Academic Press, New York, NY (1969).

Hayes-Roth, F., D. Waterman, and D. Lenat, *Building Expert Systems*, Addison-Wesley, Reading, MA (1983).

Mueller, R.A. and R.L. Page, *Symbolic Computing with LISP and Prolog*, John Wiley & Sons, New York, NY (1988).

Newell, A. and H.A. Simon, "GPS, a Program that Simulates Human Thought," in *Computers and Thought*, E. Feigenbaum and J. Feldman (Eds.), McGraw-Hill, New York NY (1963).

Rich, Elaine, *Artificial Intelligence*, McGraw Hill, New York, NY (1991).

Rich, E. and K.Knight, *Artificial Intelligence*, second edition, McGraw Hill, New York, NY (1991).

Rowe, Neil C., *Artificial Intelligence Through Prolog*, pp. 307-348, Prentice-Hall, Englewood Cliffs, NJ (1988).

Schalkoff, Robert J., *Artificial Intelligence: An Engineering Approach*, McGraw-Hill, New York, NY (1990).

Schnupp, P., C.T. Nguyen Huu and L.W. Bernhard, *Expert Systems Lab Course*, pp.121-173, Springer-Verlag, New York, NY (1989).

Taylor, W.A., *What Every Engineer Should Know About AI*, MIT Press, Cambridge, MA (1988).

Waterman, D., *A Guide to Expert Systems*, Addison-Wesley, Reading, MA (1986).

12

INTRODUCTION TO EXSEP

Part II of this book examines, using a detailed case study, what is required to develop an expert-system application in science or engineering.

In developing an expert system, we must:

(1) properly represent the knowledge;

(2) choose the computational tools to achieve the goals; and

(3) organize the knowledge and the search in a coordinated fashion.

The system we will focus on is the *EXpert system for SEParation synthesis*

(EXSEP), used for chemical-engineering flowsheet development. Given a feed mixture of volatile chemicals and a set of desired products resulting from separation processes, EXSEP develops the flowsheet for a separation system to yield those products technically and economically.

In this part of the book, we assume that the reader has a general knowledge of chemistry and chemical engineering. Multicomponent separation science can get complex. However, we have chosen a problem-representation scheme that *does not require advanced skills in chemistry and mathematics.* Admittedly, developing the operating heuristics for EXSEP required a lot of "behind the scenes" chemistry and mathematics. But the end results, the heuristics themselves, are relatively simple for a practicing engineer, or even an upper-level undergraduate student to understand.

12.1 THE APPLICATION: SEPARATION-FLOWSHEET DEVELOPMENT AND MULTICOMPONENT SEPARATION SEQUENCING

A. Separation-System Synthesis or Separation-Flowsheet Development

The separation of multicomponent mixtures into desired products is of critical importance to the chemical industry. Separations are used to process and purify raw materials, intermediates, and finished products. Common examples include:

- fractionation of crude oil
- recovery of reactor effluent streams
- removal of environmentally damaging materials
- purification of finished products.

Ordinary distillation is the most common separation method in the chemical process industry (CPI) today. Distillation uses heat, which is an energy-separating agent (ESA). Liquid and vapor contact occurs, and the more volatile components partition into the vapor. As the vapor travels to the top of the distillation column, it gets increasingly richer in the more volatile components.

The design of a cost-effective and thermodynamically feasible distillation column is a well-developed process. With the advent of computer-aided design (CAD), engineers have access to the rigorous design calculations that allow them to optimize the problem solution.

A more complex, and less understood area, however, is *separation-system synthesis*. Here the goal is to identify plausible process sequences that are:

(1) technically feasible and capable of achieving the given separation tasks, and
(2) safe, reliable, and economically attractive.

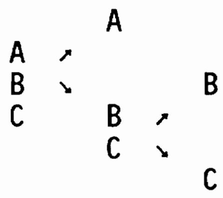
Chemical engineering research has focused on the systematic synthesis of separation process flowsheets, particularly those including distillation, largely because the typical chemical plant uses more energy for distillation than for any other type of process.

B. Multicomponent Separation-Sequencing Problem

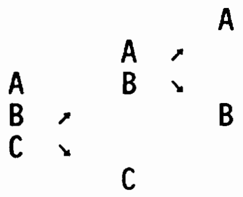
Multicomponent separation sequencing, or the selection of the best method and sequence for the separation, is an important design problem in separation-system synthesis. To solve this problem, we often first rank the components to be separated according to some appropriate physical and/or chemical properties such as relative volatility or solubility in water. This ranked list (RL) gives the component name and its property ranking relative to other components in the mixture. With this list, we can treat a separation step as a list-splitting operation, that separates components above a certain property value from components below that value.

1. Sharp Separation

Thus, the generation of a separation sequence involves selecting an appropriate property list and the choosing the component splits within that list. For example, to find sequences for separating a three-component mixture of A, B, and C into pure components by two ordinary distillation columns, we first arrange the components according to volatility from the most volatile (A) to the least volatile (C). We then examine the different splits in the two columns. We can split A/BC in the first column and B/C in the second column. Or, we can split AB/C in the first column and A/B in the second. Schematically, these sequences are:



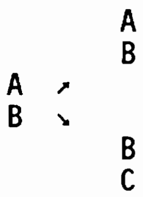
or



In this example, each component appears in one and only one product stream; this type of separation, which yields products with non-overlapping components, is a *high-recovery or sharp separation.*

2. Sloppy Separations

In industrial practice, we sometimes want components to appear in two or more product streams. For example, consider another a scheme for separating the same three-component mixture considered above:



This type of separation, resulting in products with overlapping components, is called a *sloppy* or *nonsharp separation.*

Why do we use sloppy separations? Consider a typical chemical process consisting of a reaction system and a separation system. The separation system generally involves reactant recovery, product separation, and byproduct separation. In the development of the optimum flowsheet for such a reaction process, we often treat reaction conversion as a key parameter. Economically, the high reactor costs and significant loss of selectivity associated with high conversions are balanced against the large separation costs associated with low conversions. This trade-off suggests that we may not need to purify unconverted reactants to a high level before recycling them to the reactor system. In other words, sloppy separations may be sufficient for many processes. Therefore, in the development of separation process flowsheets, we need to consider not only sharp separations, but sloppy separations as well. An effective and flexible technique for multicomponent separation sequencing must incorporate all three types of separation schemes, namely, all-sharp, all-sloppy, and mixed (both sharp and sloppy) sequences.

For excellent discussion of background information related to multicomponent separation-sequencing problems, the reader may refer to the texts by Rudd et. al. (1973, pp. 155-208 and 288-295), King (1980, pp. 710-720), Henley and Seader (1981, pp. 527-55), and reviews by Westerberg (1985) and Liu (1987).

C. The Need for Efficient Problem-Solving Techniques for Separation Sequencing.

To appreciate the need for some simple, practical methods for synthesizing multicomponent separation sequences, let us consider the number of *theoretically possible* sequences for separating a multicomponent mixture into pure-component product streams (Henley and Seader, 1981, pp. 530-533).

1. The number of theoretically possible sequences, S_N, for separating an N-component mixture into N pure-component products by one separation method only is:

$$S_N = \frac{[2(N-1)]!]}{N!(N-1)!} \tag{12.1}$$

2. The number of theoretically possible sequences, S, for separating an N-component mixture into N pure-component products by T separation methods is:

$$S = T^{N-1} \bullet S_N = T^{N-1} \frac{[2(N-1)]!}{N!(N-1)!} \tag{12.2}$$

Table 12.1 compares the number of possible sequences for T = 1 and T = 2. Separating a 10-component mixture into pure-component products using two methods approaches 10 million possible sequences. In view of the enormous number of alternatives available, we obviously need some *creative* methods to synthesize efficient sequences for multicomponent mixtures.

Table 12.1. Number of theoretically possible sequences for separating a N-component mixture into pure-component products with one (T=1) and two (T = 2) separation methods.

NUMBER OF COMPONENTS, N	THEORETICALLY POSSIBLE SEQUENCES	
	T = 1	T = 2
3	2	8
5	14	224
7	132	33,792
10	4,862	9,957,376

D. Techniques for Separation Sequencing

Researchers have developed a number of techniques for the systematic synthesis of multicomponent separation sequences. The techniques fall into four basic categories:

(1) *Algorithmic or optimization approaches*- these approaches attempt to mathematically model and optimize the system. One of the earliest investigations of multicomponent separation sequencing involved the use of a well-known optimization technique called dynamic programming (Hendry and Hughes, 1972).

(2) *Thermodynamic approaches-* these methods approach process synthesis from a thermodynamic standpoint. They use known principles of thermodynamics to synthesize the sequence in an energy-efficient fashion (e.g., Gomez-Munoz and Seader, 1985).

(3) *Heuristic approaches-* these methods use general rules of thumb (heuristics), to synthesize the sequence. The heuristics can focus on process characteristics, material properties, thermodynamics, etc. (Rudd et. al., 1973, pp. 155-208).

(4) *Evolutionary approaches-* these methods first develop an initial sequence, and then make systematic improvements to it based on evolutionary rules (e.g., Stephanopoulos and Westerberg, 1976).

These four categories, however, do not represent exclusive boundaries. Some methods combine two or more of these approaches, as does the heuristic-evolutionary method by Seader and Westerberg (1977).

Table 12.2 lists the advantages and disadvantages of the heuristic and evolutionary methods, and compares them with optimization methods. The table excludes thermodynamic approaches, since they are the relatively less developed, and consequently less understood.

Table 12.2. A comparison of three common methods for the synthesis of multicomponent separation sequences.

ADVANTAGES	DISADVANTAGES
HEURISTIC METHODS	
Straightforward to apply by hand Do not require mathematical background or computational skill. Can easily generate an initial sequence for other methods.	Heuristics often contradict or overlap one another. Requires a strategy to implement (which heuristics are used first?)
EVOLUTIONARY METHODS	
May reveal new sequences through evolutions.	Need an initial sequence generated by other methods. Require a strategy to implement (which evolutionary rules are used first?). Need quantitative performance criteria (may involve design calculations and equipment costing). Limited by problem size (many sequences need to be compared).
OPTIMIZATION METHODS	
Can be computerized. Can easily find sub-optimal sequences.	Ignore qualitative information (corrosive, hazardous, and cryogenic properties of feed). Dependent on cost-equation. Limited by problem size.

E. Separation Heuristics and EXSEP

Heuristic rules to guide separation sequencing have long been available. Liu (1987) gives a survey of heuristics for the synthesis of multicomponent separation sequences published since 1947. Examples of these separation heuristics are:

- Favor ordinary distillation.
- Avoid vacuum and refrigeration.
- Remove valuable or desired product last.
- Perform difficult separations last.
- Carry out easy separations first.
- Remove the most plentiful component first.

Section 13.6 discusses useful separation heuristics in more detail.

At this point, we should note that most of the heuristics reported thus far offer no indication of the conditions under which we should favor a specific heuristic. They simply say, "All other things being equal, favor the sequence which ...." Furthermore, the heuristics often contradict or overlap one another. For instance, consider a material present in excess but that is difficult to separate. The heuristic "Remove the most plentiful product first" suggests early removal of this material. However, the heuristic "Perform difficult separations last" favors late removal of this same material. These two heuristics contradict each other, and it is unclear how to proceed.

The development of rank-ordered heuristics (Seader and Westerberg, 1977; Nath and Motard, 1981; Nadgir and Liu, 1983) has resolved many of the conflicts and thus enhanced the applicability of heuristic methods. In rank-ordered methods, the heuristics are applied one-by-one in the order specified by the method. If one heuristic is not important or not applicable to a given synthesis problem, we move on to the next one.

Perhaps the most straightforward rank-ordered heuristic method is that proposed by Nadgir and Liu (1983). EXSEP uses this heuristic method because it is both accurate and flexible. The ranking is a valuable asset when developing an expert system, because it facilitates easy resolution of any contradictions.

Cheng and Liu (1988) extended upon the method of Nadgir and Liu applied to sloppy separations. Liu et. al. (1990) further enhanced the method, including sharp, sloppy, and both sharp and sloppy (called mixed) sequences; EXSEP uses this version of the method.

Chapters 13-15 discuss in detail the method used by EXSEP to solve multicomponent separation-sequencing problems. Specifically, we discuss the chemical-engineering, general user, and AI perspectives of EXSEP. But before we approach these topics, let us first discuss expert-system development more broadly.

12.2 INTRODUCTION TO EXPERT SYSTEM DEVELOPMENT

This section introduces expert-system development, focusing on effective development techniques, EXSEP's development plans, and the limitations of expert systems. In section 16.2, we shall return to the topic of expert-system development in more detail, considering when development is possible, justified, and appropriate.

A. Common Development Techniques

Developing expert systems typically involves two types of people:

- *Domain experts*- scientists or engineers who are experts in the field that the system is targeted to model.
- *Knowledge engineers*- computer scientists who take knowledge from the domain expert, characterize it, and ultimately write the computer program.

The process of meeting with the domain expert, "capturing" his knowledge, and programming it into the computer is called the *knowledge-acquisition process*.

The knowledge-acquisition process is frequently the longest and most difficult part of the development. Some difficulties associated with it are:

(1) *Poor communication*

Source: The knowledge engineer and the domain expert fail to understand each other's objectives and expertise. They communicate poorly.

Result: Rule-quality suffers and the expert system does not perform accurately.

(2) *Poor knowledge representation*

Source: This problem arises in one of two ways. The knowledge engineer may choose the most familiar representation, even though it may not be best for the problem. Or, the domain expert may understand his field very well, but not understand AI. Consequently, he does not analyze nor express the *technical* aspects of the problem in a form that takes advantage of known AI techniques.

Result: Both accuracy and efficiency of the expert system may suffer.

(3) *Not enough time*

Source: The domain expert is busy in his field. He cannot devote enough time to expert system development and does not see the benefits of the time expended.

Result: The project gets off the ground slowly, and people lose interest and enthusiasm. The project may even be canceled.

B. EXSEP's Development Plan

To avoid some of the difficulties associated with building expert systems, we developed EXSEP differently. Instead of having separate individuals as knowledge engineer and domain expert, a chemical engineer filled both roles. This kind of unified approach is becoming increasingly popular in commercial expert-system development, as chemical engineers become more proficient in artificial intelligence.

Executing the project this way has two key advantages:

(1) *Difficulties between the knowledge engineer and domain expert are eliminated.*
The difficulties discussed above (communication, time allocation, knowledge representation, etc.), are all avoided. Once the project is moving, it progresses quickly.

(2) *Knowledge representation is usually better.* A solid understanding of both the domain (in this case, multicomponent separations) and expert systems produces an effective problem representation that is "natural" for the expert system. It also allows us to incorporate domain-specific search techniques. As a result, the system solves problems accurately and efficiently.

The approach has one disadvantage, however. Since the development team lacks an experienced knowledge engineer with a strong computer science background, we lose some potential insight and experience. Consequently, we have a more pronounced learning curve for expert-system development, especially on the more subtle aspects. Locating available expert system tools takes more time, and the final system may lack some sophistication.

Overall, we feel the EXSEP project proceeded faster than it would have if a typical "knowledge engineer and domain expert" approach was taken. Perhaps this is why the technique of having a single individual perform the combined role of knowledge engineer and domain expert is catching on commercially.

C. Limitations of Expert Systems

The goals of EXSEP focus more on chemical engineering rather than on computer science. Our fundamental goal of EXSEP was to develop a knowledge representation for a complex chemical-engineering design problem that would allow an expert system to solve the problem accurately and efficiently. To achieve this goal, we had to recognize the inherent limitations of expert systems. Expert systems can be very powerful tools, but the wise developer will also know what they *cannot* do. Currently, expert systems do not effectively:

- handle inconsistent knowledge,

- handle some aspects of *nonmonotonic reasoning*,
- refine their own knowledge base,
- process in mixed representation schemes, and
- perform knowledge acquisition automatically.

A detailed discussion of these areas follows.

1. Inconsistent Knowledge

Because their knowledge bases rely upon rules, expert systems cannot handle inconsistent knowledge. They do not reason from basic scientific principles; therefore, they cannot recognize when a situation violates those principles. Moreover, if the knowledge is incorrect, the expert system may not notice. Essentially, expert systems are vulnerable to the "garbage in, garbage out" syndrome.

2. Monotonic and Nonmonotonic Reasoning

An important aspect of knowledge is *monotonic* versus *nonmonotonic reasoning*. As an expert system progresses through a problem, it can add new statements, intermediate conclusions, and new information to the database or inference chain. The system is considered *monotonic* if none of the additions will invalidate a previous conclusion. If inconsistencies do develop or previous conclusions become invalid, then the system is *nonmonotonic*.

As an example, let us consider an expert system doing material allocation and scheduling in a chemical plant. The system decides that sending the entire supply of isobutylene to the isobutylene rubber unit will maximize profits for the month. It makes this decision and stores it in the database. Later in the program, it performs the analysis for the t-butyl alcohol unit, which requires isobutylene as the feedstock. Without enough feed, the unit will have to be shut down for the month.

Using *monotonic reasoning*, the expert system would shut down the t-butyl alcohol plant for the month, because the isobutylene has already been allocated. To avoid contradicting a previous conclusion (send all isobutylene to the rubber unit), it has no choice but to shut the alcohol unit down.

Nonmonotonic reasoning, however, may allocate isobutylene to the t-butyl alcohol unit even if none is currently available. This step invalidates the decision to send all the isobutylene to the rubber unit. The conclusion violates the material-balance constraint (we have over-allocated isobutylene), and the resulting conflict requires some resolution. To resolve the material balance constraint violation, the expert system would have to execute some type of *consistency enforcer* to maintain the material balance. This enforcer would adjust the allocation of isobutylene until it satisfied all the essential constraints.

Table 12.3 compares the advantages and disadvantages of both monotonic and nonmonotonic reasoning.

Table 12.3. Advantages and disadvantages of monotonic and nonmonotonic reasoning.

MONOTONIC REASONING

Advantages

- Simple; easier program development
- Fast reasoning and efficient program execution
- No need for checks when information is added to database
- No need to remember the basis upon which conclusions rest

Disadvantages

- Awkward for handling incomplete information
- Weak at handling dynamic, changing situations
- Weak at default reasoning
- Cannot temporarily accept assumptions that currently contradict but may be reconciled later

NONMONOTONIC REASONING

Advantages

- Handles those problems that cannot be represented monotonically
- Good at default reasoning and handling incomplete information
- Can generate a complete solution that requires temporary assumptions about a partial solution

Disadvantages

- Requires a consistency enforcer after taking actions, which wastes time and effort
- Frequently lacks a clear way to reconcile contradictions when they occur
- Inferences can get complex, out of control, and difficult to understand
- Must remember justifications for each statement

As a general rule of thumb, if an engineering problem can be represented monotonically, do so. Do not go into nonmonotonic reasoning unless it is essential. Although some tools are available for nonmonotonic reasoning, these tools are not well developed. The only exception may be default reasoning (using, for instance, inheritance in frame-based expert systems), discussed in section 9.2C.

3. Refining Their Own Knowledge Base

Expert systems are not well-suited for refining their own knowledge base. The reason? Most of the rules in expert systems come from experiential knowledge. The expert system has rules in its database, but does not understand the fundamental physical principles underlying the rules. Without this understanding, expert systems have difficulty refining their own knowledge base.

4. Processing in Mixed-Representation Schemes

Expert systems apply best to a single knowledge representation scheme, and generally do not perform well in mixed-representation schemes. What do we mean by "mixed-representation ?" Mixed representation combines two or more approaches to solving problems. Usually, these approaches require completely different problem-solving techniques. For example, a mixed-representation scheme might require both predicate logic (discussed in section 10.2D) and Boolean algebra (introduced in section 10.2C),and a single expert system could not easily handle both. These two methods are so different that even trying to mesh the two representations into a single system is nearly impossible. Instead, expert systems work most effectively with a unified representation scheme. The challenge is developing a representation that solves the problem accurately and efficiently.

5. Automated Knowledge Acquisition

The *knowledge-acquisition process*, discussed in 12.3A, normally involves the knowledge engineer developing a program based on information from the domain expert. In theory at least, we could develop an expert system with an intelligent user interface capable of acquiring knowledge directly from the domain expert. The intelligent user interface could take in the knowledge, create rules, and assert these rules into the database, thereby automating the knowledge acquisition process.

Automated knowledge acquisition is a much sought-after goal in artificial intelligence. Unfortunately, this area is very challenging, and expert systems cannot yet perform automated knowledge acquisition effectively.

6. Additional Problems with Expert Systems

Rich and Knight (1991, pp. 556-557) discuss some additional problems facing expert systems today. These difficulties are:

- *Brittleness*- Because expert systems only have access to highly specific domain knowledge, they cannot fall back on more general knowledge when the need arises. For, example, suppose that we make a mistake in entering data for a medical expert system, and we describe a patient who is 130 years old and weighs 40 pounds. Most systems would not be able to guess that we reversed the two fields since the values are not very plausible.

- *Lack of Meta-Knowledge-* Meta-knowledge is "knowledge about knowledge." Expert systems do not have very sophisticated knowledge about their own operation. They typically cannot reason about their own scope and limitations, making it even more difficult to deal with the brittleness problem.

- *Knowledge Acquisition-* Despite the recent development of the software tools to aid the process, acquisition still remains the major bottleneck in applying expert system technology to new areas.

- *Validation-* Measuring the performance of an expert system (i.e., "quality control") is difficult because we do not know how to quantify the use of knowledge. Certainly it is impossible to present formal proofs of correctness for expert systems. One thing we can do is pit these systems against human experts on real-world problems. For example, MYCIN participated in a panel of experts in evaluating ten selected meningitis cases, scoring higher than any of its human competitors (Buchanan, 1982).

7. Summary

The areas discussed above represent areas that expert systems do not perform well in. These areas are some of the key challenges for the future for expert systems. A great deal of research is underway to advance expert system performance and meet these challenges. Expert systems will probably improve in these areas over the next decade, and some of the research is already bearing fruit, such as in nonmonotonic reasoning.

But given the current state of the art, these areas still form inherent limitations of expert systems. If we want to build an expert system and save ourselves a lot of time and headache, we should avoid the above areas if at all possible.

EXSEP's development plan intentionally targeted the strengths of expert systems and avoided the areas of known weaknesses. We specifically sought a plan that:

(1) had a consistent knowledge representation with monotonic reasoning;

(2) only asserted and retracted deduced facts, and avoided refining its own knowledge base through assertion and retraction of dynamically created rules, and did no automatic

knowledge acquisition; and

(3) used rule-based representation exclusively.

While these restrictions do not guarantee that the resulting expert system will be successful, they certainly help its chances.

These restrictions place a *greater* challenge on the knowledge-representation scheme. It must be thorough and technically accurate to perform well, which presents quite a challenge for a complicated problem such as multicomponent separation sequencing. But, as we shall see in the next chapter, the representation that we developed does achieve the above goals.

12.3 CHAPTER SUMMARY

EXSEP is the EXpert system for SEParation synthesis. It is a rule-based expert system using monotonic reasoning. Given a multicomponent mixture, EXSEP will recommend a technically feasible and economically attractive separation sequence for separating the multicomponent mixture into desired products.

The main focus of the EXSEP project was to develop a knowledge-representation scheme for the complex problem of multicomponent separations that was compatible with rule-based expert systems. EXSEP can perform reasoning on both sharp and sloppy separations.

We placed some restrictions on EXSEP in the early stages to improve its chances of success. EXSEP focused on the current strengths of expert systems and avoided potential pitfalls such as inconsistent knowledge, refining its own knowledge base, automated knowledge acquisition, and processing in mixed-representation schemes.

A LOOK AHEAD

We have introduced separation sequencing. In the next chapter, we shall discuss in-depth the knowledge representation used in EXSEP. The next chapter's importance cannot be understated, since knowledge representation is critical to the success of an expert-system implementation.

REFERENCES

Cheng, S.H., and Y.A. Liu, "Studies in Chemical Process Design and Synthesis. 8. A Simple Heuristic Method for the Synthesis of Initial Sequences for Sloppy Multicomponent Separations", *Ind. Eng. Chem. Res.*, 27, 2304 (1988).

Gomez-Munoz, A. and J.D. Seader, "Synthesis of Distillation trains by Thermodynamic Methods," *Comput. Chem. Eng.*, 9, 311 (1985).

Hendry, J.E. and R.R. Hughes, "Generating Separation Process Flowsheets," *Chem Eng. Prog.*, 68, No. 6, 71, (1972).

Henley, E.J. and J.D. Seader, *Equilibrium-Stage Separation Operations in Chemical Engineering*, John Wiley & Sons, New York, NY (1981).

King, C.J., *Separation Processes*, 2nd Ed., McGraw-Hill, New York, NY (1980).

Liu, Y.A., H.A. McGee, and W.R. Epperly (Eds.), *Recent Developments in Chemical Process and Plant Design*, John Wiley & Sons, New York, NY (1987).

Liu, Y.A., T.E. Quantrille, and S.H.Cheng, "Studies in Chemical Process

Design and Synthesis: 9. A Unifying Method for the Synthesis of Multicomponent Separation Sequences with Sloppy Product Streams", *Ind. Eng. Chem. Res.*, **29**, 2227 (1990).

Nadgir, V.M. and Y.A. Liu, "Studies in Chemical Process Design and Synthesis: Part V. A Simple Heuristic Method for Systematic Synthesis of Initial Sequences for Multicomponent Separations", *AIChE J.*, **29**, 926 (1983).

Nishida, N., G. Stephanopoulos, and A.W. Westerberg, "A Review of Process Synthesis", *AIChE J.*, **27**, 321 (1981).

Stephanopoulos, G. and A.W. Westerberg, "Studies in Process Synthesis-II. Evolutionary Synthesis of Optimal Process Flowsheets," *Chem. Eng. Prog.*, **62**, No. 2, 75 (1964).

Westerberg, A.W., "The Synthesis of Distillation-Based Separation Systems", *Comput. Chem. Eng.*, **9**, 421 (1985).

13

CHEMICAL ENGINEERING PERSPECTIVE OF EXSEP

In this chapter, we introduce EXSEP from a chemical engineering perspective. The purpose of this chapter is to understand the chemical engineering problem representation that EXSEP utilizes.

13.1 INTRODUCTION

This chapter describes the chemical engineering perspective of the EXSEP (EXpert system for SEParation sequencing). The primary objective of the EXSEP is to synthesize alternative flowsheets for separating a multicomponent feed into several *sloppy* product streams, in which some feed components appear in two or more product streams.

A. Example 1: A Four-Component Separation

Table 13.1 outlines example 1, a problem of separating a four-component mixture of hydrocarbons (components A to D) into four sloppy product streams (products P1 to P4). Figures 13.1 a-c then illustrate three different separation sequences that give the desired product streams. Note that in distillation, we frequently refer to the term "key components." For sharp separations, the *light-key* and *heavy- key* components are defined as (Coulson et al., 1980): "The light key (LK) is the lightest component appearing in the bottoms, and the heavy key (HK) is the heaviest component in the distillate (or overhead)."

Figure 13.1a represents an *all-sharp sequence*, S1, consisting of three sharp (S) separations in which each component appears almost completely in one and only one product. Sharp separations correspond to very high recovery fractions of the light key in the overhead (denoted by d_{LK}) and of the heavy key in the bottoms (denoted by b_{HK}), that is,

Table 13.1. Feed and product specifications in example one*.

Desired product streams	Component flow rate (mol/hr) A	B	C	D	Product flow rate (mol/hr)
P4	0	0	0	15	15
P3	0	0	20	10	30
P2	10	12.5	0	0	22.5
P1	15	12.5	5	0	32.5
Component flow rate (mol/hr)	25	25	25	25	100

* Data taken form Nath (1977). Component A, B, C, and D are, respectively, n-butane, n-pentane, n-hexane, and n-heptane with normal boiling points of -0.5, 36.1, 68.7, and 98.4 °C. For a feed mixture at 92.2 °C and 466.1 kPa, the equilibrium ratios are $K_A = 2.46$, $K_B = 1.00$, $K_C = 0.47$, $K_D = 0.21$.

$0.98 \leq d_{LK} \leq 1.0$ and $0.98 \leq b_{HK} \leq 1.0$.

Sequence S1 begins with the sharp split ABC/D in separator V3, followed by the sharp split AB/C in separator V2′, where the prime implies that there is bypass around the separator. Here, a portion of the feed bypasses the separator and forms a part of the overhead. If there is no bypass, as in split V3, we do not use the prime. The S1 sequence ends with a sharp split A/B in separator V1′′, where the double prime indicates two portions of the feed bypass the separator and form part of both the overhead and bottoms products.

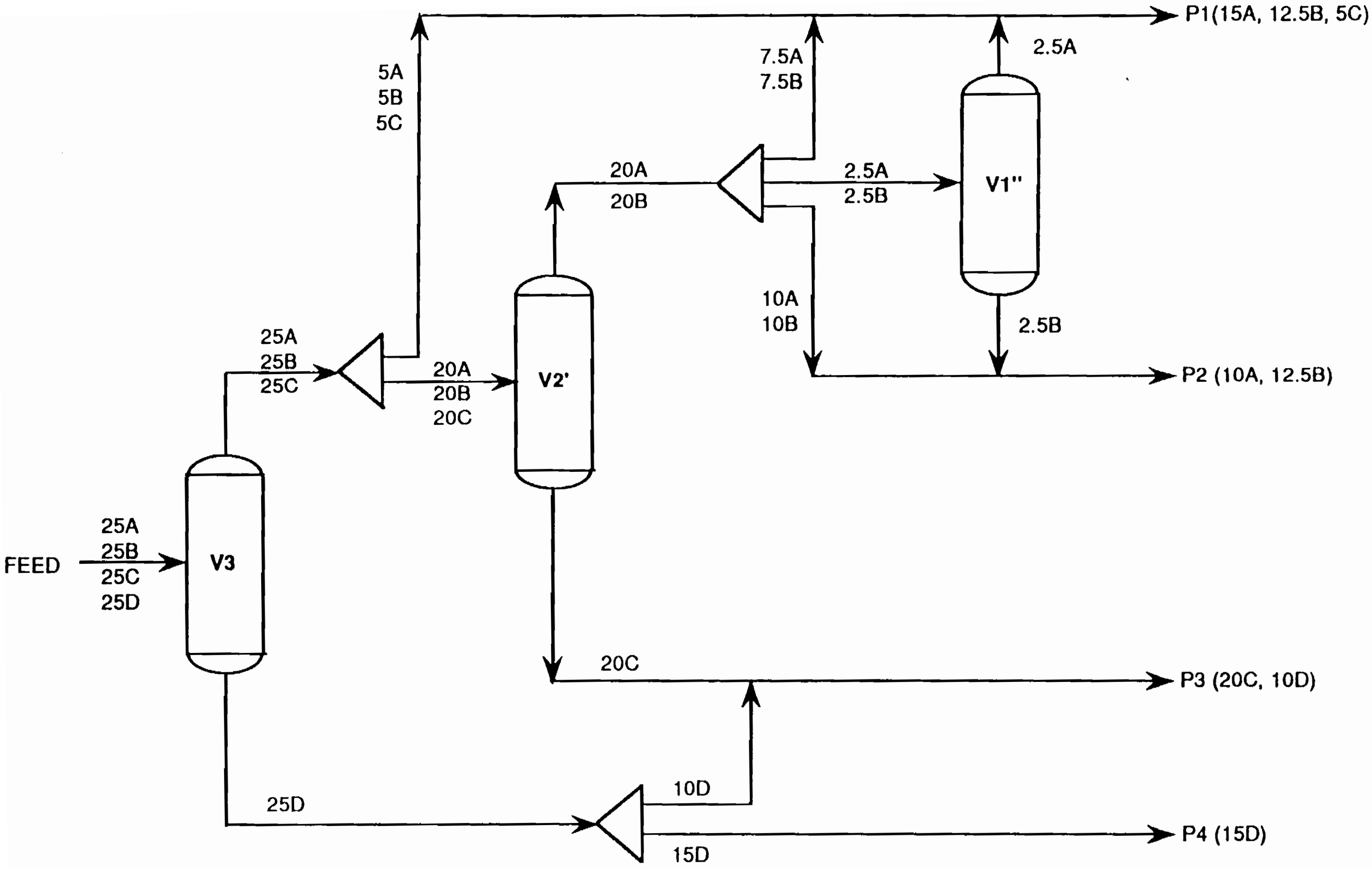


Figure 13.1a. Three-separator, all-sharp sequence S1.

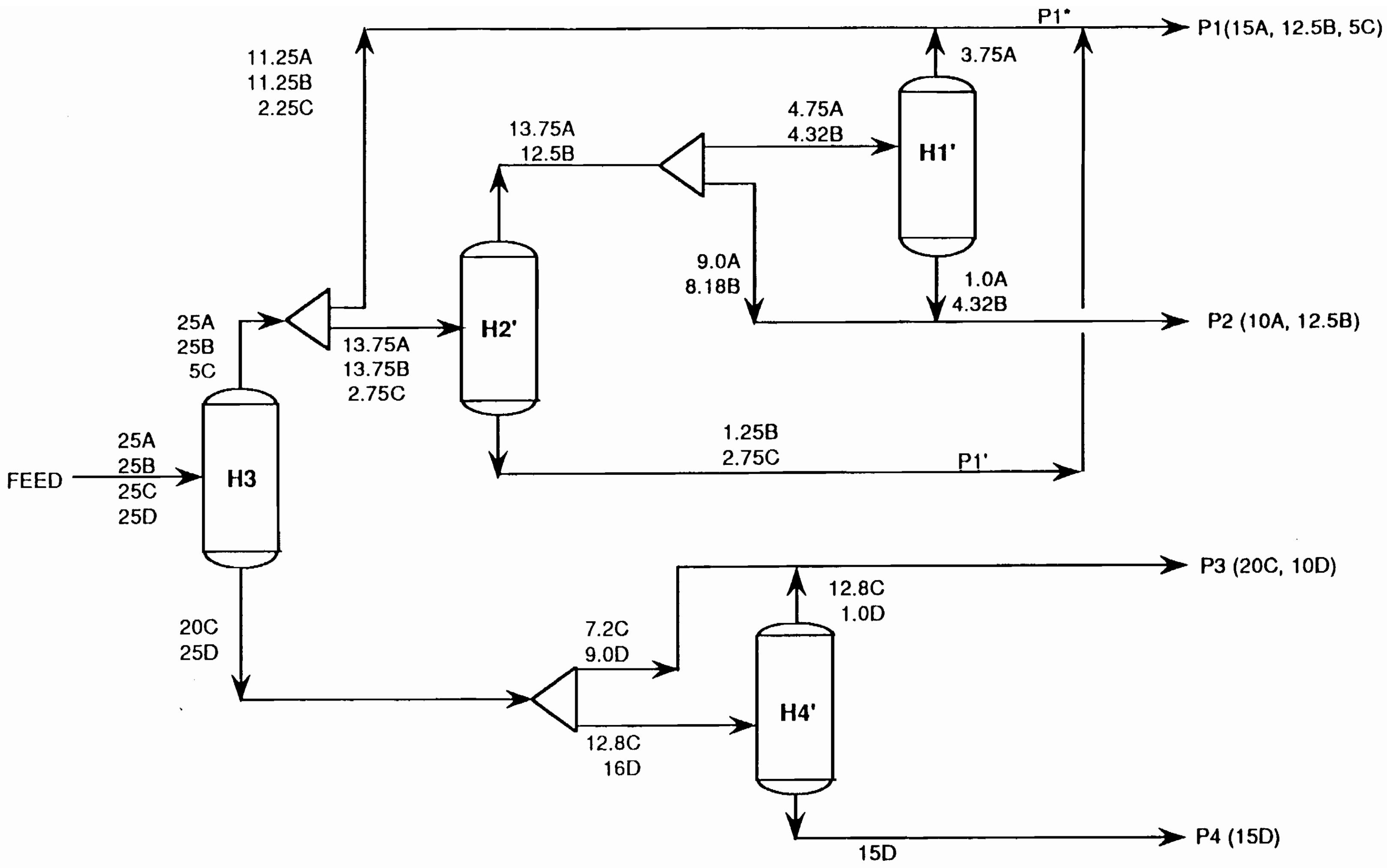


Figure 13.1b. Four-separator, all-sloppy sequence SL1.

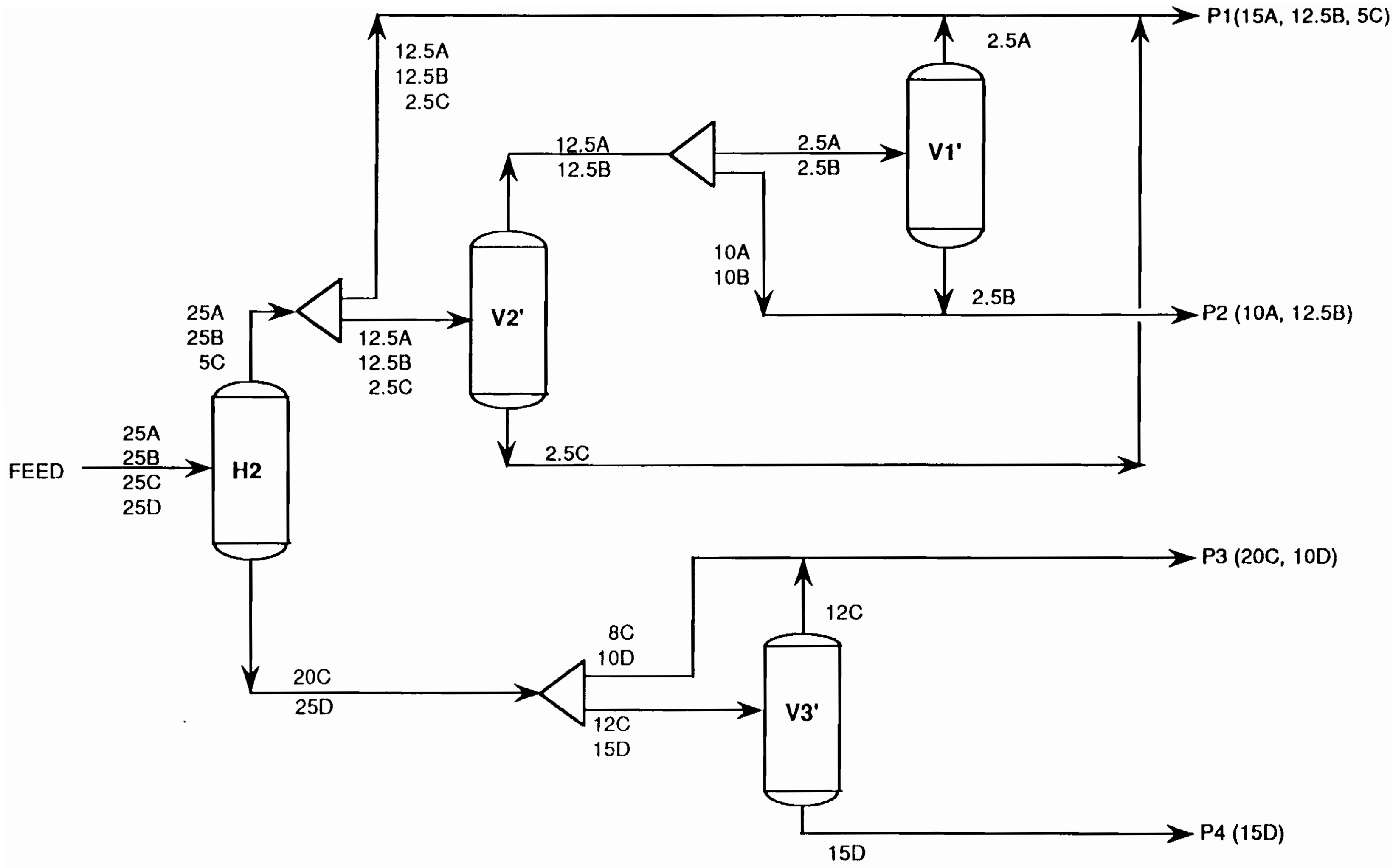


Figure 13.1c. Four-separator, mixed-separation sequence MS2.

S1 has three separators (S=3). This number corresponds to the apparent minimum number of separators, S_{min}, specified by the following equation (Cheng and Liu, 1988):

$$S_{min} = \min (C,P) - 1 \qquad (13.1)$$

where C is the number of components and P is the number of products. Thus,

$$\begin{aligned} S_{min} &= \min (C,P) - 1 \\ &= \min (4,4) - 1 \\ &= 4 - 1 = 3 \end{aligned}$$

Figure 13.1b shows a four-separator, *all-sloppy sequence*, SL1. This sequence consists of four sloppy (SL) or low-recovery separations, in which some feed components appear in two or more products. SL1 begins with the sloppy split ABC/CD in separator H3. The overhead from H3 goes to another sloppy separator H2'(AB/BC) and that overhead goes to H1'(A/AB), while the bottoms from H3 goes to H4'(CD/D). This all-sloppy sequence includes one more separator than S_{min} (=3). We only consider the synthesis of sloppy sequences that have at most one separator more than other the minimum-separator sequences, because less stringent component recovery fractions specified in sloppy separations may yield an all-sloppy sequence that costs less than a competing minimum-separator sequence.

Figure 13.1c depicts a four-separator, *mixed-separation sequence*, MS2, that utilizes *both sloppy and sharp separators*. This sequence begins with the sloppy split ABC/CD in separator H2. The overhead from H2 goes to a sharp separator, V2'(AB/C), whose overhead goes to another sharp separator V1'(A/B). The bottoms from H2 goes to a sharp separator, V3'(C/D). When properly designed, a mixed-separation sequence can be economically competitive compared to a minimum-separator, all-sharp sequence, or to an all-sloppy sequence with an equal number of separators.

We have introduced separation sequencing. To develop these sequences, we have sharp and sloppy separations at our disposal. Depending on which type of separator we use, we can develop all-sharp, all-sloppy, or mixed (both sharp and sloppy) sequences. In addition, throughout the entire flowsheet development process, we have the option of stream bypass at our disposal.

B. Challenges in the Synthesis of Process Flowsheets

The problem of multicomponent separation sequencing is to synthesize separation flowsheets with a minimum number or a nearly minimum number of all of separators, giving *all-sharp*, *all-sloppy*, and *mixed-separation* schemes that will achieve the given product specifications in a technically feasible and an economically attractive way. Combining both technical feasibility and economic attractiveness is a key challenge in chemical engineering design.

1. Synthesis versus Problem Decomposition

One challenge in using expert systems for engineering design is that design is a *synthesis process*. As shown in Part I of this book, we can readily program a Prolog expert system to solve problems via *decomposition* (see also section 15.B). However, synthesis problems are more challenging, and perhaps the most crucial step in designing an expert system for chemical process synthesis is developing simple and flexible tools to represent the synthesis problem. This chapter introduces a problem representation, called the *component assignment diagram (CAD)*, coupled with the *component assignment matrix (CAM)*, for conveniently synthesizing the desired separation sequences.

2. Technical and Practical Feasibility

Another challenge of engineering design concerns feasibility and practicality. First, a design must be technically feasible to be viable. In separation sequencing, for instance, a specified separation task must be thermodynamically feasible from a vapor-liquid-equilibrium standpoint. To address technical feasibility, we introduce a *separation specification table* (SST) to facilitate feasibility analysis of separation tasks based on product specifications.

Besides being feasible, though, a design must also be practical and economically attractive. Once all the thermodynamic constraints are satisfied, the engineer must capitalize on the feasible options to configure an economically attractive design. To ensure that a feasible sequence is synthesized economically, we use *rank-ordered heuristics*.

The following sections discuss these basic tools and concepts in detail. Before moving on, however, let us first outline some of our technical assumptions. Specifically, we assume that:

(1) *Normal distillation is appropriate* - no azeotropes form, and relative-volatility differences are high enough to make ordinary distillation the separation of choice (EXSEP will identify splits where ordinary distillation is inappropriate, however).

(2) *Relative volatility is constant* - relative volatility does not change with composition.

(3) *Only conventional columns will be considered* - EXSEP uses standard, single-feed, two-product distillation columns.

13.2 REPRESENTATION OF THE SYNTHESIS PROBLEM

A. Component Assignment Diagram (CAD) and Component Assignment Matrix (CAM)

We can conveniently represent the problem of synthesizing multicomponent separation sequences by using the *component assignment diagram (CAD)* introduced by Cheng and Liu (1988).

As an illustration, Figure 13.2 shows the CAD for example 1. In the diagram, components A to D appear from left to right in order of decreasing separation factor, such as relative volatility. Horizontal product lines (PLs), denoted by H1 to H3, separate one product from the other. We write down the product split corresponding to each PL on the right-hand side of the CAD. For example, H1(P1/P234) represents a split that gives an overhead of product P1 and a bottoms of P234 (= P2 + P3 + P4).

Vertical component lines (CLs) specify the component distribution in each product stream. The length of the CL represents the flow rate of a component in the specific product stream bounded by two PLs, and the numerical value of the component molar flow rate is indicated next to each CL. Vertical product lines, denoted by V1 to V3, separate one component from the other. We write the product split at the bottom of the diagram. Thus, V1(A/BCD) represents a split that gives an overhead of component A and a bottoms of components B to D.

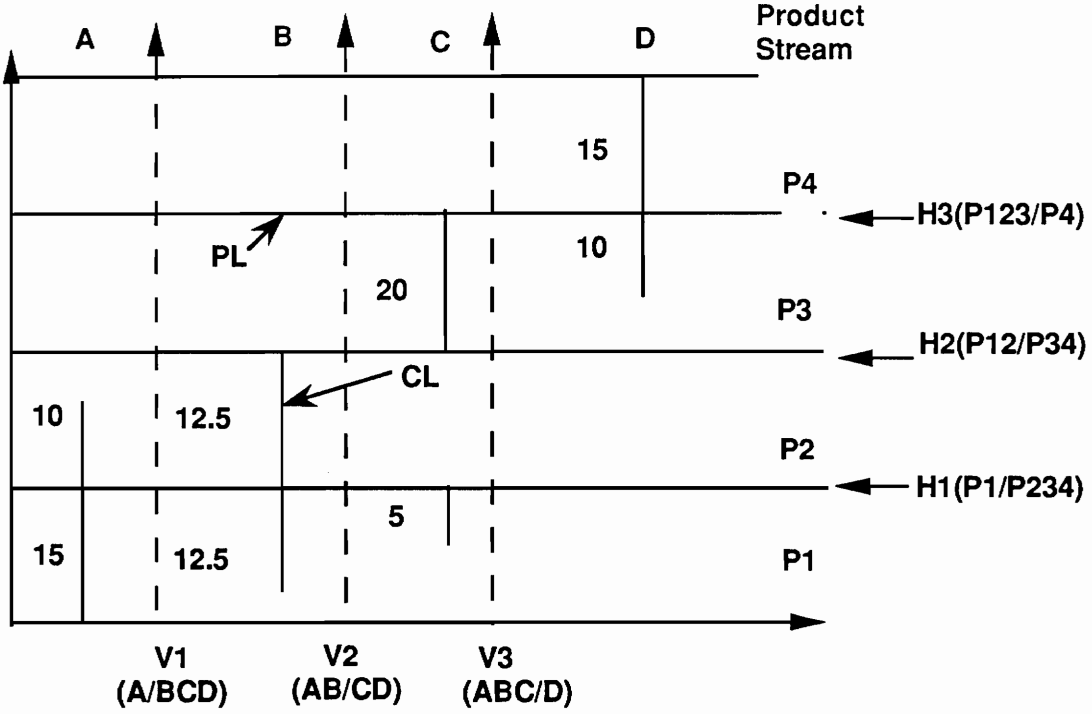


Figure 13.2. Component assignment diagram (CAD) for example one.

To facilitate computer implementation of this approach, we can simplify the CAD by using a *component assignment matrix (CAM)*. The elements of this matrix correspond to the component molar flow rates indicated next to each CL between the two PLs in the CAD. Thus, the CAM for example 1 (called CAM1) represented by Figure 13.2 is:

$$\begin{array}{c|ccccccc|l}
 & A & & B & & C & & D & \\
\hline
P4 & 0 & & 0 & & 0 & & 15 & \\
 & & & & & & & & \leftarrow H3\ (P123/P4) \\
P3 & 0 & & 0 & & 20 & & 10 & \\
 & & & & & & & & \leftarrow H2\ (P12/P34) \\
P2 & 10 & & 12.5 & & 0 & & 0 & \\
 & & & & & & & & \leftarrow H1\ (P1/P234) \\
P1 & 15 & & 12.5 & & 5 & & 0 & \\
\hline
 & & \uparrow & & \uparrow & & \uparrow & & \\
 & & V1 & & V2 & & V3 & & \\
 & & (A/BCD) & & (AB/CD) & & (ABC/D) & &
\end{array} \qquad (13.2)$$

As illustrated by equation 13.2, the CAM is a P x C matrix. We formally define the CAM as:

$$CAM = [\ f_{ij}\ ]$$

the f_{ij} is the flow rate the j-th component in the i-th product (i = 1,2,..., P, j = 1,2,...,C). P is the number of products and C is the number of components. Each column represents a component, and as in the CAD, they appear from left to right in order of decreasing volatility. Therefore, component A (n-butane) is the most volatile, and component D (n-heptane) is the least volatile.

Each row of the matrix represents a product. The rows are ordered in decreasing volatility of products. The bottom row of the matrix is the most volatile product, and the top row is the least volatile.

B. Representing Component and Product Splits

Component splits represented by vertical PLs, such as V1(A/BCD), V2(AB/CD) and V3(ABC/D) in CAM1 (equation 13.2), are *sharp* separations. These splits may not directly yield the desired sloppy product streams. For some synthesis problems, we need to blend two or more products from different vertical component splits, Vm (m = 1 to C-1), to obtain the desired multicomponent products. Remember that sequence S1 in Figure 13.1a employs blending.

Let us look again at the CAM1 and consider the vertical split V2(AB/CD). When we split a CAM, we create two sub-matrices. The overhead and bottoms CAMs resulting from vertical component split Vm we denote as CAM(Vm, ovhd) and CAM (Vm, btm), respectively. Using this notation, and referring to the CAM1, we represent the products from V2(AB/CD), by:

$$\text{CAM(V2, ovhd)} = \begin{array}{c} \\ \text{P4} \\ \text{P3} \\ \text{P2} \\ \text{P1} \end{array} \begin{array}{c} \begin{array}{cc} \text{A} & \text{B} \end{array} \\ \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 10 & 12.5 \\ 15 & 12.5 \end{bmatrix} \end{array} = \begin{array}{c} \\ \text{P2} \\ \text{P1} \end{array} \begin{array}{c} \begin{array}{cc} \text{A} & \text{B} \end{array} \\ \begin{bmatrix} 10 & 12.5 \\ 15 & 12.5 \end{bmatrix} \end{array} \qquad (13.3a)$$

$$\text{CAM(V2,btm)} = \begin{array}{c} \\ \text{P4} \\ \text{P3} \\ \text{P2} \\ \text{P1} \end{array} \begin{array}{c} \begin{array}{cc} \text{A} & \text{B} \end{array} \\ \left[\begin{array}{cc} 0 & 15 \\ 20 & 10 \\ 0 & 0 \\ 0 & 0 \end{array}\right] \end{array} = \begin{array}{c} \\ \text{P4} \\ \text{P3} \end{array} \begin{array}{c} \begin{array}{cc} \text{A} & \text{B} \end{array} \\ \left[\begin{array}{cc} 0 & 15 \\ 20 & 10 \end{array}\right] \end{array} \qquad (13.3b)$$

We see row and its corresponding product identified (P1, P2, P3, and P4), as well as each column and its corresponding component (A, B, C, D).

Another, and perhaps easier way to represent the split schematically is:

$$\begin{tikzcd} & \begin{array}{c}\text{25A}\\\text{25B}\end{array} \\ \begin{array}{c}\text{25A}\\\text{25A}\\\text{25B}\\\text{25C}\end{array} \arrow[ur] \arrow[dr] & \\ & \begin{array}{c}\text{25C}\\\text{25D}\end{array} \end{tikzcd} \qquad (13.3c)$$

No component is distributed in both the overhead and bottoms; thus, V2(AB/CD) is a sharp split between B and C. *Vertical splits are always sharp.*

Product splits represented by the horizontal PLs (H1 - H3) in CAM1 correspond to *either sloppy or sharp* separations depending on the component recovery specifications. For convenience, we represent the CAMs for the overhead and bottoms of horizontal product split Hn (n = 1 to P-1) as CAM(Hn,ovhd) and CAM(Hn,btm), respectively. The horizontal product split H2(P12/P34) gives:

$$\text{CAM(H2,ovhd)} = \begin{array}{c|cccc|} & A & B & C & D \\ \hline 2 & 10 & 12.5 & 0 & 0 \\ 1 & 15 & 12.5 & 5 & 0 \\ \hline \end{array} = \begin{array}{c|ccc|} & A & B & C \\ \hline 2 & 10 & 12.5 & 0 \\ 1 & 15 & 12.5 & 5 \\ \hline \end{array} \qquad (13.4b)$$

$$\text{CAM(H2, btm)} = \begin{array}{c|cccc|} & A & B & C & D \\ \hline 4 & 0 & 0 & 0 & 15 \\ 3 & 0 & 0 & 20 & 10 \\ \hline \end{array} = \begin{array}{c|cc|} & C & D \\ \hline 4 & 0 & 15 \\ 3 & 20 & 10 \\ \hline \end{array} \qquad (13.4a)$$

We can also represent the split H2(P12/P34) as follows:

$$\begin{array}{ccc} & & \begin{array}{c} 25A \\ 25B \\ 5C \end{array} \\ \begin{array}{c} 25A \\ 25B \\ 25C \\ 25D \end{array} & \begin{array}{c} \nearrow \\ \\ \\ \searrow \end{array} & \\ & & \begin{array}{c} 20C \\ 25D \end{array} \end{array} \qquad (13.4c)$$

Since component C appears simultaneously in both the overhead and bottom, split H2(P12/P34) is sloppy.

13.3 FEASIBILITY ANALYSIS OF SEPARATION TASKS

A key difference between vertical component splits (Vm's) and horizontal product splits (Hn's) is that every *Vm is feasible, but not every Hn is technically and/or practically feasible*, assuming, that ordinary distillation is the appropriate separation method. In this section, we introduce basic tools for systematically analyzing the feasibility of horizontal product splits.

CAM1 (equation 13.2) offers the following candidates for horizontal product splits, for separating the feed into product streams 1 to 4: (a) first splits, H1(P1/P234), H2(P12/P34), and H3(P123/P4), or more simply, P1/P234, P12/P34 and P123/P4; (b) second splits, P2/P34, P23/P4, P12/P3, and P1/P23; (c) third or final splits, 1/2, 2/3, and 3/4. We may view all of the candidate splits together in the search tree for horizontal product splits shown in Figure 13.3.

The feasibility of horizontal product splits depends on:

(1) The specification of component recovery fractions in the overhead and bottoms, (d_i and b_i, respectively.

(2) The choice of the *light-key* (LK) and *heavy-key* (HK) components.

(3) The possibility of significant, undesirable distributions of the nonkey components in either the overhead or bottoms.

We discuss these feasibility issues in the following sections.

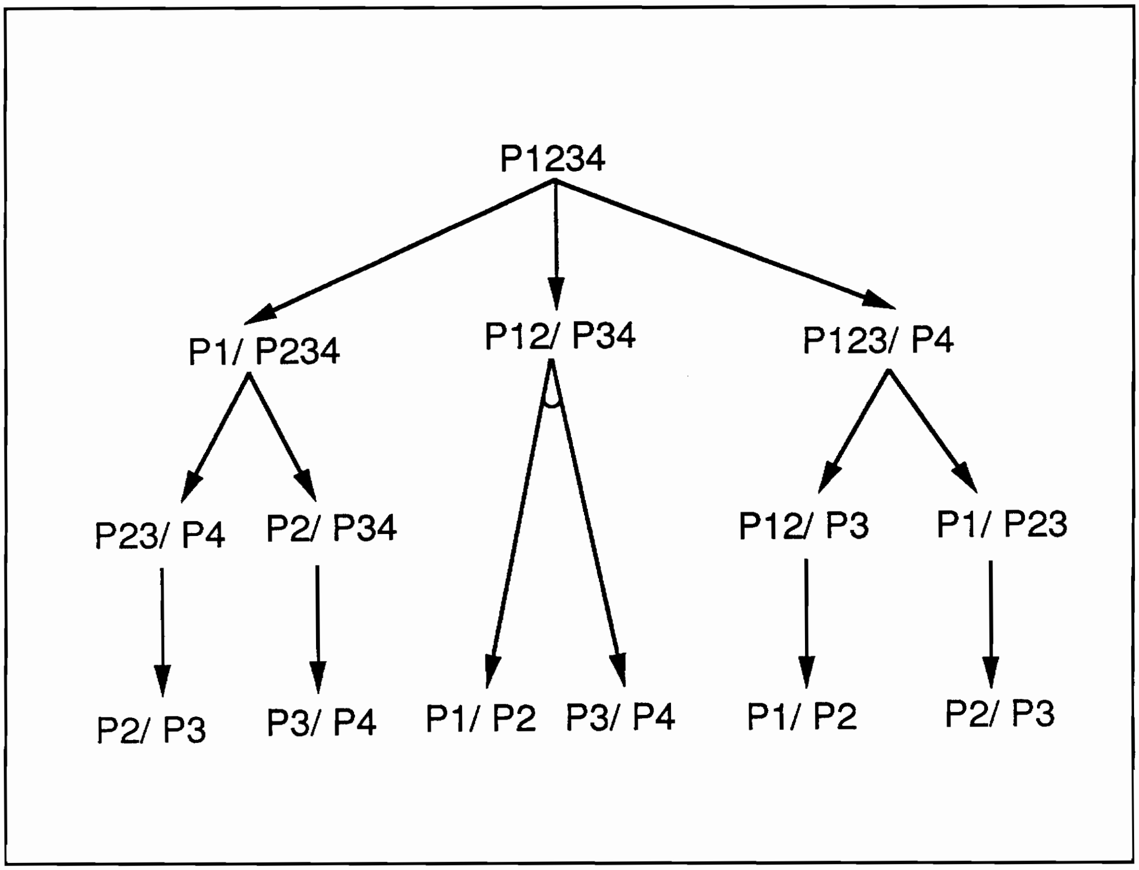


Figure 13.3. Search tree for candidate splits of the product set 1234.

A. Choice of Light-Key (LK) and Heavy-Key (HK) Components

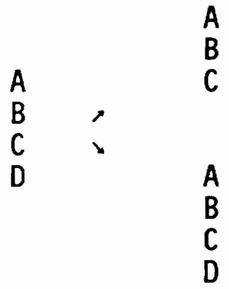
In sharp splits, the determination of the LK and HK components is normally obvious. The key components are typically the two adjacent components between which the split occurs. Thus, for the sharp split,

```
              A
   A          B
   B    ↗
   C    ↘
   D          C
              D
```

B is the LK and C is the HK. As discussed before, in a sharp split, the

LK is the *least volatile* component in the overhead, and the HK is the *most volatile* component in the bottoms.

With a sloppy separation such as:



the choice of LK and HK is less obvious because A, B, and C all occur in both the overhead and bottoms.

In what follows, we suggest three rules for specifying key components in sloppy separations (Cheng and Liu, 1988). These rules are fairly general and have been proven effective in the selection of proper LK and HK components in a large number of sloppy separation problems (Cheng, 1987).

Rule 1: Most splits have a distinctive discontinuity of the component split ratio, (d/b). Choose the LK and HK to fall adjacent to each other around this discontinuity.

Consider for example, the following split ratios resulting from the sloppy separation H2(P12/P34) from CAM1 according to equations 13.3a to 13.3c:

$$\begin{array}{lcccc:cc}
\text{Component} & \rightarrow & A & B & C & D & \\
d/b & \rightarrow & 25/0 & 25/0 & 5/20 & 0/25 & \\
 & & \approx.98/0.02 & \approx.98/.02 & =0.2/0.8 & \approx 0.02/0.98 & \quad (13.5)
\end{array}$$

Here, we use a limiting ratio of 0.98/0.02 to approximate the sharp cut represented by the calculated ratio of 25/0 for light components A and B. (Limiting ratios avoid division by zero, as we shall see in 13.3B.) Likewise, we use a limiting ratio to approximate the sharp cut represented by the calculated ratio of 0/25 for key component D. We use these limiting ratios for sharp splits, be they horizontal or vertical.

We observe a discontinuity in d/b between components B and C, since $d_B > b_B$ and $d_C < b_C$. Thus, we draw a dashed vertical line between components B and C; since B and C are immediately adjacent to that line, we treat them as the key components (LK = B and HK = C).

Rule 2: For sloppy separations with only one distributed component which corresponds to the most volatile or least volatile component, choose this distributed component and its neighbor as keys.

As an illustration, let us consider the CAM for a horizontal product split H(P1/P2) with the most volatile component, A, in both the overhead and bottoms:

$$\begin{array}{c}
 \\ P2 \\ P1
\end{array}
\begin{array}{c}
\begin{array}{cccc} A & B & C & D \end{array} \\
\begin{bmatrix} 5 & 15 & 25 & 12.5 \\ 25 & 0 & 0 & 0 \end{bmatrix}
\end{array}
\leftarrow H(P1/P2) \qquad (13.6)$$

Rule 2 suggests that keys are A (LK) and B (HK). Let us consider the following split:

$$\begin{array}{cc} & \begin{array}{cccc} A & B & C & D \end{array} \\ \begin{array}{c} P2 \\ P1 \end{array} & \left[\begin{array}{cccc} 5 & 15 & 25 & 12.5 \\ 25 & 0 & 0 & 5 \end{array}\right] \end{array} \leftarrow H(P1/P2)$$

Rule 2 does not apply here, since there are several candidates for key components and it is unclear whether A/B or C/D should be the LK/HK. To resolve this difficulty, we rely upon Rule 3.

Rule 3: For other types of sloppy separations, several candidates exist for LK/HK. We can estimate the component distributions resulting from each set of LK/HK using a shortcut feasibility analysis. The set that avoids undesirable distributions for splits of nonkey components in both the overhead and bottom products is the best choice of LK/HK.

Undesirable splits occur because in designing a distillation column, we have only limited degrees of freedom. When we choose the LK and HK components, we typically design the column (number of trays, operating temperature, pressure, reflux ratio, etc.) to match the specified recovery fractions of the LK and HK components. But the limitation in degrees of freedom can create difficulties with nonkey components.

That is, the system can fall victim to equilibrium thermodynamics. Once we "lock in" parameters such as operating pressure and temperature to meet LK and HK component-recovery specifications, we have little control over the distribution of nonkey components in the overhead and bottoms. Vapor-liquid equilibrium thermodynamics determines the distributions. As a result, if the nonkey-component distributions do not satisfy the desired product-recovery specifications, the split is infeasible.

Some "tricks" exist for avoiding undesirable nonkey-component distributions. We can adjust the column height and/or the feed-stage location. However, if the nonkey-component distributions differ significantly from the specifications (say, $\geq$ 10-20% of targeted values), these remedies become prohibitively expensive. In that case, the split is *practically* infeasible. For these reasons, we must determine the feasibility of a sloppy split before incorporating it into the sequence.

B. Shortcut Feasibility Analysis

To assess the feasibility of a sloppy split, we could do rigorous design calculations based on the chosen LK and HK components. Multistage, multicomponent vapor-liquid equilibrium calculations are required. Clearly, this approach can become tedious and time-consuming. In addition, such calculations are not suitable for expert-system implementation. Instead, EXSEP uses a *shortcut* feasibility analysis introduced by Cheng and Liu (1988). The analysis consists of two parts:

(1) a *component-recovery specification test*

(2) a *nonkey-component distribution test.*

Both tests rely on the well-known *Fenske equation*, used to calculate the minimum number of equilibrium stages, N_{min}, for a separation, given the light-key and heavy-key components:

$$N_{min} = \frac{\log[(\frac{d}{b})_{LK}(\frac{b}{d})_{HK}]}{\log \alpha_{LK,HK}} \tag{13.7}$$

where:

N_{min} = minimum number of equilibrium stages.

$\alpha_{LK,HK}$ = relative volatility between the LK and HK components.

d, b = distillate and bottoms molar recovery fractions, respectively.

As an example, we wish to split between n-butane and isobutane. Through thermodynamic information, we find that the mean relative volatility is 1.20 under the desired operating temperature and pressure. We design the column to recover 98% of the isobutane in the overhead and 80% of the n-butane in the bottoms. The minimum number of stages required is:

$$N_{min} = \frac{\log[(\frac{0.98}{0.02})\ (\frac{0.80}{0.20})]}{\log(1.20)} = 28.9$$

We need a minimum of nearly 29 theoretical equilibrium stages to achieve this split.

Looking at the Fenske Equation, we see that N_{min} is a function of the recovery ratios and the relative volatility of the LK and HK components.

Nonkey components *do not* directly affect the equation. They *can* affect N_{min} indirectly, however. For example, if their presence changes the relative volatility between the LK and the HK, N_{min} changes.

Armed with the Fenske equation, we can test the feasibility of a nonsharp split using the component-recovery specification test and the nonkey-component distribution test.

1. Component-Recovery Specification Test

Consider a multicomponent mixture in order of decreasing volatility,

$$\ldots\ LLK3,\ LLK2,\ LLK1,\ LK,\ HK,\ HHK1,\ HHK2,\ HHK3,\ \ldots.$$

where LK and HK are the light- and heavy-key components. The nonkey components are LLK1, LLK2, HHK1, HHK2, etc. LLK1 is a *lighter-than-light-key* adjacent in volatility to the LK, while HHK1 is a heavier-than-heavy-key adjacent in volatility to the HK.

The component-recovery specification test says that for a sloppy split to be feasible, the desired component-recovery specifications must satisfy the following relationships:

$$\cdots\ d_{LLK3} \geq d_{LLK2} \geq d_{LLK1} \geq d_{LK} \geq d_{HK} \geq d_{HHK1} \geq d_{HHK2} \geq d_{HHK3}\ \cdots \qquad (13.8a)$$

$$\cdots\ b_{LLK3} \leq b_{LLK2} \leq b_{LLK1} \leq b_{LK} \leq b_{HK} \leq b_{HHK1} \leq b_{HHK2} \leq b_{HHK3}\ \cdots \qquad (13.8b)$$

For a proof of this using the Fenske equation, see Cheng and Liu (1988).

Essentially, equations 13.8a and b say that thermodynamically, the distribution of components with higher volatilities will favor the overhead, while lower volatilities will favor the bottoms. If the component-recovery specifications violate these two equations, the split is infeasible.

As an example, we consider the split ratios resulting from the sloppy separation H2(P12/P34) for example 1 with B/C as LK/HK. From the CAM1 and equation 13.5, we find:

$$(d/b)_{LLK} = (d/b)_A \approx 0.98/0.02 \qquad (d/b)_{LK} \approx (d/b)_B \approx .98/.02$$

$$(d/b)_{HK} = (d/b)_C \approx .20/.80 \qquad (d/b)_{HHK} \approx 0.02/0.98$$

Writing in the form of equations 13.8a and b, we get:

$$\text{distillate:}\quad d_{LLK1} \geq d_{LK} \geq d_{HK} \geq d_{HHK1} \rightarrow 0.98 = 0.98 > 0.20 > 0.02$$

$$\text{bottoms:}\quad b_{LLK} \leq b_{LK} \leq b_{HK} \leq b_{HHK1} \rightarrow 0.02 = 0.02 < 0.80 < 0.98$$

Since the results satisfy equations 13.a and b, the sloppy split H2(P12/P34) with B/C as LK/HK is feasible according to the component-recovery specification test.

As another example, we consider the following submatrix of the CAM1 (equation 13.2) for representing the sloppy split H(P1/P2):

$$\begin{array}{cc} & \begin{array}{ccc} A & B & C \end{array} \\ \begin{array}{c} P2 \\ P1 \end{array} & \left[\begin{array}{ccc} 10 & 12.5 & 0 \\ 15 & 12.5 & 5 \end{array}\right] \end{array} \leftarrow H(P1/P2) \tag{13.9}$$

Choosing A/B as LK/HK for H(P1/P2) gives

$$\begin{aligned} (d/b)_{LK} &= (d/b)_A = 15/10 && = 0.6/0.4 \\ (d/b)_{HK} &= (d/b)_B = 12.5/12.5 && = 0.5/0.5 \\ (d/b)_{HHK1} &= (d/b)_C = 5/0 && \approx 0.98/0.02 \end{aligned}$$

When we apply equations 13.8a and 13.8b we get:

overhead: $0.6 > 0.5 \not> 0.98$

bottoms: $0.4 < 0.5 \not< 0.02$

The component-recovery specifications fail to satisfy the feasibility conditions; therefore, H(P1/P2) with A/B as LK/HK is technically infeasible. A common-sense approach also indicates that H1(P1/P2) is infeasible. Since component B is more volatile than component C, we cannot logically expect to recover only 50% of B, but 100% of C, in the overhead product.

2. Nonkey-Component Distribution Test

If the sloppy split specifications pass the component-recovery specification test, then they must undergo the nonkey-component distribution test before EXSEP's shortcut method declares the split feasible. To demonstrate the nonkey-component distribution test, we consider again CAM1 (equation 13.2):

$$\begin{array}{c|ccccccc|l}
 & A & & B & & C & & D & \\
P4 & 0 & & 0 & & 0 & & 15 & \\
 & & & & & & & & \leftarrow \mathrm{H3}\ (\mathrm{P123/P4}) \\
P3 & 0 & & 0 & & 20 & & 10 & \\
 & & & & & & & & \leftarrow \mathrm{H2}\ (\mathrm{P12/P34}) \\
P2 & 10 & & 12.5 & & 0 & & 0 & \\
 & & & & & & & & \leftarrow \mathrm{H1}\ (\mathrm{P1/P234}) \\
P1 & 15 & & 12.5 & & 5 & & 0 & \\
 & & \uparrow & & \uparrow & & \uparrow & & \\
 & & \mathrm{V1} & & \mathrm{V2} & & \mathrm{V3} & & \\
 & & (\mathrm{A/BCD}) & & (\mathrm{AB/CD}) & & (\mathrm{ABC/D}) & &
\end{array} \tag{13.2}$$

H1(P1/P234) has the following component split ratios:

$$\begin{array}{rcccc}
\text{component} \rightarrow & A & B & C & D \\
d/b \rightarrow & 15/10 & 12.5/12.5 & 5/20 & 0/25 \\
 & = 0.6/0.4 & = 0.5/0.5 & \approx 0.2/0.8 & \approx 0.02/0.98
\end{array} \tag{13.10}$$

These split ratios pass the component-recovery specification test:

$$\begin{aligned} \text{overhead:} \quad & 0.6 > 0.5 > 0.2 > 0.02 \\ \text{bottoms:} \quad & 0.4 < 0.5 < 0.8 < 0.98 \end{aligned}$$

Now we consider the nonkey-component distribution test for H1. The flow chart of Figure 13.4 summarizes the following steps required to administer the test:

Step 1. Determine potential LK and HK components. From the H1 split, we see that A/B, B/C, and C/D are all LK/HK candidates.

Step 2. Determine N_{min} using the Fenske equation, (13.7), for each LK/HK candidate. Table 13.2 summarizes these calculations.

Table 13.2. Calculation of N_{min} using the Fenske equation. for all LK/HK's for the sloppy split H1(P1/P234).

LK/HK candidate	K-Value		Relative volatility	Recovery ratios		N_{min}
	LK	HK		$(d/b)_{LK}$	$(d/b)_{HK}$	
A/B	2.46	1.00	2.46	0.6/0.4	0.5/0.5	0.45
B/C	1.00	0.47	2.13	0.5/0.5	0.2/0.8	1.83
C/D	0.47	0.21	2.24	0.2/0.8	0.02/0.98	3.10

Note that on the C/D split, the $(d/b)_{HK}$ is in actuality 0.0/1.0, since component D is non-distributed. For a non-distributed heavy component, we use the limiting ratio of 0.02/0.98. Likewise, for a non-distributed light component, we use 0.98/0.02. These limiting ratios facilitate use of the Fenske equation by eliminating division by zero.

Step 3. Calculate nonkey-component distributions.

To calculate nonkey component distributions, we also use the Fenske equation. For LLKs, we write equation 13.6 as follows:

$$N_{min} = \frac{\log[(\frac{d}{b})_{LLK} \ (\frac{b}{d})_{HK}]}{\log(\alpha_{LLK,HK})} \tag{13.11}$$

This equation has two unknowns: d_{LLK} and b_{LLK}. The N_{min} was calculated in step 2 (see Table 13.2). The $(b/d)_{HK}$ comes from external specifications. The $\alpha_{LLK,HK}$ is the relative volatility between the lighter-than-light-key (LLK) and the heavy-key (HK) components, calculated from thermodynamic properties. We also know that for any component i,

$$d_i + b_i = 1 \tag{13.12}$$

Therefore, we can solve equations 13.11 and 13.12 simultaneously for the recovery ratio $(d/b)_{LLK}$. We can apply this same principle to the HHKs too:

$$N_{min} = \frac{\log\ [(\frac{d}{b})_{LK} \ (\frac{b}{d})_{HHK}]}{\log(\alpha_{LK,HHK})} \tag{13.13}$$

For example, let us calculate $(d/b)_{HHK}$, or $(d/b)_c$, for the split H1(P1/P234) with A/B as LK/HK. Table 13.2 shows that N_{min} equals 0.45. From Table 13.1, we know that $K_A = 2.46$ and $K_c = 0.47$, and $\alpha_{AC} = 5.23$. Equations 13.12 and 13.13 become:

$$0.45 = \frac{\log\left[\left(\frac{0.6}{0.4}\right)\left(\frac{b}{d}\right)_c\right]}{\log(5.23)}$$

$$d_c + b_c = 1$$

Solving these equations simultaneously gives:

$$\left(\frac{d}{b}\right)_c = \frac{0.42}{0.58}$$

According to equation 13.10, the desired $(d/b)_c = (0.2/0.8)$. We see that choosing A/B as LK/HK produces an undesirable distribution of nonkey component C. This split fails the nonkey-component distribution test and is *economically* or *practically infeasible*. Note that we could make the split feasible by adding more equilibrium stages in the column, but as discussed in Cheng and Liu (1988), such modifications become cost-prohibitive. This economic problem leads to step 4.

Step 4. Identify all splits with undesirable nonkey-component distributions as infeasible.

Based on the shortcut calculations from the Fenske equation, we say that if $(d/b)_{calculated}$ is within ± 20% of $(d/b)_{desired}$, the split is feasible. In the sloppy split H1(P1/P234) above with A/B as LK/HK, the calculated $(d/b)_c$ would need to fall into the following range to be considered feasible:

$$\frac{0.23}{0.77} \leq (\frac{d}{b})_c \leq \frac{0.17}{0.83}$$

If the calculated (d/b) falls outside this range, the separator will probably give a prohibitively expensive design.

13.4 SEPARATION SPECIFICATION TABLE (SST)

To facilitate the analysis of the technical feasibility of separation tasks, Cheng and Liu (1988) have proposed a separation specification table (SST).

An SST contains the following information:

(1) The type of separation, Hn's or Vm's.

(2) The overhead (ovhd or D) and bottoms (btm or B) products.

(3) The choice of LK and HK components.

(4) The separation factor between the LK and HK components (either relative volatility or boiling-point difference).

(5) The calculated and estimated ratios of the component recovery fraction in the overhead to that in the bottoms.

(6) For feasible splits, the *coefficient of ease of separation* (CES), defined by the following equation:

$$CES = f \cdot \Delta \cdot \frac{1}{\log[(\frac{d}{b})_{LK} \quad (\frac{b}{d})_{HK} \quad]} \tag{13.14}$$

where

- f = min (D/B, B/D) i.e., the ratio of the total molar flow rate of the overhead and bottoms products, D/B or B/D, whichever is smaller.
- Δ = ΔT, the boiling-point difference between key

components, or $\Delta = 100\ (\alpha_{LK,HK} - 1)$.

In the equation for the CES (13.14), the logarithmic term reflects the effect of split sloppiness on the ease of separation, mimicking a similar effect on the minimum number of theoretical stages N_{min} according to the Fenske equation (13.7). In particular, the higher the CES value, the easier is the split.

Table 13.3 is an SST for the first splits shown in CAM1 (equation 13.2). We can take a closer look at the construction of the table to illustrate some applications of the feasibility analysis described in section 13.3. The feasibility analysis summarized in Table 13.3 does not consider *split keys* (e.g., a LK/HK of A/C). Only components of *adjacent* volatility are considered as a LK/HK combination. The reason? With split keys, such as a LK/HK of A/C, the "middle key" components (in this case, B) almost invariably have undesirable nonkey-component distributions and are therefore infeasible.

Consider split H1(P1/P234) with A/B as LK/HK according to Table 13.3:

$$(d/b)_{LK} = (d/b)_A = 15/10 = 0.6/0.4$$

$$(d/b)_{HK} = (d/b)_B = 12.5/12.5 = 0.5/0.5$$

The $(d/b)_{HHK1} = (d/b)_C$, and the $(d/b)_{HHK2} = (d/b)_D$, are initially unknown and must be estimated using the Fenske Equation.

Table 13.3. SST for the first splits in example 1 represented by equation (13.2).

Split	Overhead/Bottoms	LK/HK	Δ°C	$(\frac{d}{b})_A$	$(\frac{d}{b})_B$	$(\frac{d}{b})_C$	$(\frac{d}{b})_D$	CES
V1	A/BCD → 25/75	A/B	36.6	.98/.02	.02/.98	.02/.98	.02/.98	3.61
V2	AB/CD → 50/50	B/C	32.6	.98/.02	.98/.02	.02/.98	.02/.98	9.67
V3	ABC/D → 75/25	C/D	29.7	.98/.02	.98/.02	.98/.02	.02/.98	2.93
H1[b]	P1/P234 → 32.5/67.5	A/B	36.6	.6/.4	.5/.5	.42/.58	.02/.98	infeasible[a]
H1	P1/P234 → 32.5/67.5	B/C	32.7	.95/.05	.5/.5	.2/.8	.02/.98	infeasible[a]
H2[b]	P12/P34 → 55/45	B/C	32.6	.98/.02	.98/.02	.2/.8	.02/.98	11.7
H2	P12/P34 → 55/45	C/D	29.7	.98/.02	.73/.27	.2/.8	.02/.98	infeasible[a]
H3	P123/P4 → 85/15	C/D	29.7	.98/.02	.98/.02	.98/.02	.4/.6	2.81

[a] Infeasible product splits due to undesirable nonkey-component distributions.

[b] Separations with split keys, that is, H1(P12/P34) with A/C as LK/HK and H2(P12/P34) with B/D as LK/HK are not included due to undesirable nonkey-component distributions.

Notation: underlined recovery ratios are those calculated by the Fenske equation. Thick vertical lines represent the partition between LK and HK components.

To estimate the split ratio of nonkey components (in this case, $(d/b)_C$ and $(d/b)_D$) , we take advantage of the fact that the Fenske equation is applies to different combinations of light and heavy components such as (LLK2, HK), (LLK1, HK), (LK, HHK1), and (LK, HHK2). To find $(d/b)_C$, or $(d/b)_{HHK1}$, we use the relative volatilities given in Table 13.1 and first apply the Fenske equation to (LK, HK), or (A, B) to determine N_{min}:

$$N_{min} = \frac{\log[(\frac{d}{b})_{LK}\ (\frac{b}{d})_{HK}]}{\log\alpha_{LK,HK}} = \frac{\log[(\frac{d}{b})_{A}\ (\frac{b}{d})_{B}]}{\log\alpha_{AB}}$$

$$= \frac{\log[(\frac{0.6}{0.4})(\frac{0.5}{0.5})]}{\log\ 2.46} = 0.4504$$

We then use this calculated N_{min} to apply the equation to (LK,HHK1), or (A, C):

$$N_{min} = 0.4504 = \frac{\log[(\frac{d}{b})_{LK}\ (\frac{b}{d})_{HHK1}]}{\log\ \alpha_{LK,\ HHK1}}$$

$$= \frac{\log[(\frac{d}{b})_{A}(\frac{b}{d})_{C}]}{\log\ \alpha_{AC}} = \frac{\log[(\frac{0.6}{0.4})(\frac{b}{d})_{C}]}{\log\ 5.23}$$

Since $d_i + b_i = 1$ for any component i, we can solve the last equation to obtain b_C = 0.4159 and d_C = 0.5841, or $(d/b)_{HHK1}$ = $(d/b)_C$ = 0.42/0.58. In Table 13.3, a underline indicates this split ratio is as an estimated value. This estimated split ratio of $(d/b)_C$ = 0.42/0.58 represents a significant distribution of a nonkey component C in both the overhead and bottoms. As discussed in Section 13.3, to avoid this undesirable distributions of nonkey components in both the overhead and bottoms, the designer often needs to use a distillation column with a large number of theoretical stages, resulting in a costly design. Since $(d/b)_{calculated}$ is *not* within 20% of the $(d/b)_{desired}$, we designate H1(P1/P234) with A/B as LK/HK as *an economically or a practically infeasible split*, and write "infeasible" in the CES column.

We can also apply the shortcut feasibility analysis to the other Hn's and Vm's in Table 13.3. For example, for H2(P12/P34) with B/C as LK/HK, we find:

$$(d/b)_{LK} = (d/b)_B = 25/0 \approx 0.98/0.02$$

$$(d/b)_{HK} = (d/b)_C = 5/20 = 0.2/0.8$$

As with H1(P1/P234), we can apply the Fenske equation to estimate the split ratio of nonkey component HHK1 (component D) for split H2(P12/P34) with B/C as LK/HK. We find $b_D \approx 0.999$, and $d_D \approx 0.001$, as is normalized to 0.02/0.98 and underlined in Table 13.3. This horizontal product split is "feasible" because: (1) it does not produce undesirable nonkey-component distributions; and (2) its component-recovery specifications satisfy the feasibility conditions (equations 13.8a and b) discussed in section 13.3B.1. For this split, Table 13.3 shows:

$$f = \min(D/B,\ B/D) = \min(55/45,\ 45/55) = 45/55$$

$$\Delta = 32.6^{\circ}C$$

$$(d/b)_{LK} = (d/b)_{B} = 0.98/0.02$$

$$(d/b)_{HK} = (d/b)_{C} = 0.2/0.8$$

We can then calculate the CES value from equation 13.14:

$$CES = f \cdot \Delta \cdot \frac{1}{\log[(\frac{d}{b})_{LK}\ (\frac{b}{d})_{HK}\ ]}$$

$$= (45/55) \cdot (32.6) \cdot \frac{1}{\log[(\frac{0.98}{0.02})(\frac{0.8}{0.2})]} = 11.7$$

Thus for the split H2(P12/P34) with B/C as LK/HK Table 13.3 lists the CES as 11.7.

Although split H2 is feasible with B/C as LK/HK, it may not be feasible with a different set of LK/HK components. For example, compare the nonkey component distributions for split H2 when B/C and C/D are LK/HK, respectively:

(a) B/C as LK/HK:

$(d/b)_{LLK1} = (d/b)_A = 0.98/0.02 \quad (d/b)_{HHK1} = (d/b)_D \approx 0.02/0.98$

(b) C/D as LK/HK:

$(d/b)_{LLK2} = (d/b)_A = 0.98/0.02 \quad (d/b)_{LLK1} = (d/b)_B \approx 0.73/0.27$

To choose the most appropriate set of LK/HK components, we can rely upon an important observation of Cheng and Liu (1988): if the split ratio of a light component, LLK, predicted by the Fenske equation at total reflux is greater than or equal to a limiting ratio of 0.98/0.02, then this light component does not significantly distribute in both the overhead and bottoms under common operating reflux ratios R_D = 1.1-1.5 $R_{D,min}$. This observation holds true for a heavy component, HHK, that has a split ratio less than or equal to the limiting ratio of 0.02/0.98. For H2(P12/P34) with B/C as LK/HK, $(d/b)_{LLK1}$ and $(d/b)_{HHK1}$ satisfy these limiting conditions and the split is feasible. In contrast, choosing C/D as LK/HK results in the nonkey component split ratio $(d/b)_{LLK1} \approx 0.73/0.27 < 0.98/0.02$. Therefore, choosing C/D as LK/HK makes H2(P12/P34) infeasible.

As a final note, we recall that the sharp splits (V1, V2, and V3) are always feasible. No nonkey-component distribution calculation is required, and in Table 13.3, we simply enter the limiting ratios for the nonkeys (0.98/0.02 for LLK's, and 0.02/0.98 for HHK's).

To summarize, combining the SST with the CAM provides a simple means to analyze the important characteristics (both the technical feasibility and the ease of separation) of horizontal product splits and vertical component splits. The SST is a useful tool to properly define and specify key and nonkey components, quickly identify feasible and infeasible separation tasks, and systematically compare the relative ease of all feasible splits.

13.5 BYPASS ANALYSIS AND PSEUDOPRODUCT TRANSFORMATION

A. Using Bypass

When dealing with multicomponent systems, we may at times be able to *bypass* some of the feed (or an intermediate product stream) directly to a final product stream. In general, bypassing reduces cost because it decreases the mass load being sent to the separators. Having less material to process reduces both capital and operating costs.

When can bypass be used? What streams are subjected to bypass? The CAM representation of the separation-synthesis problem provides clear criteria for stream bypass. Essentially, a stream is subjected to bypass if it has at least one *all-component-inclusive product.*

Consider, for example, the CAM1 (equation 13.2). We see from equations 13.4a and 13.4b that the overhead and bottoms of split H2(P12/P34) are:

$$\text{CAM(H2,ovhd)} = \begin{array}{c} \\ \text{P2} \\ \text{P1} \end{array} \begin{array}{c} \begin{array}{ccc} \text{A} & \text{B} & \text{C} \end{array} \\ \begin{bmatrix} 10 & 12.5 & 0 \\ 15 & 12.5 & 5 \end{bmatrix} \end{array} \tag{13.4a}$$

$$\text{CAM(H2,btm)} = \begin{array}{c} \\ \text{P4} \\ \text{P3} \end{array} \begin{array}{c} \begin{array}{cc} \text{C} & \text{D} \end{array} \\ \begin{bmatrix} 0 & 15 \\ 20 & 10 \end{bmatrix} \end{array} \tag{13.4b}$$

Now look at the overhead CAM (equation 13.4a). The overhead stream consists of 25A + 25B + 5C, and needs to be separated further to meet product specifications for products P1 and P2. This stream will be the feed into the next separator, and product P1 contains all the components (A, B and C) in this feed. We therefore call P1 an all-component-inclusive product, and the stream represented by CAM(H2,ovhd) is subjected to bypass. Likewise, CAM(H2,btm) consists of 20C + 25D, and shows that product P3 is also all-component-inclusive, containing of all the components (C and D) that appear in the stream.

We can examine the bypass mechanism in more detail using CAM(H2,ovhd). As mentioned, the stream flow rate consists of 25 mol/hr of A, 25 mol/hr of B and 5 mol/hr of C. We also see from the CAM (equation 13.4a) that product P1 consists of 15 mol/hr of A, 12.5 mol/hr of B, and 5 mol/hr of C. To determine the maximum amount that can bypass the subsequent separator and directly form part of product P1, we use the comparison shown in Table 13.4.

Table 13.4. Determination of the maximum amount of the overhead stream that can be bypassed for the CAM of equation (13.4a).

	A	B	C
Component flow in product P1, mol/hr	15	12.5	5
Component flow in stream (feed), mol/hr	25	25	5
Component flow in the product divided by that in the feed, with percentage	15/25 = 60%	12.5/25 = <u>50%</u>	5/5 = 100%

We see from this table that component B is *a limiting component*, since product P1 requires a lower percentage of the total B from the stream (50% of B) compared to other components. We can therefore bypass at most 50% of the feed stream around the subsequent separator to product P1. A 50% bypass corresponds to 12.5 mol/hr of A, 12.5 mol/hr of B and 2.5 mol/hr of C, which gives:

$$\underset{\substack{\text{Feed = 55 mol/hr}\\ \text{to next separator}}}{\begin{array}{cc} & \begin{array}{ccc} A & B & C \end{array} \\ \begin{array}{c} P2 \\ P1 \end{array} & \begin{bmatrix} 10 & 12.5 & 0 \\ 15 & 12.5 & 5 \end{bmatrix} \end{array}} \xrightarrow[\text{2.5C) to P1}]{\text{-(12.5A, 12.5B,}} \underset{\substack{\text{Feed = 27.5 mol/hr}\\ \text{to next separator}}}{\begin{array}{cc} & \begin{array}{ccc} A & B & C \end{array} \\ \begin{array}{c} P2 \\ P1 \end{array} & \begin{bmatrix} 10 & 12.5 & 0 \\ 2.5 & 0 & 2.5 \end{bmatrix} \end{array}} \tag{13.15}$$

This bypass reduces the feed rate to the subsequent separator from 55 to 27.5 mol/hr. In general, stream bypass greatly reduces the mass load of downstream separations.

B. Effects of Stream Bypass

Bypass has both obvious and more subtle effects on multicomponent separation sequencing. First and foremost, bypass reduces the mass load of downstream separations (illustrated in equation 13.15), which in turn reduces separation costs.

On the more subtle side, bypass tends to *increase the sharpness* of downstream separations. Consider the following CAM:

$$\begin{array}{cc} & \begin{array}{ccccc} A & & B & & C \end{array} \\ \begin{array}{c} P2 \\ P1 \end{array} & \left[\begin{array}{ccccc} 0 & & 0 & & 15 \\ 15 & & 15 & & 15 \end{array}\right] \leftarrow H1 \\ & \begin{array}{ccccc} & & & \uparrow & \\ & & & V2 & \end{array} \end{array} \xrightarrow[\text{to } P1]{-(7.5A,\ 7.5B,\ 15C)} \begin{array}{cc} & \begin{array}{ccccc} A & & B & & C \end{array} \\ \begin{array}{c} P2 \\ P1 \end{array} & \left[\begin{array}{ccccc} 0 & & 0 & & 15 \\ 7.5 & & 7.5 & & 0 \end{array}\right] \leftarrow H1 \\ & \begin{array}{ccccc} & & & \uparrow & \\ & & & V2 & \end{array} \end{array}$$

In this case, bypass has made splits H1 and V2 equivalent, and has turned H1 from a sloppy to a sharp split.

Because bypass affects the sharpness of the split, it can also affect the *feasibility* of the split. Consider the CAM2 shown below, where A is propane, B is isobutane, and C is n-butane.:

$$\begin{array}{cc} & \begin{array}{ccccc} A & & B & & C \end{array} \\ \begin{array}{c} P2 \\ P1 \end{array} & \left[\begin{array}{ccccc} 20 & & 70 & & 80 \\ 15 & & 15 & & 15 \end{array}\right] \leftarrow H1 \\ & \begin{array}{ccccc} & \uparrow & & \uparrow & \\ & V1 & & V2 & \end{array} \end{array} \qquad \text{(CAM2)}$$

We can bypass portions of the feed to both products P1 and P2, since both are all-component-inclusive. Before we bypass, however, let us consider the separation specification table for this CAM, shown in Table 13.5a. We see the horizontal split H1 is infeasible for both sets of LK/HK, A/B and B/C. Now, however, we bypass directly to both products P1 and P2. The CAM representation is:

Table 13.5a. SST for the first splits in CAM2 without bypass.

Split	Overhead/Bottoms	LK/HK	Δ°C	$(\frac{d}{b})_A$	$(\frac{d}{b})_B$	$(\frac{d}{b})_C$	CES
V1	A/BC → 35/180	A/B	30.4	.98/.02	.02/.98	.02/.98	1.75
V2	AB/C → 120/95	B/C	11.2	.98/.02	.98/.02	.02/.98	2.62
H1	P1/P2 → 45/170	A/B	30.4	.43/.57	.18/.82	.12/.88	infeasible
H1	P1/P2 → 45/170	B/C	11.2	.23/.77	.18/.82	.16/.84	infeasible

Table 13.5b. SST for the first splits in CAM2 with bypass.

Split	Overhead/Bottoms	LK/HK	Δ°C	$(\frac{d}{b})_A$	$(\frac{d}{b})_B$	$(\frac{d}{b})_C$	CES
V1	A/BC → 11.5/59	A/B	30.4	.98/.02	.02/.98	.02/.98	1.75
V2	AB/C → 39.4/31.1	B/C	11.2	.98/.02	.98/.02	.02/.98	2.62
H1	P1/P2 → 11.1/59.4	A/B	30.4	.83/.17	.06/.94	.02/.98	2.98
H1	P1/P2 → 11.1/59.4	B/C	11.2	.52/.48	.06/.94	.02/.98	infeasible

$$\begin{array}{cc} & \begin{array}{ccc} A & B & C \end{array} \\ \begin{array}{c} P2 \\ P1 \end{array} & \left[\begin{array}{ccc} 20 & 70 & 80 \\ 15 & 15 & 15 \end{array} \right] \\ & \begin{array}{cc} \uparrow & \uparrow \\ V1 & V2 \end{array} \end{array} \xrightarrow[-(5.5A,\ 13.4B,\ 15C) \text{ to } P1]{-(18A,\ 43.7B,\ 48.9C) \text{ to } P2} \begin{array}{cc} & \begin{array}{ccc} A & B & C \end{array} \\ \begin{array}{c} P2 \\ P1 \end{array} & \left[\begin{array}{ccc} 2 & 26.3 & 31.1 \\ 9.5 & 1.6 & 0 \end{array} \right] \\ & \begin{array}{cc} \uparrow & \uparrow \\ V1 & V2 \end{array} \end{array}$$

The SST for CAM2 after bypass is shown in Table 13.5b. Now we compare the feasibility H1 with A/B as LK/HK for CAM2 with and without bypass (Tables 13.5 a and b). Without bypass, we see that H1 is infeasible. With bypass, however, H1 with A/B as LK/HK *is* feasible. The conclusion: *bypass can turn a previously infeasible split into a feasible one.*

C. Pseudoproduct Transformation

Stream bypass is significant because it provides an opportunistic way to: 1) reduce the mass load and total system cost, and 2) possibly make previously infeasible horizontal splits feasible.

We can, however, make horizontal splits feasible even if stream bypass is not possible; we can use pseudoproduct transformation. With CAM1 (equation 13.2) and its accompanying SST in Table 13.3, we saw that H1(P1/P234) was infeasible due to undesirable nonkey-component distributions. We can make H1(P1/P234) feasible by splitting product P1 into two *pseudoproducts*, P1* and P1′.

The following CAM illustrates this pseudoproduct transformation:

$$\begin{array}{c|ccccccc|c}
 & A & & B & & C & & D & \\
P4 & 0 & & 0 & & 0 & & 15 & \\
 & & & & & & & & \leftarrow H4 \\
P3 & 0 & & 0 & & 20 & & 10 & \\
 & & & & & & & & \leftarrow H3 \\
P1' & 0 & & 12.5 & & 5 & & 0 & \\
 & & & & & & & & \leftarrow H2 \\
P2 & 10 & & 12.5 & & 0 & & 0 & \\
 & & & & & & & & \leftarrow H1 \\
P1^{*} & 15 & & 0 & & 0 & & 0 & \\
 & & \uparrow & & \uparrow & & \uparrow & & \\
 & & V1 & & V2 & & V3 & &
\end{array}$$

Since all component recovery fractions corresponding to component flow rates in the equivalent product set (P1*,P2,P1′,P3,P4) satisfy equations 13.8a and 13.8b, splits H1(P1*/P21′34), H2(P1*2/P1′34), H3(P1*21′/P34) and H4(P1*21′3/P4) are technically feasible. The SST is shown in Table 13.6. After carrying out these splits, we can readily obtain the desired product P1 by blending together pseudoproducts P1* and P1′.

One drawback of this approach is when a pseudoproduct is formed, an additional separator may be needed to achieve objectives. Of course, an additional separator generally means higher capital cost. When we desire to make infeasible horizontal splits feasible, it is preferable to use stream bypass instead of pseudoproduct transformation wherever possible.

Table 13.6. SST for the first splits in CAM1 with pseudoproduct transformation.

Split	Overhead/Bottoms	LK/HK	Δ°C	$(\frac{d}{b})_A$	$(\frac{d}{b})_B$	$(\frac{d}{b})_C$	$(\frac{d}{b})_D$	CES
V1	A/BCD → 25/75	A/B	36.6	.98/.02	.02/.98	.02/.98	.02/.98	3.61
V2	AB/CD → 50/50	B/C	32.6	.98/.02	.98/.02	.02/.98	.02/.98	9.67
V3	ABC/D → 75/25	C/D	29.7	.98/.02	.98/.02	.98/.02	.02/.98	2.93
H1	P1*/P21'34 → 15/85	A/B	36.6	.6/.4	.02/.98	.02/.98	.02/.98	3.46
H2	P1*2/P1'34 → 38/62	A/B	36.6	.98/.02	.5/.5	.04/.96	.02/.98	13.0
H2	P1*2/P1'34 → 38/62	B/C	32.6	.98/.02	.5/.5	.02/.98	.02/.98	11.6
H3	P1*21'/P34 → 55/45	B/C	32.6	.98/.02	.98/.02	.2/.8	.02/.98	11.7
H3	P1*21'/P34 → 55/45	C/D	29.7	.98/.02	.75/.25	.2/.8	.02/.98	infeasible
H4	P1*21'3/P4 → 85/15	C/D	29.7	.98/.02	.98/.02	.98/.02	.4/.6	2.81

13.6 HEURISTICS FOR SEPARATION SYNTHESIS

Thus far, we have developed a number of basic concepts for analyzing multicomponent separations. In particular, we introduced the component assignment matrix (CAM), shortcut feasibility analysis and the separation specification table (SST), bypass analysis, and pseudoproduct transformation.

Once we have identified potential multicomponent splits, we can choose one based on *rank-ordered heuristics*. Rank-ordered heuristics are rules of thumb applied one by one in the order specified to guide the selection of splits for developing efficient separation sequences. If one heuristic is not applicable to a given separation problem, we simply move on to the next one.

EXSEP uses the rank-ordered heuristic method of Nadgir and Liu (1983), including the extensions developed by Cheng and Liu (1988) as well as Liu et. al. (1990). In the method, heuristics are classified into four categories:

(1) *Method Heuristics* - designated M heuristics, they favor separation methods based on problem specifications.

(2) *Design Heuristics* - designated D heuristics, they favor separation sequences based on desirable product properties.

(3) *Species Heuristics* - designated S heuristics, they are based on property differences between the species to be separated.

(4) *Composition Heuristics* - designated C heuristics, they relate

the effects of feed and product composition to overall sequence cost.

A. Chosen Rank-Ordered Heuristics

EXSEP uses the following rank-ordered heuristics:

- M1 Favor ordinary distillation and remove mass-separating agents first.
- M2 Avoid vacuum distillation and refrigeration.
- D1 Favor smallest product set.
- S1 Remove corrosive and hazardous components first.
- S2 Perform difficult separations last.
- C1 Remove most plentiful product first.
- C2 Favor 50/50 split.

Method heuristics M1 and M2 first decide the separation methods to be used. Design heuristic D1 and species heuristics S1 and S2 give guidelines about forbidden splits resulting from product specifications, and about essential first and last separations. Finally, heuristics C1 and C2, along with the CES calculated from equation 12.14 if needed, give the actual sequence synthesis.

A complete description of these heuristics follows.

1. *Heuristic M1 - Favor ordinary distillation and remove mass-*

separating agent first.

(a) All other things being equal, favor separation methods such as ordinary distillation that use only energy separating agents, and avoid using separation methods such as extractive distillation that require species not normally present in the feed (i.e., the mass separating agents). (Refer to Rudd, et al., 1973, pp.174-181, for details.) However, if the separation factor or relative volatility of the key components, $\alpha_{LK,HK}$, is less than 1.05 (Van Winkle, 1967, p.381; Seader and Westerberg, 1977) or 1.10 (Nath and Motard, 1981), ordinary distillation should not be used. Instead, an MSA may be used to improve the relative volatility between the key components.

(b) When using an MSA, remove it in the separator immediately following the one in which it was used (Hendry and Hughes, 1972; Rudd et al.,1973, pp.174-180; Seader and Westerberg, 1977).

2. *Heuristic M2 - avoid vacuum distillation and refrigeration.*

All other things being equal, avoid extremes in temperature and pressure, but aim higher rather than lower (Rudd et al., 1973, pp. 182-183) if adjustments must be made. If vacuum distillation is required, instead consider liquid-liquid extraction with various solvents. If the process requires refrigeration (e.g., for separating materials of low boiling points with high relative volatilities as distillate products), consider less expensive alternatives to distillation, such as absorption (Souders, 1964; Seader and Westerberg, 1977; Nath and Motard, 1981).

3. *Heuristic D1 - favor smallest product set.*

Favor sequences that yield the minimum necessary number of products. Avoid sequences that separate components that are in the same final product (Thompson and King, 1972; King, 1980, p.720). In other words, for multicomponent products, favor sequences that produce these products directly or with a minimum of blending, unless relative volatilities for such a sequence are appreciably lower than those for one that requires additional separators and blending (Seader and Westerberg, 1977; Henley and Seader, 1981, p.541).

4. *Heuristic S1 - remove corrosive or hazardous components first.*

(Rudd et al., 1973, p.170)

5. *Heuristic S2 - perform difficult separations last.*

All other things being equal, perform difficult separations last (Harbert, 1957; Rudd et al., 1973, pp. 171-174). In particular, perform separations in the absence of nonkey components when the relative volatility of the key components is close to unity. Select sequences that do not allow nonkey components to be present in separations where key components are close together in volatility or separation factor (Heaven, 1969; King, 1980, p.715).

6. *Heuristic C1 - remove the most plentiful component first.*

A product composing of a large fraction of feed should be separated first, provided that the separation factor or relative volatility is reasonable (Nishimura and Hiraizumi, 1971; Rudd et al., 1973, pp.167-169, King, 1980, p.715).

7. *Heuristic C2 - favor a 50/50 split.*

If the component distributions do not vary widely, favor sequences that give a more nearly 50/50 equimolal split of the feed between the distillate (D) and bottoms (B) products, provided the separation factor or relative volatility is reasonable (Harbert, 1957; Heaven, 1969; King, 1980, p.715). If it is difficult to judge which split is closest to 50/50 and has a reasonable separation factor or relative volatility, then perform the split with the highest value of the coefficient of ease of separation first (see equation 12.14).

B. An Illustrative Example

To illustrate an application of these ranked-ordered heuristics, let us consider an example of multicomponent separation in the industrial purification of n-butylene by ordinary and extractive distillation (Hendry and Hughes, 1972). Table 13.6 specifies the feed mixture for this example.

Table 13.6. Specification of the feed mixture for the n-Butylene purification by ordinary and extractive distillation.

Species	Mol %	Relative volatility*		$(CES)_I$	$(CES)_{II}$
		$(\alpha)_I$	$(\alpha)_{II}$		
A: Propane	1.47				
		2.45		2.163	
B: 1-Butene	14.75				
		1.18	1.17	3.485	3.29†
C: n-Butane	50.29				
		1.03	1.70	3.485	35.25
D: *trans*-Butene-2	15.62				
E: *cis*-Butene-2	11.96				
		2.50		9.406	
F: *n*-Pentane	5.90				

*$(\alpha)_I$ = adjacent relative volatility at 65.6°C and 1.03 MPa for separation method I, ordinary distillation; $(\alpha)_{II}$ = adjacent relative volatility at 65.6°C and 1.03 MPa for separation method II, extractive distillation.

†The CES value for the split AB/CDEF by method II (extractive distillation) is found from

$$\begin{aligned} CES &= (D/B \text{ or } B/D) \times (\alpha_{II} - 1) \times 100 \\ &= [(1.47 + 14.75)/(50.29 + 15.62 + 11.96 + 5.90)] \times (1.17 - 1) \\ &= 3.29 \end{aligned}$$

The same procedure is used to calculate the CES for ABC/DEF by method 2. The rank lists (RL) of decreasing adjacent relative volatility corresponding to separation methods I and II are given by:

RL(I): ABCDEF RL(II): ACBDEF

The desired products of the separation are A, C, BDE, and F.

For the purpose of illustrating the sequential applications of our rank-ordered heuristics, we need only to consider the always feasible, vertical component splits Vm's. Our calculations of CES values listed in Table 13.6 reflect this assumption. In the next chapter, we present detailed examples using both vertical component splits and horizontal product splits.

The separation synthesis using the rank-ordered heuristics is as follows:

1. *Heuristic M1*: use extractive distillation for split C/DE and ordinary distillation for all other splits.

2. *Heuristic M2*: use low temperature and ambient-to-moderate pressure.

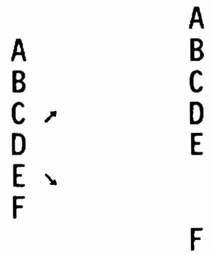
3. *Heuristic D1*: avoid splitting DE as both D and E are in the same product, and blend together B and DE to obtain a multicomponent product BDE.

4. *Heuristic S1*: not applicable.

5. *Heuristic S2*: since split C/DE is difficult and requires extractive distillation, it should be performed last in the absence of A, B, and F.

6. *Heuristic C1*: although C is a large fraction of the feed, it should not be separated first because of the preceding heuristic (S2). Further, the extractive distillation for splitting C/DE should come at the end of the sequence to avoid having the mass separating agent as a possible contaminant in intermediate separations.

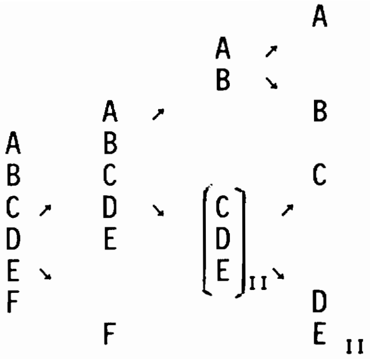
7. *Heuristic C2*: for separating ABCDEF, we perform split ABC/DEF last. We must choose between A/BCDEF, AB/CDEF, and ABCDE/F for the first split. Since ABCDE/F has the largest $(CES)_I$, we perform it first:



To separate ABCDE, the possible splits are A/BCDE and AB/CDE. Since:

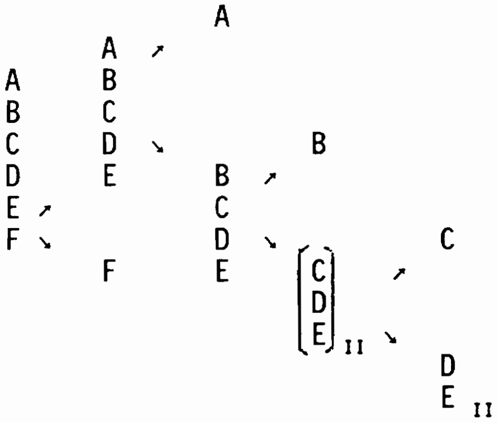
	A/BCDE	AB/CDE
f = D/B or B/D < 1	1.47/92.63	16.22/77.88
$(\alpha - 1) \times 100$	145	18
CES	2.301	3.749

we prefer split AB/CDE over A/BCDE. The resulting sequence, which splits A/B and C/DE last, is as follows.



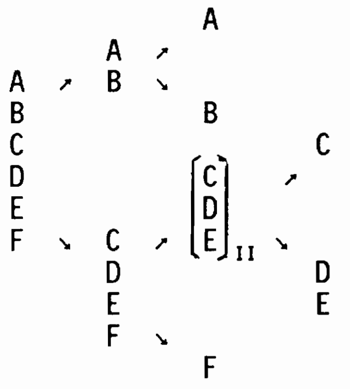
(sequence a)

We can produce a second sequence by splitting A/BCDE first:



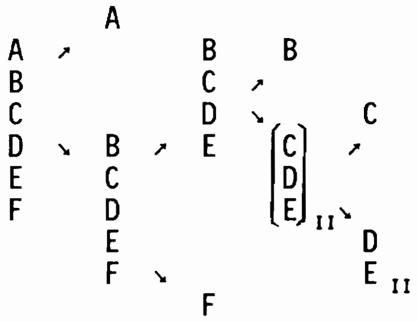
(sequence b)

Since no sequences with the initial split ABCDE/F satisfy constraints imposed by heuristics M1-C1, we instead find a third sequence by examining the alternative initial splits for ABCDEF. We can begin with second best initial split, AB/CDEF, which has the second largest CES_I of 3.485. The resulting sequence, which splits A/B and C/DE last, is:

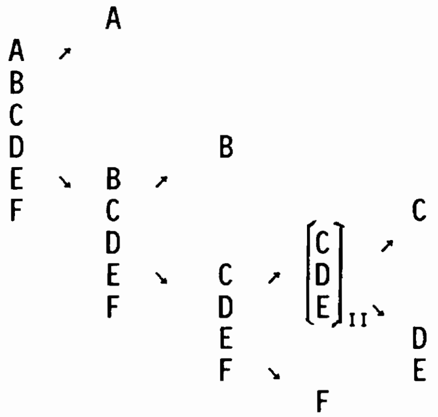


(sequence c)

Alternatively, if we prefer first the third best initial split, A/BCDEF, (CES_I = 2.163), we find two other sequences that split C/DE last:



(sequence d)

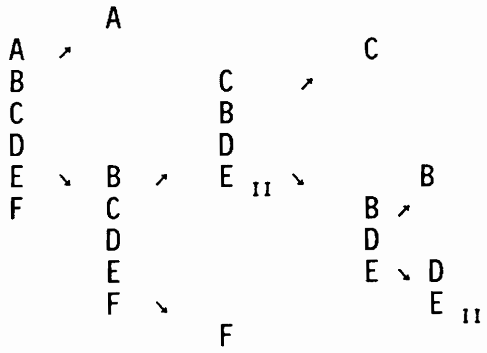


(sequence e)

Note that sequence d is better than sequence e, according to the CES values, in separating BCDEF:

	BCDE/F	B/CDEF
f	5.90/92.62	14.75/83.77
$(\alpha - 1) \times 100$	84.06	24.85
CES	5.355	4.375

How good are our heuristically synthesized sequences for the n-butylene purification? In Table 13.7, we compare sequences synthesized by this heuristic technique with other sequencing techniques (algorithmic, and heuristic-evolutionary techniques). The additional sequence (f), which involves replacing the split B/C in sequence by extractive distillation, is as follows:



(sequence f)

Table 13.7 shows that sequence a is less expensive than the initial sequences obtained by the heuristic-evolutionary techniques of Seader and

TABLE 13.7. A Comparison of Reported Sequences for the n-Butylene Purification

Sequence	Seven rank-ordered heuristics	Algorithmic technique	Heuristic-evolutionary techniques	
		Hendry and Hughes (1972)[a]	Seader and Westerberg (1977)	Nath and Motard (1981)
a	Initial sequence	$867,400/yr (0.8%)		
b	Second sequence	$878,200/yr (1.8%)		
c	Third sequence	$860,400/yr (best)	Final sequence $860,400/yr (best)	Final sequence $658,737/yr (best)
d	Fourth sequence	$878,000/yr (2.0%)	Initial sequence $878,000/yr (2.0%)	
e	Fifth sequence	$872,400/yr (1.5%)	Second sequence $872,400/yr (1.5%)	Second sequence $669,844/yr (1.7%)
f	Final sequence	$1,095,600/yr (27.3%)		Initial sequence $1,171,322/yr (77.8%)

[a] As reported in Hendry (1972) and quoted in Henley and Seader (1981, p. 547)

Westerberg, and Nath and Motard (sequences d and f, respectively). Based on cost data reported by Hendry (1972), sequence a (the initial sequence by EXSEP's method) is only 0.8% more than the best sequence (c) synthesized by the heuristic-evolutionary technique of Seader and Westerberg, and Nath and Motard.

Note that the magnitudes of cost differences among sequences listed in Table 13.7, are relatively small. Such small differences occur not only in the present n-butylene purification problem, but in a large number of other separation-synthesis problems. As Tedder suggests (1975), the magnitude of possible round-off errors (noise) that result from applying optimization techniques to sequencing problems with different initial solutions can often be greater than the cost differences among the sequences. Under such situations, we should select the best sequence from several possibilities based on performance criteria other than the total cost, such as safety and sensitivity, ease of start-up and shutdown, operating temperature and pressure relative to the critical points, overall tower dimension, etc. The rank-ordered heuristics given here provide a simple, effective procedure for the systematic synthesis of good initial sequences for such a multi-objective design optimization.

C. Implementation of the Heuristics in EXSEP

Figure 13.4 shows the step-by-step procedure for implementing the seven rank-ordered heuristics in EXSEP. Section 15.2 describes the details of this implementation from an AI perspective. In what follows, we briefly summarize the essential features of the heuristic implementation. For reference, the seven-rank ordered heuristics again are:

- M1 Favor ordinary distillation and remove mass-separating agents first.
- M2 Avoid vacuum distillation and refrigeration.
- D1 Favor smallest product set.
- S1 Remove corrosive and hazardous components first.
- S2 Perform difficult separations last.
- C1 Remove most plentiful product first.
- C2 Favor 50/50 split.

In EXSEP, we apply heuristic D1 implicitly, and apply M1, M2, S1, S2, C1, and C2 explicitly. In multicomponent separations, implementing M1 and M2 is straightforward. A relative-volatility analysis will determine if ordinary distillation is acceptable. If ordinary distillation is not acceptable, the sequence may require an MSA (mass-separating agent). The MSA is always removed as soon as possible.

For M2, we can use bubble- and dew-point calculations to assess column operating conditions. Heuristic M2 does not affect the flowsheet development (assuming distillation is acceptable) as much as it guides column operating conditions once we have developed the distillation sequence. EXSEP, therefore, focuses on heuristics M1, D1, S1, S2, C1, and

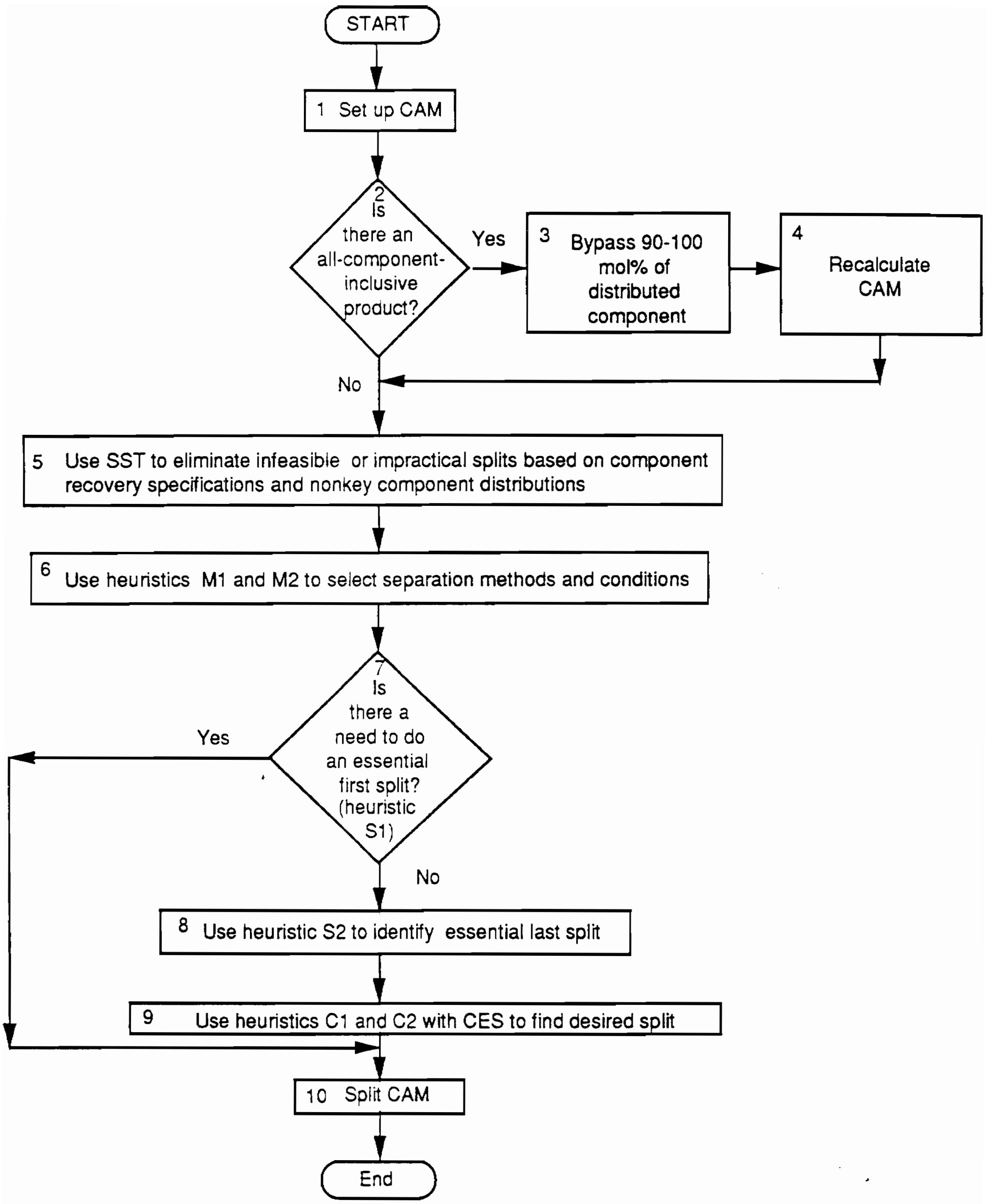

START
1 Set up CAM
2 Is there an all-component-inclusive product?
Yes
3 Bypass 90-100 mol% of distributed component
4 Recalculate CAM
No
5 Use SST to eliminate infeasible or impractical splits based on component recovery specifications and nonkey component distributions
6 Use heuristics M1 and M2 to select separation methods and conditions
7 Is there a need to do an essential first split? (heuristic S1)
Yes
No
8 Use heuristic S2 to identify essential last split
9 Use heuristics C1 and C2 with CES to find desired split
10 Split CAM
End

Figure 13.4. The procedure for implementing the rank-ordered heuristics.

C2, allowing M2 to be used "off line." Future enhancements of EXSEP may also recommend operating conditions.

EXSEP implements heuristic D1 implicitly through the component assignment matrix (CAM). Refer to the problem specified in Table 13.1 and its CAM (equation 13.2). All potential splits (V1, V2, V3, H1, H2, and H3), satisfy heuristic D1. When split a stream into products, we always favor the smallest product set, thus minimizing unnecessary blending of separated products and the associated energy waste.

The core of EXSEP is the application of heuristics S1, S2, C1, and C2, coupled with stream bypass and feasibility analysis. EXSEP inputs thermodynamic data (in the form of K-values at the feed conditions), product specifications, and component properties. It performs its analysis and recommends a split. If the user can then accept or reject this recommendation. If the user accepts the split, EXSEP locks that split in and analyzes the overhead and bottoms streams to recommend subsequent splits. If the user rejects the split, EXSEP backtracks and generates alternate options.

D. Rejected Separation Heuristics

We have chosen the seven rank-ordered heuristics them based on our investigations of research done over the past decade on heuristic synthesis of multicomponent separation sequences (Nadgir and Liu, 1983; Liu, 1987; Cheng and Liu, 1988; Liu et al., 1990).

Comparisons of sequences synthesized by various methods for a large number of multicomponent separation problems (some of which are included in the above references) demonstrate the simplicity and effectiveness of the rank-ordered heuristics. Professor J. D. Seader of the University of Utah, a recognized separation expert who previously developed an alternative set of rank-ordered heuristics (Seader and Westerberg, 1977), has confirmed the superiority of the chosen heuristics over others (Seader, 1990). The recent work of Barnicki and Fair (1990) has successfully applied these seven rank-ordered heuristics to account for separation problems with azeotropes.

A large number of separation heuristics that have appeared in the literature since 1947, and interested readers may refer to a published survey of available heuristics in Liu (1987). Here, we give two examples of heuristics that we consider to be less effective and significant in most separation-synthesis problems, and explain why we do not favor these two heuristics. Our discussion should be helpful to those interested in the selection of effective heuristics for the development of rule-based expert systems for multicomponent separations.

1. *Heuristic D3: favor direct sequence.* During distillation, when neither the relative volatility nor the molar percentage in the feed varies widely, remove the components one by one as distillate products. The resulting sequence is commonly called the direct sequence; the operating pressure tends to be highest in the first separator and reduces

in each subsequent separator (Lockart, 1947; Harbert, 1957; Heaven, 1969; Rudd et al., 1973, pp. 183-184; King, 1980, p. 715).

The direct-sequence rule tends to maximize the total separation load, thus requiring large separators with high investment costs. Furthermore, this heuristic fails to consider the important concept of a balanced separator (column) that features both an equimolar (50/50) split of the feed between the overhead and bottoms products. The 50/50 split minimizes energy consumption (Harbert, 1957; King, 1980, pp. 715-716). Examples that demonstrate the advantage of heuristic C2 (favor 50/50 split) over heuristic D3 (favor direct sequence) appear in Rudd et al. (1973, pp. 167-173).

2. *Heuristic S4: perform easy separations first.* Specifically, arrange the components being separated according to their relative volatilities or separation factors. When the adjacent relative volatilities of the ordered components in the feed vary widely, sequence the splits in the order of decreasing adjacent relative volatility (Seader and Westerberg, 1977). Alternatively, arrange the separations in an increasing order of the coefficient of difficulty of separation (CDS), defined by:

$$CDS = \frac{\log\left\{\dfrac{sp_{LK}}{1-sp_{LK}}\cdot\dfrac{sp_{HK}}{1-sp_{HK}}\right\}}{\log\alpha_{LK,HK}}\frac{D}{D+B}\cdot\left\{1+\left|\frac{D-B}{D+B}\right|\right\} \qquad (13.17)$$

where sp_{LK} and sp_{HK} are the split fractions of the light- and heavy-key components, respectively, in the overhead and bottoms products; D and B are the molal flow rates of the overhead and bottoms products, respectively; and $\alpha_{LK,HK}$ is the relative volatility of the key components. Indirectly, then, favoring a low CDS favors: (1) large $\alpha_{LK,HK}$ (i.e., heuristic S4, perform easy separations first), (2) a balanced column where D = B (i.e., heuristic C2, favor 50/50 split), (3) sloppy splits with low recoveries of the keys so that the split ratios $sp_{LK}/(1 - sp_{LK})$ and $sp_{HK}/(1 - sp_{HK})$ are small, and (4) less overhead product (i.e., heuristic D3, favor direct sequence) (Nath and Motard, 1981).

As described by Seader and Westerberg, heuristic S4 favors easy separations first, with "easy" based on a ranking of separation factors. However, this heuristic fails to consider the effect of feed composition and the importance of a balanced column, and it can make subsequent separations difficult or costly. Consider, for example, the separation of the mixture of light olefins and paraffins specified in Table 13.8 by ordinary distillation (Thompson and King, 1972).

Table 13.8. Specification of the feed mixture for the separation of light olefins and paraffins by ordinary distillation.

Species	Mole Fraction	Relative Volatility* α	CES
A: Ethane	0.20		
		3.50	62.5
B: Propylene	0.15		
		1.20	10.7
C: Propane	0.20		
		2.70	139.1
D: 1-Butane	0.15		
		1.21	9.0
E: *n*-Butane	0.15		
		3.00	35.3
F: *n*-Pentane	0.15		

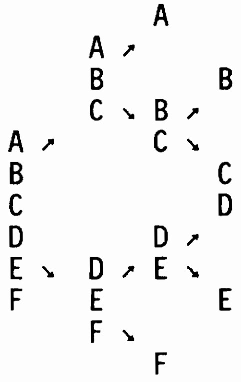
*At 37.8°C and 0.1 MPa.

The literature includes the following two sequences for separating the feed into pure components.

Sequence g: Apply heuristic S4 and arrange the splits in the order of decreasing adjacent relative volatility; total annual cost = $1,234,000/yr (Seader and Westerberg, 1977).

```
       A
A  ↗                    B
B                   B ↗
C             B  ↗  C ↘
D             C         C                      (sequence g)
E  ↘   B  ↗   D         D
F      C      E  ↘  D ↗
       D            E ↘
       E                E
       F  ↘
              F
```

Sequence h: Apply heuristic C2 and arrange the splits to favor a balanced column (favor 50/50 splits); total annual cost = $1,153,000/yr (Seader and Westerberg, 1977; Nadgir and Liu, 1983).



(sequence h)

Comparing sequences g and h suggests that *the dominating heuristic in multicomponent separation sequencing is heuristic C2 (favor 50/50 splits) and not heuristic S4 (perform easy separations first)*. Comparisons of initial sequences for many other multicomponent separation problems confirms this observation (Nadgir and Liu, 1983; Seader, 1990).

Finally, note that applying heuristic S4 based on the CDS (Nath and Motard, 1981), actually attempts to combine heuristics D3 (favor direct sequence) and C2 (favor 50/50 split) by incorporating them into the CDS. As explained before, heuristic D3 fails to consider the importance of a balanced separator, and thus, conflicts with heuristic C2. Incorporating these two conflicting together into a single parameter makes little sense. *We therefore recommend using the CES parameter of equation 13.14 rather than the CDS parameter of equation 13.17.*

13.7 CHAPTER SUMMARY

The development of simple methods for the systematic design of multicomponent separation systems with sloppy product streams has been one of the most challenging problems in process design research for the past twenty years. Such separation systems find much use in the fractionation of refinery light ends and saturates-gas components, and in recycled reactor systems for reactant recovery and byproduct separation where the generally more expensive, sharp separation systems with high component recoveries are often unnecessary.

In this chapter, we have introduced EXSEP from a chemical engineering perspective, using known engineering principles, along with useful rank-ordered heuristics to create a technique that develops a flowsheet for multicomponent separation sequencing in a simple, systematic way. We have introduced some chemical engineering tools and concepts that guide the synthesis of multicomponent separation sequences using EXSEP. Our discussion are includes:

(1) effective and flexible tools for presenting the synthesis problem (the component assignment matrix, CAM), and for analyzing the technical feasibility of splits (separation specification table, SST);

(2) practical design guidelines for a shortcut feasibility analysis of separation tasks using the Fenske equation;

(3) the simple concept of the all-component-inclusive product to

help identify stream-bypass opportunities that minimize the mass load of downstream separators; and

(4) effective rank-ordered heuristics for the synthesis of initial separation sequences.

A LOOK AHEAD

The next chapter discusses EXSEP from a user's perspective. We describe how to input data actually run and use EXSEP to generate efficient, feasible, flowsheets for multicomponent separations.

NOMENCLATURE

b_i = i-th component molar flow rate in the bottoms, mol/h; or in normalized situation, i-th component recovery fraction in the bottoms, dimensionless

B = molar flow rate of the bottoms, mol/h

C = number of components

CAM = component assignment matrix

CAM(Hn,btm), = CAM(Hn,ovhd) CAM for representing the bottoms and overhead, respectively, resulting from horizontal product split Hn.

CAM(Vm,btm), = CAM(Vm,ovhd) CAM for representing the bottoms and overhead, respectively, resulting from vertical component split Vm.

CES = coefficient of ease of separation defined in equation 12.14.

d_i = i-th component molar flow rate in the overhead, mol/hr; or, in normalized situation, i-th component recovery fraction in the overhead, dimensionless

D = molar flow rate of the overhead, mol/hr

f = D/B or B/D, whichever is smaller than or equal to unity, dimensionless

f_{ij} = elements of the component assignment matrix representing the flow rate of the j-th component in the i-th product (i = 1,2,...P; j = 1,2,...C)

Hn = horizontal product split n (n = 1,2,...P-1)

HHK1-3 = heavier-than-heavy key or heavy components 1 to 3 whose volatilities are in a descending order

HK = heavy-key component

K_i = vapor-liquid equilibrium ratio of component i, dimensionless

LK = light-key component

LLK1-3 = lighter-than-light key or light components 1 to 3 whose volatilities are in an ascending order

N_{min} = minimum number of theoretical stages

P = number of product streams; or column pressure, Pa

R_D = operating reflux ratio

$R_{D,min}$ = minimum reflux ratio

S_{min} = apparent minimum number of separators, defined in equation 12.1

sp_{LK} = split fraction of the LK in the overhead

sp_{HK} = split fraction of the HK in the bottoms

SST = separation specification table

Vm = vertical component split m (m = 1,2, ..., C-1)

Superscripts

' = prime, indicate the CAM resulting from, or the horizontal product split H_i (vertical component split Vm) made after, bypassing a portion of the feed around the separator to form directly part of an overhead or a bottoms

'' = double prime, indicates the CAM resulting from, or the horizontal product split H_i (vertical component split Vm) made after, bypassing two portions of the feed around the separator to form directly parts of both overhead and bottoms

Greek Letters

$\alpha_{LK,HK}$ = relative volatility of LK with respect to that of HK

Symbol

Δ = ΔT (normal boiling-point difference, °C) or 100 (α - 1)

REFERENCES

Barnicki, S. D. and J. R. Fair, "Separation System Synthesis: A Knowledge-Based Approach. 1. Liquid Mixture Separations," *Ind. Eng. Chem. Res.*, 29, 421 (1990).

Cheng, S. H., "Systematic Synthesis of Sloppy Multicomponent Separation Sequences," Ph.D. dissertation, Virginia Polytechnic Institute and State University, Blacksburg, VA, 1987.

Cheng, S. H., and Y. A. Liu, "Studies in Chemical Process Design and Synthesis. 8. A Simple Heuristic Method for the Synthesis of Initial Sequences for Sloppy Multicomponent Separations," *Ind. Eng. Chem. Res.*, 27, 2304 (1988).

Harbert, W. D., "Which Tower Goes Where?" *Pet. Refiner.*, 36 (3), 169

(1957).

Heaven, D. L., "Optimum Sequencing of Distillation Columns in Multicomponent Fractionation," M.S. thesis, University of California, Berkeley, 1969.

Hendry, J. E., "Computer Aided Synthesis of Optimal Multicomponent Separation Sequences," Ph.D. dissertation, University of Wisconsin, Madison, WI, 1972.

Hendry, J. E. and R. R. Hughes, "Generating Separation Process Flowsheets," *Chem. Eng. Prog.*, 68(6), 71 (1972).

Henley, E. J. and J. D. Seader, *Equilibrium-Stage Separation Operations in Chemical Engineering*, Wiley, New York, 1981.

King, C. J., *Separation Processes*, 2nd ed., McGraw-Hill, New York, 1980.

Liu, Y. A., "Process Synthesis: Some Simple and Practical Developments," In *Recent Developments in Chemical Process and Plant Design*; Liu, Y. A., McGee, H. A., Jr., Epperly, W. R., Eds.; Wiley: New York, New York, 1987; pp. 147-260.

Liu, Y. A., T. E. Quantrille, and S. H. Cheng, "Studies in Chemical

Process Design and Synthesis. 9. A Unifying method for the Synthesis of Multicomponent Separation Sequences with Sloppy Product Streams," *Ind. Eng. Chem. Res.*, **29**, 2227 (1990).

Lockhart, F. J., "Multi-Column Distillation of Natural Gasoline," *Pet. Refiner*, **26** (8), 105 (1947).

Nadgir, V. M. and Y. A. Liu, "Studies in Chemical Process Design and Synthesis: Part V. A. Simple Heuristic Method for Systematic Synthesis of Initial Sequences for Multicomponent Separations," *AIChE J.*, 29, 926 (1983).

Nath, R., "Studies in the Synthesis of Separation Processes," Ph.D. dissertation, University of Houston, Houston, TX, 1977.

Nath, R. and R. L. Motard, "Evolutionary Synthesis of Separation Processes," *AIChE J.*, **27**, 578 (1981).

Rudd, D. F., G. J. Powers, and J. J. Siirola, *Process Synthesis*, Prentice-Hall, Englewood Cliffs, NJ (1973).

Seader, J.D., Personal Communication, Blacksburg, VA, April (1990).

Seader, J. D. and A. W. Westerberg, "A Combined Heuristic and Evolutionary

Strategy for the Synthesis of Simple Separation Sequences," *AIChE J.*, 23, 951 (1977).

Souders, M., "The Countercurrent Separation Processes," *Chem. Eng. Prog.*, 62 (2), 75 (1964).

Tedder, D. W., "The Heuristic Synthesis and Topology of Optimal Distillation Networks," PhD dissertation, Univ. of Wisconsin, Madison, WI (1975).

Thompson, R. W. and C. J. King, "Systematic Synthesis of Separation Schemes," *AIChE J.*, 18, 941 (1972).

Van Winkle, M., *Distillation*, McGraw-Hill, New York, 1967.

Westerberg, A. W., "The Synthesis of Distillation-Based Separation Systems," *Comput. Chem. Eng.*, 9, 421 (1985).

14

USER'S PERSPECTIVE OF EXSEP

In this chapter, we explain EXSEP from the user's perspective. The typical EXSEP user is a practicing chemical engineer involved in process design engineering. EXSEP provides assistance in process synthesis and flowsheet development for multicomponent separation sequences. We discuss here what EXSEP needs in terms of hardware and technical information required for input, and then move on to actually running and using EXSEP. Finally, we discuss its results and methods to generate alternate flowsheets.

14

USER'S PERSPECTIVE OF EXSEP

In this chapter, we explain EXSEP from the user's perspective. The typical EXSEP user is a practicing chemical engineer involved in process design engineering. EXSEP provides assistance in process synthesis and flowsheet development for multicomponent separation sequences. We discuss here what EXSEP needs in terms of hardware and technical information required for input, and then move on to actually running and using EXSEP. Finally, we discuss its results and methods to generate alternate flowsheets.

14.1 REQUIREMENTS FOR USE

A. Hardware Requirements

EXSEP, developed on an IBM PC AT, will run on any IBM PC, XT, AT or compatible that uses MS DOS. In addition, EXSEP will run on PS/2 machines. From a video standpoint, EXSEP works best with a CGA monitor and card. Though the program will run with a CGA, EGA, VGA and even (Hercules compatible) Monographics systems, we cannot assure compatibility. For printing results, we recommend EPSON or IBM graphics printers.

EXSEP is written in compiled Prolog. As a result, it is a relatively compact program that runs surprisingly fast. Traditionally, expert systems implemented on personal computers have run too slowly to be useful. However, EXSEP can easily outpace our capacity to absorb the information it generates. It can develop a four-separator system in less that 1.0 seconds of CPU time on an 80286 microprocessor running at 8 Mhz.

B. Technical Information

Before running EXSEP, we need to collect the necessary technical information, which includes:

- number of components and products in the system,
- names of each component and product,

- complete material balance, i.e., specified flow rate of each component in each product,
- normal boiling point (in °C) of each component ("normal boiling point" means the temperature where the vapor pressure equals 1 atmosphere),
- equilibrium ratios for each component at feed conditions(i.e., y_i/x_i, where y_i and x_i represent the mole fractions of component i in the vapor and liquid phases, respectively. Note that y_i/x_i is frequently called the *K-value* of component i),
- any particularly corrosive or hazardous components,
- any pseudoproducts (see section 14.5C).

Once we collect the required information, we enter the information into the expert system via EXSEP's user interface. If we desire, EXSEP can save the information on a file. By storing information on a file, we only need to enter it via the keyboard once. For future uses, EXSEP can then conveniently access the file directly.

14.2 USING EXSEP: EXAMPLE 1

A. EXSEP's Operating Process

EXSEP is menu-driven and very easy to use. When using EXSEP for the first time on a given problem, we must enter the data manually. After entering the data, we can then save it in Prolog clause form on a file. If we run the program again and want to use the same information, EXSEP will **reconsult** the file (see the built-in Prolog predicate **reconsult**, section 5.5).

Once EXSEP has all the information, it constructs the CAM (component assignment matrix). It then performs a stream-bypass analysis. When bypass is possible, it interactively asks if we want to use it. If we approve the bypass, EXSEP re-calculates the material balance and reconstructs the CAM.

After performing the bypass analysis, EXSEP builds the separation specification table (SST). The SST assesses feasibility of each split, and calculates the coefficient of ease of separation (CES) for all feasible splits. EXSEP labels any split deemed infeasible and rejects it from consideration.

The SST generates a set of technically feasible splits, but disregards practical and economic concerns. To deal with these concerns, EXSEP then applies the rank-ordered heuristics of section 13.6 to all feasible splits. Specifically, it implements design heuristic D1, method heuristic M1, species heuristics S1 and S2, and composition heuristics C1 and C2. (Note that method heuristic M2 must be implemented off-line, and design heuristic D1 is implemented implicitly through the CAM.) EXSEP gives the complete analysis on the applicability and the ramifications of each heuristic.

After completing the heuristic analysis, EXSEP recommends a split. We can interactively accept or reject EXSEP's recommendation. If we accept the split, EXSEP recalculates the material balance and generates two new process streams (the overhead and bottoms) that now need further separation.

EXSEP analyzes the overhead stream first. It recursively goes through the entire synthesis process (CAM construction, bypass analysis, SST and feasibility analysis, and heuristic analysis) for this new stream. EXSEP then recommends the next split for the overhead stream.

EXSEP continues this process until it develops a complete flowsheet for the overhead products (including the initial splitter). After handling all overhead streams, EXSEP moves on to develop a flowsheet for the bottoms streams. Once it has processed all bottoms streams, EXSEP reports the results. It summarizes the entire flowsheet, specifying for each separator:

- feed composition and flow rate.
- overhead composition and flow rate.
- bottoms composition and flow rate.
- bypass flow rate around the splitter (if applicable).

B. The Problem

For our first example, we use the four-product, four-component problem introduced in chapter 13. For convenience, we have repeated the material specifications in Table 14.1 below. Components A, B, C, and D are n-butane, n-pentane, n-hexane, and n-heptane, respectively.

Table 14.1. Feed and product specifications for example one.

Desired product streams	Component flow rate (mol/hr) A	B	C	D	Product flow rate (mol/hr)
P4	0	0	0	15	15
P3	0	0	20	10	30
P2	10	12.5	0	0	22.5
P1	15	12.5	5	0	32.5
Component flow rate (mol/hr)	25	25	25	25	100

Given these material specifications, we want EXSEP to develop a number of technically feasible, economically attractive sequences.

C. Starting the Program

1. Keyboard Information Input

To start the program, we place the EXSEP diskette in the A: drive (A: is the default drive) and enter:

```
A> EXSEP
```

The program begins and the large "EXSEP" logo appears on the screen. EXSEP then provides us with our first menu choice: we can enter data automatically from a **file**, or manually through the **screen** and keyboard. Figure 14.1 shows the screen for this choice. Since this is our first time using EXSEP, we choose "Screen" to enter data manually. EXSEP responds to the choice by placing a new window on the screen, titled **Data Input From Keyboard**.

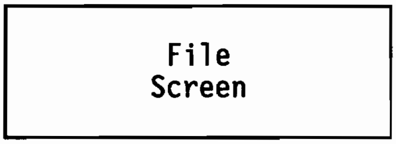


Figure 14.1. Method choice.

An interactive session now begins where we input the required information. The following dialogue results:

```
How many products are in the system?
4
How many components are in the system?
4

What is the name of component 1?
c4
What is the name of component 2?
c5
What is the name of component 3?
c6
What is the name of component 4?
c7

What is the name of product 1?
p1
What is the name of product 2?
p2
What is the name of product 3?
```

```
p3
What is the name of product 4?
p4
```

```
What is the flow of c4 in p1?
15
What is the flow of c5 in p1?
12.5
What is the flow of c6 in p1?
5
What is the flow of c7 in p1?
0
```

```
What is the flow of c4 in p2
10
What is the flow of c5 in p2?
12.5
What is the flow of c6 in p2?
0
What is the flow of c7 in p2?
0
```

```
What is the flow of c4 in p3?
0
What is the flow of c5 in p3?
0
What is the flow of c6 in p3?
20
What is the flow of c7 in p3?
10
```

```
What is the flow of c4 in p4
0
What is the flow of c5 in p4?
0
What is the flow of c6 in p4?
0
What is the flow of c7 in p4?
15
```

```
What is the normal boiling point (°C) of c4?
-0.49
What is the K-value of c4?
1.3
What is the normal boiling point (°C) of c5?
36.08
What is the K-value of c5?
0.39
```

```
What is the normal boiling point (°C) of c6?
68.75
What is the K-value of c6?
0.14
What is the normal boiling point (°C) of c7?
98.44
What is the K-value of c7?
0.05
```

```
Are any components hazardous or corrosive?
no
Are there any pseudoproducts?
no
Do you wish to save this information to a file?
yes
Please enter file name:
a: ch14.dat
```

Based on the last answer, EXSEP saves the information in the form of Prolog clauses in the file CH14.DAT. Figure 14.2 lists the contents of file CH14.DAT. Note that CH14.DAT *is an ASCII text file*. Therefore, we could bypass the data input session and use a word processor to write these clauses and save them as a text file.

2. File Information Input

Once we have stored information on a file, we can use that file as input for convenience. On EXSEP's initial menu choice (Figure 14.1), we simply choose **File** rather than **Screen**. EXSEP then asks which file to read.

All files created and read by EXSEP must have the suffix "DAT". We created the data file CH14.DAT. The EXSEP disk has additional data files, including EX1MSS.DAT, EX2.DAT, and EX3.DAT. These files define different separation problems.

When we use the **File** input mode, EXSEP lists all files on the disk that end with "DAT." To choose a file, we simply highlight it within the menu and hit the enter key. EXSEP reads the file and does a **reconsult** with the clauses to place them into the program.

```
flow(p1,c4,15).
flow(p1,c5,12.5).
flow(p1,c6,5).
flow(p1,c7,0).
flow(p2,c4,10).
flow(p2,c5,12.5).
flow(p2,c6,0).
flow(p2,c7,0).
flow(p3,c4,0).
flow(p3,c5,0).
flow(p3,c6,20).
flow(p3,c7,10).
flow(p4,c4,0).
flow(p4,c5,0).
flow(p4,c6,0).
flow(p4,c7,15).
k_value(c4,1.3).
k_value(c5,0.39).
k_value(c6,0.14).
k_value(c7,0.05).
boiling_temp(c4,-0.49).
boiling_temp(c5,36.08).
boiling_temp(c6,68.75).
boiling_temp(c7,98.44).
initial_components([c4,c5,c6,c7]).
initial_set([p1,p2,p3,p4]).
corrosive(none).
```

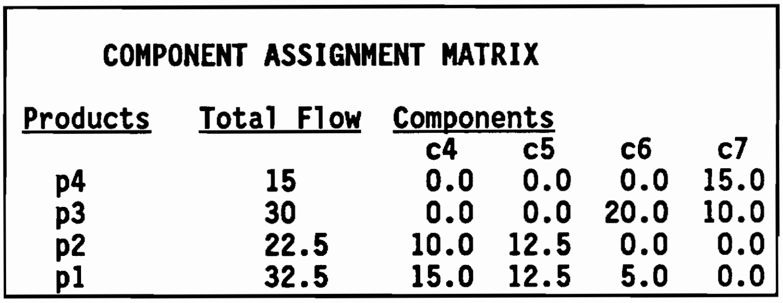
Figure 14.2. File CH14.DAT after the data input session.

D. Running the Program

After reading the information (and saving it, if desired), EXSEP automatically moves into problem analysis. The **data input window** disappears and EXSEP makes the component assignment matrix (CAM). When EXSEP builds the CAM, it creates the **Component Assignment Matrix** window. We then see the CAM on the screen, as shown in Figure 14.3.

After displaying the CAM, EXSEP pauses to let us read the screen. We hit the **ENTER** key when we are ready to continue. EXSEP removes the **Component Assignment Matrix** window and moves on to bypass analysis, creating the **Bypass Analysis** window.

If we recall from 13.5A, we know that bypass is only possible when we have an all-component-inclusive product. Looking at Figure 14.3, however, we find no all-component-inclusive product. Consequently, in the **Bypass Analysis** window EXSEP reports that bypass is not possible with this CAM. EXSEP then lists the component flow rates to the separator, as shown in Figure 14.4. This problem has equimolar flows of n-butane, n-pentane, n-hexane, and n-heptane (at 25 mol/hr each).

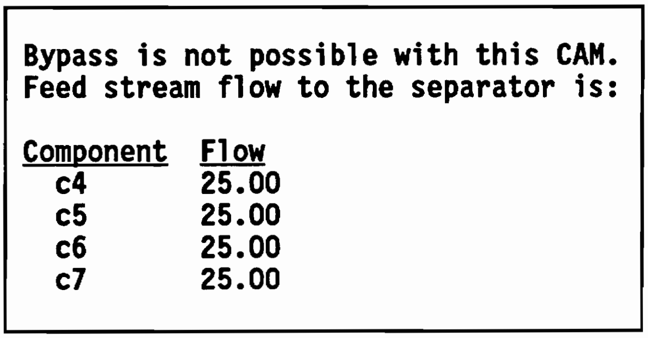


```
          COMPONENT ASSIGNMENT MATRIX

Products      Total Flow   Components
                              c4      c5     c6     c7
   p4            15          0.0     0.0    0.0   15.0
   p3            30          0.0     0.0   20.0   10.0
   p2            22.5       10.0    12.5    0.0    0.0
   p1            32.5       15.0    12.5    5.0    0.0
```

Figure 14.3. Component assignment matrix for example one.

With bypass analysis complete, EXSEP again pauses to let us see the results. When we press the **ENTER** key to continue, EXSEP removes the bypass window and creates the **Separation Specification Table Window**. It then calculates the separation specification table (SST).

Depending on the size of the problem, this step may take the longest. With a large problem (e.g., 10 components and 7 products), calculating the SST can take anywhere from 2-15 seconds depending on computer speed.

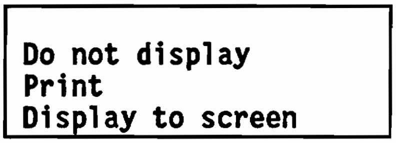


```
Bypass is not possible with this CAM.
Feed stream flow to the separator is:

Component  Flow
   c4      25.00
   c5      25.00
   c6      25.00
   c7      25.00
```

Figure 14.4. EXSEP's bypass analysis.

Once finished, EXSEP reports that the SST is complete, and asks us (via the menu) how we want the SST displayed. As the menu in Figure 13.5 indicates, we may choose to not display the SST at all, print it, or display it on the screen.

To display the SST on the screen, we highlight the **Display to screen** option and hit **ENTER**. EXSEP now reports the feasibility analysis for all horizontal and vertical splits. The analysis contains:

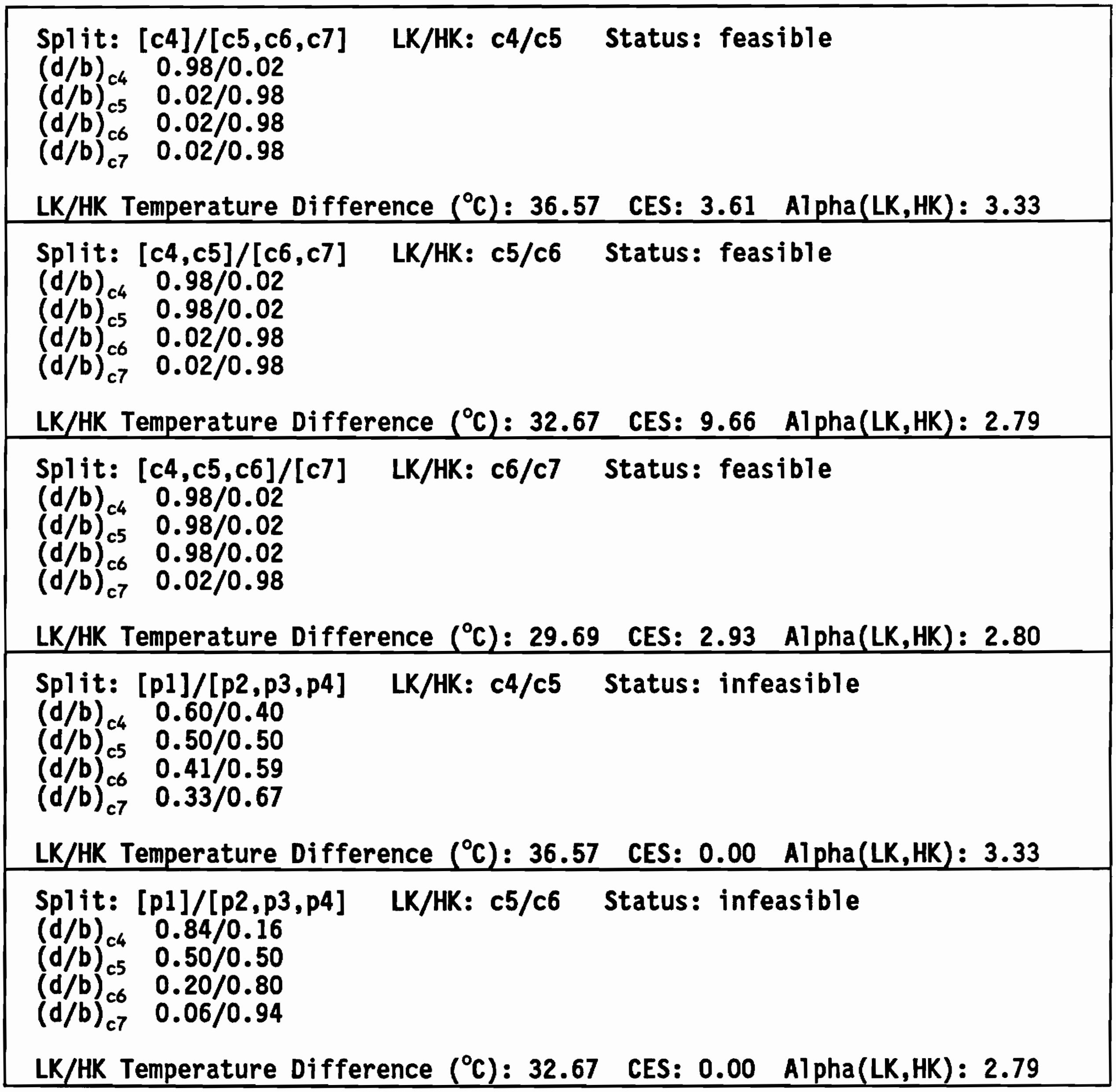


Do not display
Print
Display to screen

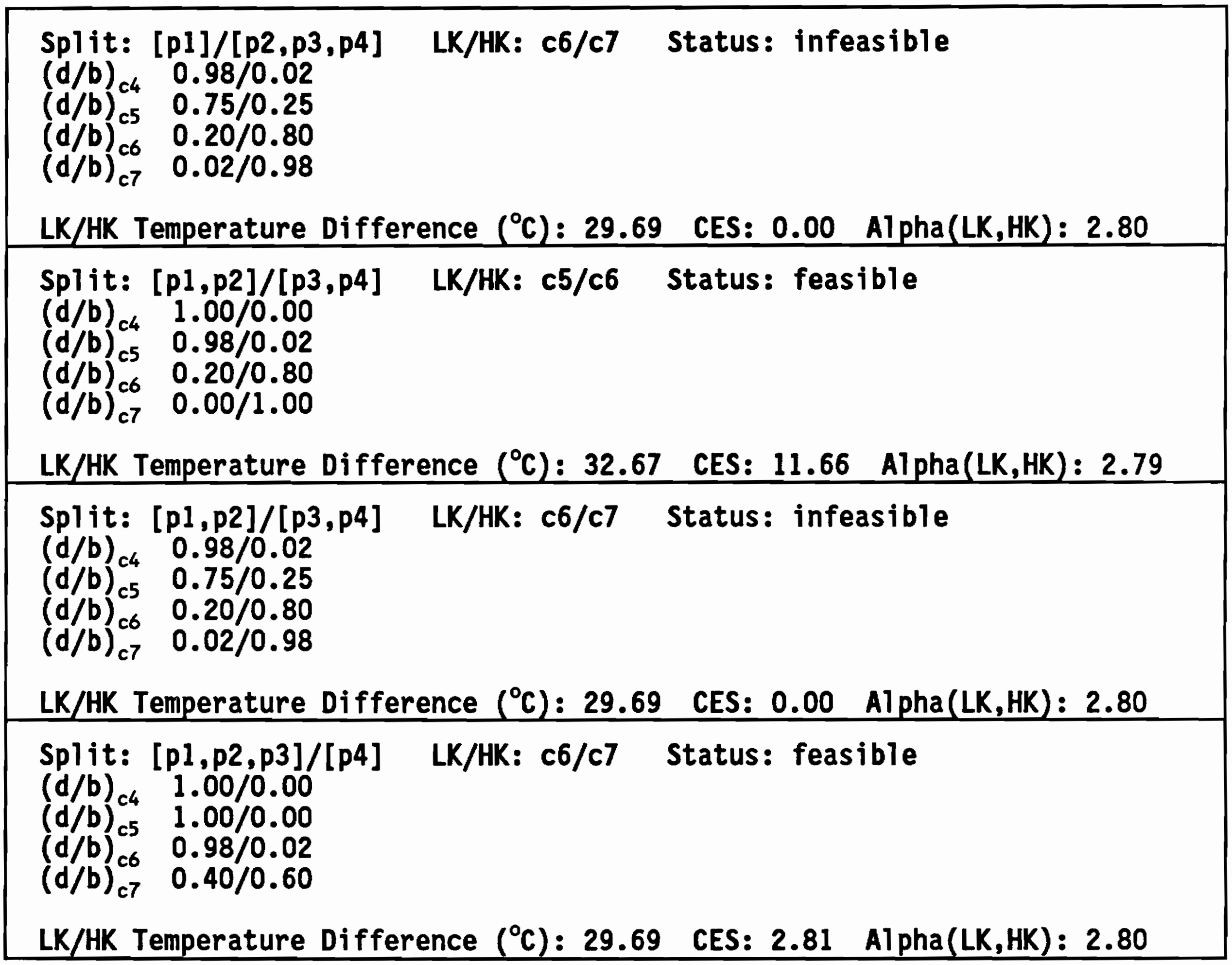
Figure 14.5. SST display menu options.

- The separation of interest (e.g., sharp split c4/c567 is displayed as [c4]/[c5, c6, c7]).
- The light-key (LK) and heavy-key (HK) components for the split.
- The distillate (d) and bottoms (b) recovery ratios for each component ($(d/b)_{c4}$, $(d/b)_{c5}$, etc.). Note that for nonkey components, the value of d/b is estimated from the shortcut analysis using the Fenske Equation.
- Whether the split is feasible or infeasible.
- The LK/HK normal boiling-point temperature difference. This value represents Δ in the equation for the CES (coefficient of ease of separation).
- The relative volatility between the LK and HK components ($\alpha_{LK,HK}$, or alpha(LK,HK)).
- The CES for feasible splits.

EXSEP displays the analysis for each split, and then pauses to let us assimilate the results. It analyzes and displays all horizontal and vertical splits are analyzed and displayed. The result is shown in Figure 14.6.



```
Split: [c4]/[c5,c6,c7]    LK/HK: c4/c5    Status: feasible
(d/b)c4  0.98/0.02
(d/b)c5  0.02/0.98
(d/b)c6  0.02/0.98
(d/b)c7  0.02/0.98

LK/HK Temperature Difference (°C): 36.57  CES: 3.61  Alpha(LK,HK): 3.33

Split: [c4,c5]/[c6,c7]    LK/HK: c5/c6    Status: feasible
(d/b)c4  0.98/0.02
(d/b)c5  0.98/0.02
(d/b)c6  0.02/0.98
(d/b)c7  0.02/0.98

LK/HK Temperature Difference (°C): 32.67  CES: 9.66  Alpha(LK,HK): 2.79

Split: [c4,c5,c6]/[c7]    LK/HK: c6/c7    Status: feasible
(d/b)c4  0.98/0.02
(d/b)c5  0.98/0.02
(d/b)c6  0.98/0.02
(d/b)c7  0.02/0.98

LK/HK Temperature Difference (°C): 29.69  CES: 2.93  Alpha(LK,HK): 2.80

Split: [p1]/[p2,p3,p4]    LK/HK: c4/c5    Status: infeasible
(d/b)c4  0.60/0.40
(d/b)c5  0.50/0.50
(d/b)c6  0.41/0.59
(d/b)c7  0.33/0.67

LK/HK Temperature Difference (°C): 36.57  CES: 0.00  Alpha(LK,HK): 3.33

Split: [p1]/[p2,p3,p4]    LK/HK: c5/c6    Status: infeasible
(d/b)c4  0.84/0.16
(d/b)c5  0.50/0.50
(d/b)c6  0.20/0.80
(d/b)c7  0.06/0.94

LK/HK Temperature Difference (°C): 32.67  CES: 0.00  Alpha(LK,HK): 2.79
```



```
Split: [p1]/[p2,p3,p4]    LK/HK: c6/c7    Status: infeasible
(d/b)c4  0.98/0.02
(d/b)c5  0.75/0.25
(d/b)c6  0.20/0.80
(d/b)c7  0.02/0.98

LK/HK Temperature Difference (°C): 29.69  CES: 0.00  Alpha(LK,HK): 2.80

Split: [p1,p2]/[p3,p4]    LK/HK: c5/c6    Status: feasible
(d/b)c4  1.00/0.00
(d/b)c5  0.98/0.02
(d/b)c6  0.20/0.80
(d/b)c7  0.00/1.00

LK/HK Temperature Difference (°C): 32.67  CES: 11.66  Alpha(LK,HK): 2.79

Split: [p1,p2]/[p3,p4]    LK/HK: c6/c7    Status: infeasible
(d/b)c4  0.98/0.02
(d/b)c5  0.75/0.25
(d/b)c6  0.20/0.80
(d/b)c7  0.02/0.98

LK/HK Temperature Difference (°C): 29.69  CES: 0.00  Alpha(LK,HK): 2.80

Split: [p1,p2,p3]/[p4]    LK/HK: c6/c7    Status: feasible
(d/b)c4  1.00/0.00
(d/b)c5  1.00/0.00
(d/b)c6  0.98/0.02
(d/b)c7  0.40/0.60

LK/HK Temperature Difference (°C): 29.69  CES: 2.81  Alpha(LK,HK): 2.80
```

Figure 14.6. Output from the separation specification table.

EXSEP has now rejected all infeasible splits and generated a set of feasible ones, and is now ready to perform heuristic analysis. The heuristics will give us the best split from the set of feasible splits.

At this point, EXSEP removes the SST window and creates the **Heuristic Analysis** window. The system favors ordinary distillation (heuristic M1), so EXSEP jumps to heuristic S1 (note that heuristic M2, avoid vacuum distillation and refrigeration, is implemented off-line based

```
** Heuristic S1: Remove corrosive and hazardous materials first **
******************************************************************

No corrosive elements exist.
It is recommended to try other heuristics.
```

Figure 14.7. EXSEP's report on heuristic S1.

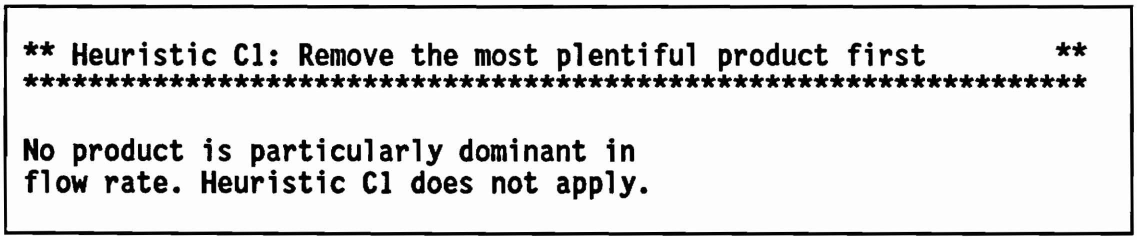
on bubble-point and dew-point calculations). Heuristic S1 says to remove corrosive and hazardous components first. Since the feed has no particularly corrosive or hazardous component, EXSEP recommends that we apply other heuristics, as shown in Figure 14.7. EXSEP pauses, and we hit the **ENTER** key to continue.

The next heuristic is S2, perform difficult separations last. For our problem, we have n-butane, n-pentane, n-hexane, and n-heptane in the feed stream. None of these materials is particularly close to any other in volatility; all the relative volatilities are than 2.70. In addition, the difference between their normal boiling points is large (> 10 °C). Since there are no particularly difficult splits, heuristic S2 does not apply. EXSEP reports this conclusion, as shown in Figure 14.8.

```
** Heuristic S2: Perform difficult separations last             **
******************************************************************

No splits are particularly difficult.
Heuristic S2 does not apply, and no splits
are identified as essential last splits.
```

Figure 14.8. EXSEP's report on heuristic S2.

EXSEP now moves on to the composition heuristics, C1 and C2. These two heuristics are key in recommending the best split with the given CAM. We go to heuristic C1, remove the most plentiful product first. EXSEP reports that no product is particularly plentiful or dominant in flow rate. For EXSEP to treat a product as the most plentiful, the flow rate must be at least 20% higher than the product with the second highest flow rate. Here, product p1, at 32.5 mol/hr, is only 8.3% higher in flow rate than product p3, at 30/mol/hr. Since no product is particularly plentiful, heuristic C1 does not apply. EXSEP reports this conclusion, as shown in Figure 14.9.



```
** Heuristic C1: Remove the most plentiful product first       **
******************************************************************

No product is particularly dominant in
flow rate. Heuristic C1 does not apply.
```

Figure 14.9. EXSEP's report on heuristic C1.

Remember that in section 13.7B, we considered product p1 most plentiful and did apply heuristic C1. Here, EXSEP is saying that C1 is not applicable. Which conclusion is the more accurate? At this point, we do not know. We must remember the nature of heuristics -- their applicability can at times be "fuzzy," as is the case with heuristic C1 here. Is product p1, with a molar flow rate of 32.5 mol/hr, the "most plentiful" when product p3 has a molar flow rate of 30.0 mol/hr?

To resolve this uncertainty, we must develop several feasible initial sequences and perform rigorous costing analysis. For one flowsheet, we apply heuristic C1. For the next flowsheet, we override it. The end result is a number of good, competitive flowsheets which will undergo rigorous costing before we choose the final sequence.

For this first flowsheet, we let EXSEP assume that heuristic C1 does not apply. After reporting the conclusion from heuristic C1, EXSEP pauses for us. We hit the **ENTER** key to continue the analysis. EXSEP then moves on to heuristic C2: favor a 50/50 split. This heuristic is the deciding factor for the initial split. EXSEP performs its analysis, and reports:

- The split of interest.
- The LK and HK components.
- The $\alpha_{LK,HK}$.
- The distillate and bottoms flow rates.
- The molar split ratio.
- The CES.

The molar split ratio is the most important factor in choosing a split. We choose the split with a ratio closest to 50/50. If two splits have approximately the same ratio, we choose the split with the highest CES.

EXSEP displays the results of its analysis in tabular form. It displays pertinent parameters for every split still under consideration. For this problem, we consider three sharp splits (i.e., c4/c567, c45/c67, c456/c7) and two sloppy splits (i.e., p12/p34 and p123/p4). Table 14.2 shows the results of EXSEP's analysis.

Table 14.2. EXSEP's report on heuristic C2.

Split	LK	HK	$\alpha_{LK,HK}$	Flow rate		Molar split ratio	CES
				Overhead	Bottoms		
c4/c567	c4	c5	3.33	25.0	75.0	75/25	3.61
c45/c67	c5	c6	2.79	50.0	50.0	50/50	9.66
c456/c7	c6	c7	2.80	75.0	25.0	75/25	2.93
p12/p34	c5	c6	2.79	55.0	45.0	55/45	11.66
p123/p4	c6	c7	2.80	85.0	15.0	85/15	2.81

After reporting these results, EXSEP recommends a split. The screen reads:

```
Split [p1,p2]/[p3,p4] with:
      -- components c5/c6 as LK/HK
      -- a CES = 11.66
      -- a molar split ratio = 55/45
is the recommended split. Do you
wish to make this split?
```

From the analysis, splits c45/c67 and p12/p34 are the closest to a 50/50 molar split ratio, with values of 50/50 and 55/45, respectively. But the p12/p34 split has the higher CES, and is therefore the split of choice. EXSEP's menu system allows us to accept or reject the split recommendation; as shown in Figure 14.10, we can choose **Split** or **Do Not Split** via the menu.

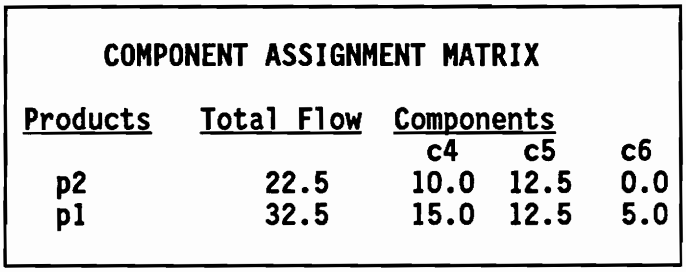
Figure 14.10. Split menu options.

In this case, we will accept the split recommendation. We highlight "Split," choice, and hit the **ENTER** key. EXSEP executes the split and separator number one (S1) is specified as:

```
          25 c4
          25 c5
          5 c6
25 c4  ↗
25 c5
25 c6
25 c7  ↘
          20 c6
          25 c7
```

EXSEP now moves on to separate the overhead stream from S1 (25 mol/hr c4, 25 mol/hr c5, and 5 mol/hr c6). It sets up a new CAM, shown in Figure 14.11.

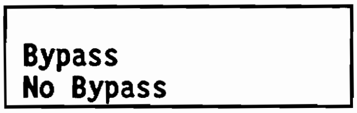
EXSEP pauses for us. When we hit **ENTER**, EXSEP proceeds to bypass analysis. Product p1 is all-component-inclusive and therefore is subjected to bypass. EXSEP asks, via its menu system, if we desire to bypass or not, as shown in Figure 14.12.



```
         COMPONENT ASSIGNMENT MATRIX

Products    Total Flow  Components
                          c4     c5     c6
   p2          22.5      10.0   12.5    0.0
   p1          32.5      15.0   12.5    5.0
```

Figure 14.11. CAM for overhead stream from S1.

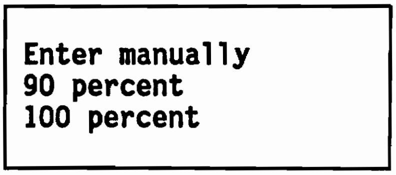
In this case, we choose **Bypass**. EXSEP then asks how much to bypass, as shown in Figure 14.13.



Bypass
No Bypass

Figure 14.12. EXSEP's bypass menu.

The material balance limits the maximum amount of material we may bypass. The feed stream to the separator has component flow rates of 25 mol/hr c4 + 25 mol/hr c5 + 5 mol/hr c6. Based on the CAM, 60% of the c4, 50% of the c5, and 100% of the c6 go to product p1 (the all-component-inclusive product). Component



```
Enter manually
90 percent
100 percent
```

Figure 14.13. EXSEP's menu for the amount of bypass

c5 is the *limiting component*, and the maximum amount we may bypass and still maintain the material specifications, is 50% of the stream, or 12.5 mo/hr c4 + 12.5 mol/hr c5 + 2.5 mol/hr c6 to p1. When EXSEP says **100 percent bypass**, it means 100% of the maximum allowable amount, or 100% of the limiting component (in this case, c5). The bypass stream would be 12.5 mol/hr c4, 12.5 mol/hr c5, 2.5 mol/hr c6. Similarly, a **90 percent bypass** will bypass 90% of the maximum flow for the limiting component, or 11.25 mol/hr c4 + 11.25 mol/hr c5 + 2.25 mol/hr c6. We will choose to bypass the maximum amount. We highlight **100 percent** on the bypass menu, and hit **ENTER**. EXSEP then recalculates the material balance, and reports the flow rates in the bypass stream and the feed stream to the separator. The screen reads:

```
Bypass flow to product p1 is:
Component    Flow Rate
   c4          12.50
   c5          12.50
   c6           2.50


Feed stream flow rate to separator is:
Component    Flow Rate
   c4          12.50
   c5          12.50
   c6           2.50
```

EXSEP again pauses for us. When we hit **ENTER**, EXSEP recalculates the CAM, since bypass changes the material balance. Figure 14.14 shows the new CAM that EXSEP displays on the screen.

```
                COMPONENT ASSIGNMENT MATRIX

Products    Total Flow  Components
                           c4    c5     c6
  p2           22.5      10.0  12.5    0.0
  pl            5.0       2.5   0.0    2.5
```

Figure 14.14. CAM after bypass.

EXSEP now moves on to the SST for this CAM. We have a three-component, two-product system, with three splits theoretically available: (1) the sharp split c4/c56, (2) the sharp split c45/c6, and (3) the sloppy split pl/p2.

EXSEP calculates the SST and asks us how to display it. If we highlight **Display to screen**, and hit **ENTER**, the screen displays the SST shown in Figure 14.15.

```
Split: [c4]/[c5,c6]    LK/HK: c4/c5   Status: feasible
(d/b)c4  0.98/0.02
(d/b)c5  0.02/0.98
(d/b)c6  0.02/0.98

LK/HK Temperature Difference (°C): 36.57  CES: 9.02  Alpha(LK,HK): 3.33
-----------------------------------------------------------------------
Split: [c4,c5]/[c6]    LK/HK: c5/c6   Status: feasible
(d/b)c4  0.98/0.02
(d/b)c5  0.98/0.02
(d/b)c6  0.02/0.98

LK/HK Temperature Difference (°C): 32.67  CES: 0.97  Alpha(LK,HK): 2.79
-----------------------------------------------------------------------
Split: [pl]/[p2]       LK/HK: any/any   Status: infeasible
(d/b)c4  0.20/0.80
(d/b)c5  0.02/0.98
(d/b)c6  0.98/0.02

LK/HK Temperature Difference (°C): 0.00   CES: 0.00  Alpha(LK,HK): 0.00
```

Figure 14.15 Output from the separation specification table.

The sloppy split pl/p2 is infeasible because it fails the component recovery-specification test (thermodynamically, we cannot recover 98% of n-hexane and only a 2% of n-pentane in the overhead, since n-pentane is more volatile than n-hexane).

EXSEP now moves on to the heuristic analysis. It tells us that heuristics S1 (remove hazardous or corrosive materials first) and S2 (perform difficult separations last) do not apply. However, heuristic C1 (remove the most plentiful product first) does apply. EXSEP reports:

```
Most plentiful product is p2. The sharp
split [c4]/[c5,c6] with c4/c5 as LK/HK
violates heuristic C1: remove the most
plentiful product first. Do you wish to
remove this split from consideration or
maintain it?
```

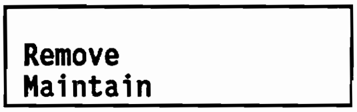


Figure 14.16. Heuristic C1 menu.

As Figure 14.16 shows, we may **Remove** this split from consideration (since it violates heuristic C1), or override the heuristic and **Maintain** the split as a choice. We will follow the heuristic and eliminate this split from consideration by highlighting **Remove** and hitting **ENTER**. EXSEP removes this split from consideration (in Prolog terms, EXSEP **retracts** the split from the database).

EXSEP now continues with on to heuristic C2 (favor a 50/50 split). We see that there is only one feasible split remaining in consideration: c45/c6. EXSEP goes through heuristic C2 (which is trivial at this point), and then recommends the c45/c6 split. We see on the screen:

```
Split [c4,c5]/[c6] with:
      -- components c5/c6 as LK/HK
      -- a CES = 0.97
      -- a molar split ratio = 91/ 9
is the recommended split. Do you
```

wish to make this split?

The **Split** or **Do Not Split** menu choices appear on the screen (see Figure 14.10). We will choose to make the split rather than override the recommendation (it is, after all, the only available split). Separator S2 is:

```
                12.5 c4
                12.5 c5
              ↗
      12.5 c4
      12.5 c5                              (Bypass: 12.5 c4 + 12.5 c5 + 2.5 c6
       2.5 c6                               to product p1.)
              ↘
               2.5 c6
```

EXSEP is not done. It immediately moves to recursively process the overhead of S2. Figure 14.17 shows the new CAM.

EXSEP continues its analysis, and reports that product p2 is all-component-inclusive and subjected to bypass. We decide to bypass the maximum amount: 100% of the limiting component, c4. EXSEP bypasses 10 mol/hr of c4 and 10 mol/hr of c5 to product p2, and then recalculates the CAM. Figure 14.18 shows the CAM after bypass.

COMPONENT ASSIGNMENT MATRIX

Products	Total Flow	Components c4	c5
p2	22.5	10.0	12.5
p1	2.5	2.5	0.0

Figure 14.17. CAM for overhead stream from separator S2.

The decision here is easy: the horizontal split p1/p2 is the same as the vertical split c4/c5. EXSEP goes through the SST and the heuristics, and recommends the sharp split c4/c5. The screen reads:

```
Split [c4]/[c5] with:
   -- components c4/c5 as LK/HK
   -- a CES = 10.82
   -- a molar split ratio = 50/50
   is the recommended split. Do you
   wish to make this split?
```

COMPONENT ASSIGNMENT MATRIX

Products	Total Flow	Components	
		c4	c5
p2	2.5	0.0	2.5
p1	2.5	2.5	0.0

Figure 14.18. CAM for overhead stream after bypass.

We highlight **Split** on the menu and hit **ENTER**. EXSEP executes the split. Our third separator, S3, is specified as:

```
               2.5 c4
             ↗
      2.5 c4               (Bypass: 10.0 c4 + 10.0 c5 to
      2.5 c5                        product p1)
             ↘
               2.5 c5
```

Now that we have completed all overhead processing, EXSEP backs up (recursively in the Prolog program) to process the bottoms streams. EXSEP first checks the bottoms stream from S3, then S2, and finally S1. The bottoms from S3 is 2.5 mol/hr c5. Since the stream is pure, no further processing is needed. EXSEP then looks at the bottoms of S2; the flow here is 2.5 mol/hr c6. Since this stream is also pure, no further processing is needed. Finally, EXSEP looks at the bottoms from S1. This stream has a flow rate of 20 mol/hr c6 and 25 mol/hr c7, as shown in the CAM in Figure 14.19.

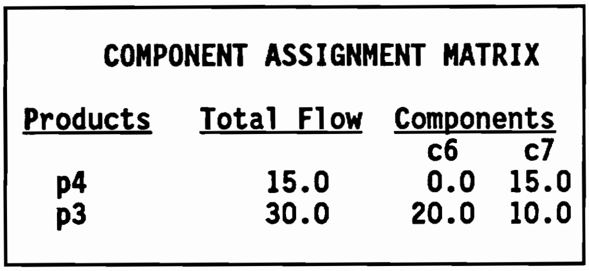
Based on this CAM, EXSEP does a bypass analysis. Product p3 is all-component-inclusive and subjected to bypass. Component c6 is the limiting component. A 100% of c6 sends 8.00 mol/hr of c5 and 10.00 mol/hr of c6 going to p3.

Figure 14.20 shows the new CAM after bypass.

This CAM closely resembles that of Figure 14.18. Both the horizontal and vertical splits represent the same sharp split, c6/c7.

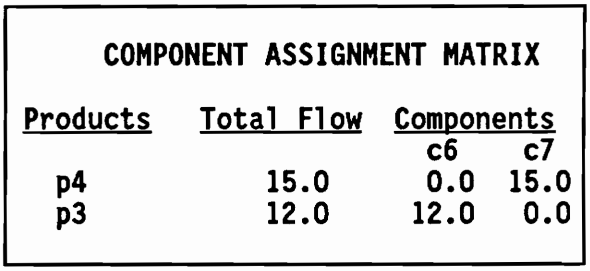
EXSEP goes through the SST and the heuristics, and recommends split c6/c7. The screen reads:

```
Split [c6]/[c7] with:
   -- components c6/c7 as LK/HK
   -- a CES = 70.3
   -- a molar split ratio = 56/44
   is the recommended split. Do you
   wish to make this split?
```



```
        COMPONENT ASSIGNMENT MATRIX

Products     Total Flow   Components
                            c6     c7
   p4           15.0        0.0   15.0
   p3           30.0       20.0   10.0
```

Figure 14.19. CAM for bottoms stream from separator S1.



```
        COMPONENT ASSIGNMENT MATRIX

Products     Total Flow   Components
                            c6     c7
   p4           15.0        0.0   15.0
   p3           12.0       12.0    0.0
```

Figure 14.20. CAM for bottoms stream after bypass.

We highlight **Split** and hit **ENTER**. EXSEP has developed the entire flowsheet, and summarizes the results in tabular form, as shown in Table 14.3.

Table 14.3. Flowsheet of initial sequence for example one.

Separator	Bypass around	Stream flow in	Overhead	Bottoms
S1	none	25.0 c4 25.0 c5 25.0 c6 25.0 c7	25.0 c4 25.0 c5 5.0 c6	20.0 c6 25.0 c7
S2	pl: 12.5 c4 12.5 c5 2.5 c6	12.5 c4 12.5 c5 2.5 c6	12.5 c4 12.5 c5	2.5 c6
S3	p2: 10.0 c4 10.0 c5	2.5 c4 2.5 c5	2.5 c4	2.5 c5
S4	p3: 8.0 c6 10.0 c7	12.0 c6 15.0 c7	12.0 c6	15.0 c7

Figure 14.21 presents a graphical representation of the same flowsheet. Note that EXSEP does not draw graphical flowsheets. Its output is in tabular form only.

The next section discusses how to generate alternate flowsheets.

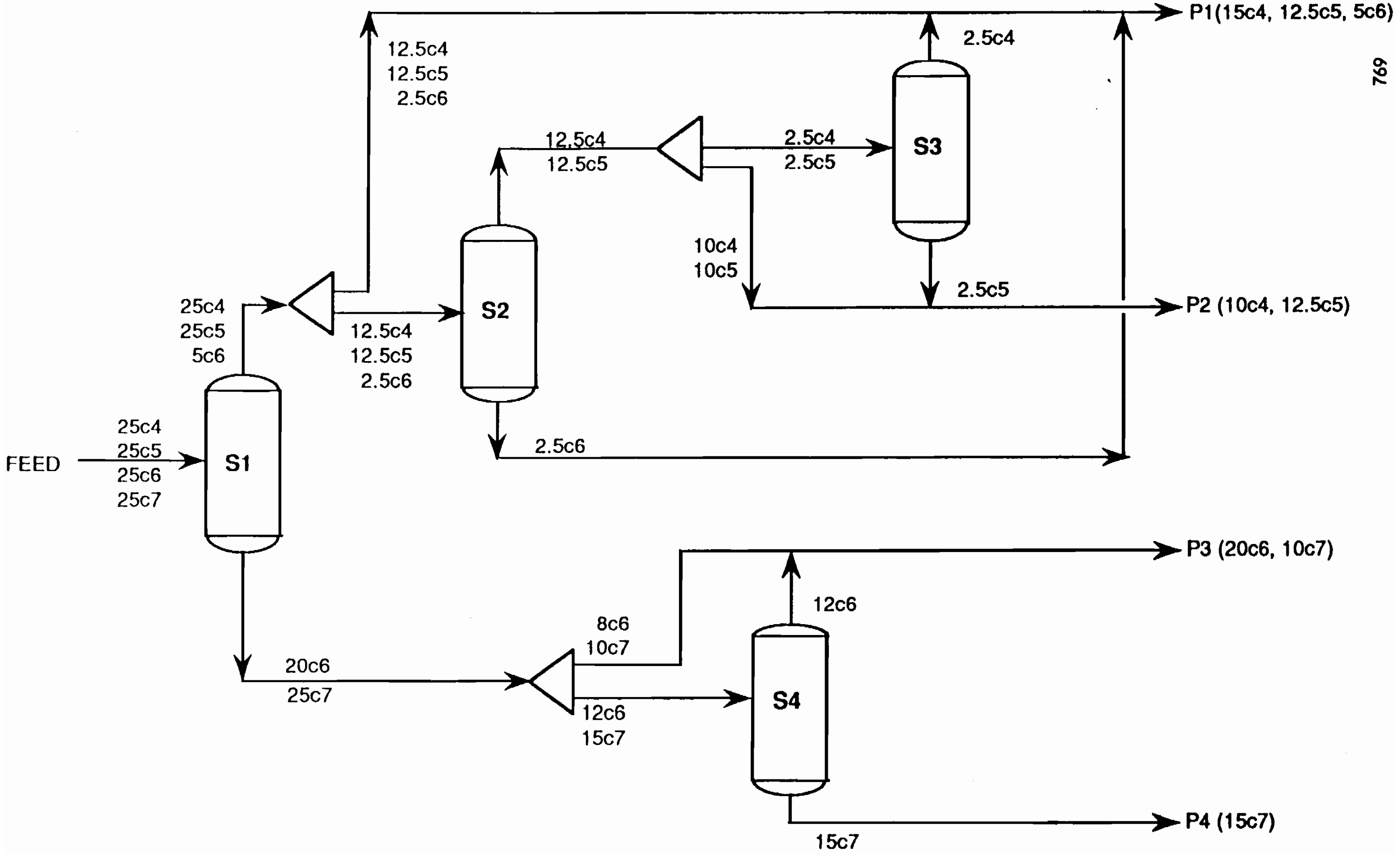


Figure 14.21. Flowsheet of initial sequence for example one.

14.3 GENERATING ALTERNATE FLOWSHEETS

We can easily generate alternate flowsheets with EXSEP, simply by rejecting one or more of its recommendations. EXSEP will then look for alternate solutions. If we desire, we can even manually choose the split we want at any given point of flowsheet development.

Let us demonstrate this process using the CAM shown in Figure 14.3. EXSEP goes through the same bypass analysis (bypass is not possible) and generates the SST of Figure 14.6. As before, EXSEP then works through the heuristics, producing the heuristic C2 report seen in Table 14.2. Finally, EXSEP then recommends the p12/p34 split.

At this point, to generate an alternate flowsheet, we override EXSEP's recommendation. We select **Do Not Split** from the menu. EXSEP then asks us if we would like to execute a split **Heuristically** (i.e., begin the heuristics all over again) or **Manually** (i.e., specify our own split). This menu choice is shown in Figure 14.22.

Heuristically
Manually

Figure 14.22. Split search choice menu.

We highlight **Manually** and hit **ENTER**. Choosing **Manually** allows us to manually override the heuristics. In chapter 13, we decided to apply heuristic C1, with product p1 as most plentiful, even though its flow rate is only 8.3% above that of product p3. In the initial flowsheet, EXSEP decided not to apply heuristic C1, since the flow rates were within 20% of each other. For our alternate sequence, however, we will apply heuristic

C1. As in chapter 13, heuristic C1 leads us to reject splits c4/c567, c45/c67, and p12/p34; we are left with p123/p4 and c456/c7 as possible choices. We choose split c456/c7, since its molar split ratio is close to a 50/50, and it has the higher CES.

Let us execute this split in EXSEP. After we choose the **Manual** option, EXSEP brings up splits on the screen and asks if we want to execute them. We first see:

```
If desired, sharp split [c4]/[c5,c6,c7]
with a CES = 3.61 can be done. Please enter your choice.
```

We highlight **Do Not Split**. We then see on the screen,

```
If desired, sharp split [c4,c5]/[c6,c7]
with a CES = 9.66 can be done. Please enter your choice.
```

We again enter **Do Not Split**. We then see on the screen:

```
If desired, sharp split [c4,c5,c6]/[c7]
with a CES = 2.93 can be done. Please enter your choice.
```

Since this is the split we want, we highlight **Split** and hit **ENTER**. Separator S1 is specified as:

```
         25 c4
         25 c5
         25 c6
25 c4  ↗
25 c5
25 c6
25 c7  ↘
         25 c7
```

EXSEP executes the split and calculates a new overhead CAM, as shown in Figure 14.23. After displaying the CAM on the screen, EXSEP performs the bypass analysis, indicating that product pl is all-component-inclusive. We agree to bypass 100%, or 5 mol/hr c4 + 5 mol/hr c5 + 5 mol/hr c6. The new CAM is shown in Figure 14.24.

COMPONENT ASSIGNMENT MATRIX

Products	Total Flow	Components		
		c4	c5	c6
p3	20	0.0	0.0	20.0
p2	22.5	10.0	12.5	0.0
pl	32.5	15.0	12.5	5.0

Figure 14.23. Component assignment matrix for overhead from separator S1.

Now EXSEP calculates the SST. Horizontal split pl2/p3 and sharp split c45/c6 are identical. Sloppy split pl/p23 is infeasible. EXSEP then moves on to the heuristics. Heuristics S1 and S2 do not apply. EXSEP also says that heuristic C1 does not apply, since the flow-rate difference between product p2 (22.5 mol/hr) and p3 (20 mol/hr) is only 2.5 mol/hr, or 12.5%. Nevertheless, we will apply heuristic C1 rigorously in this flowsheet, and treat p2 as the most plentiful product. We reject split c4/c56 and choose split c45/c6 manually. EXSEP executes the split, resulting in the separator S2 specified by:

COMPONENT ASSIGNMENT MATRIX

Products	Total Flow	Components		
		c4	c5	c6
p3	20	0.0	0.0	20.0
p2	22.5	10.0	12.5	0.0
pl	17.5	10.0	7.5	0.0

Figure 14.24. Component assignment matrix for overhead from S1 after bypass.

```
          20 c4
          20 c5
20 c4   ↗
20 c5
20 c6   ↘
          20 c6
```

After executing the split, EXSEP calculates a new overhead CAM, shown in Figure 14.25. After performing bypass analysis on this CAM, EXSEP reports that products p1 and p2 are both all-component-inclusive and subjected to bypass. We will bypass 100% of the allowable amount for each product, or 7.5 mol/hr c4 + 7.5 mol/hr c5 to p1, and 10.0 mol/hr c4 + 10.0 mol/hr c5 to product p2.

COMPONENT ASSIGNMENT MATRIX

Products	Total Flow	Components	
		c4	c5
p2	22.5	10.0	12.5
p1	17.5	10.0	7.5

Figure 14.25. Component assignment matrix for overhead from S2.

EXSEP then generates the CAM shown in Figure 14.26. Splits p1/p2 and c4/c5 represent the same sharp split, and we accept EXSEP's recommendation (the c4/c5 split) to finish off the sequence. EXSEP gives us a summary of the entire sequence, shown in Table 14.4.

COMPONENT ASSIGNMENT MATRIX

Products	Total Flow	Components	
		c4	c5
p2	2.5	0.0	2.5
p1	2.5	2.5	0.0

Figure 14.26. Component assignment matrix for overhead from S2 after bypass.

Table 14.4. Flowsheet of an alternate sequence for example one.

Separator	Bypass around	Stream flow in	Overhead	Bottoms
S1	none	25.0 c4 25.0 c5 25.0 c6 25.0 c7	25.0 c4 25.0 c5 25.0 c6	25.0 c7
S2	pl: 5.0 c4 5.0 c5 5.0 c6	20.0 c4 20.0 c5 20.0 c6	20.0 c4 20.0 c5	20.0 c6
S3	pl: 7.5 c4 7.5 c5 p2: 10.0 c4 10.0 c5	2.5 c4 2.5 c5	2.5 c4	2.5 c5

Figure 14.27 gives a graphical representation of the flowsheet. This sequence is identical to the initial sequence developed in section 13.7 (Figure 13.11). Compared to the first sequence developed in this chapter (Figure 14.20), this second flowsheet has one less separator. The first sequence has four separators, one sloppy, and three sharp, while the second sequence has only three separators, all sharp.

In a typical engineering design scenario, we may use EXSEP to generate additional alternate sequences. We have already so, as Figures 14.28, 29, and 30 demonstrate. After developing several feasible initial flowsheets, we perform a rigorous cost analysis on each to help us select the final process flowsheet.

To summarize, we followed EXSEP's recommendations to generate an initial flowsheet (Figure 14.20). We then generated alternate flowsheets (Figure 14.27, 28, 29, and 30) by selectively overriding EXSEP's heuristic recommendations and choosing our splits manually. After executing any

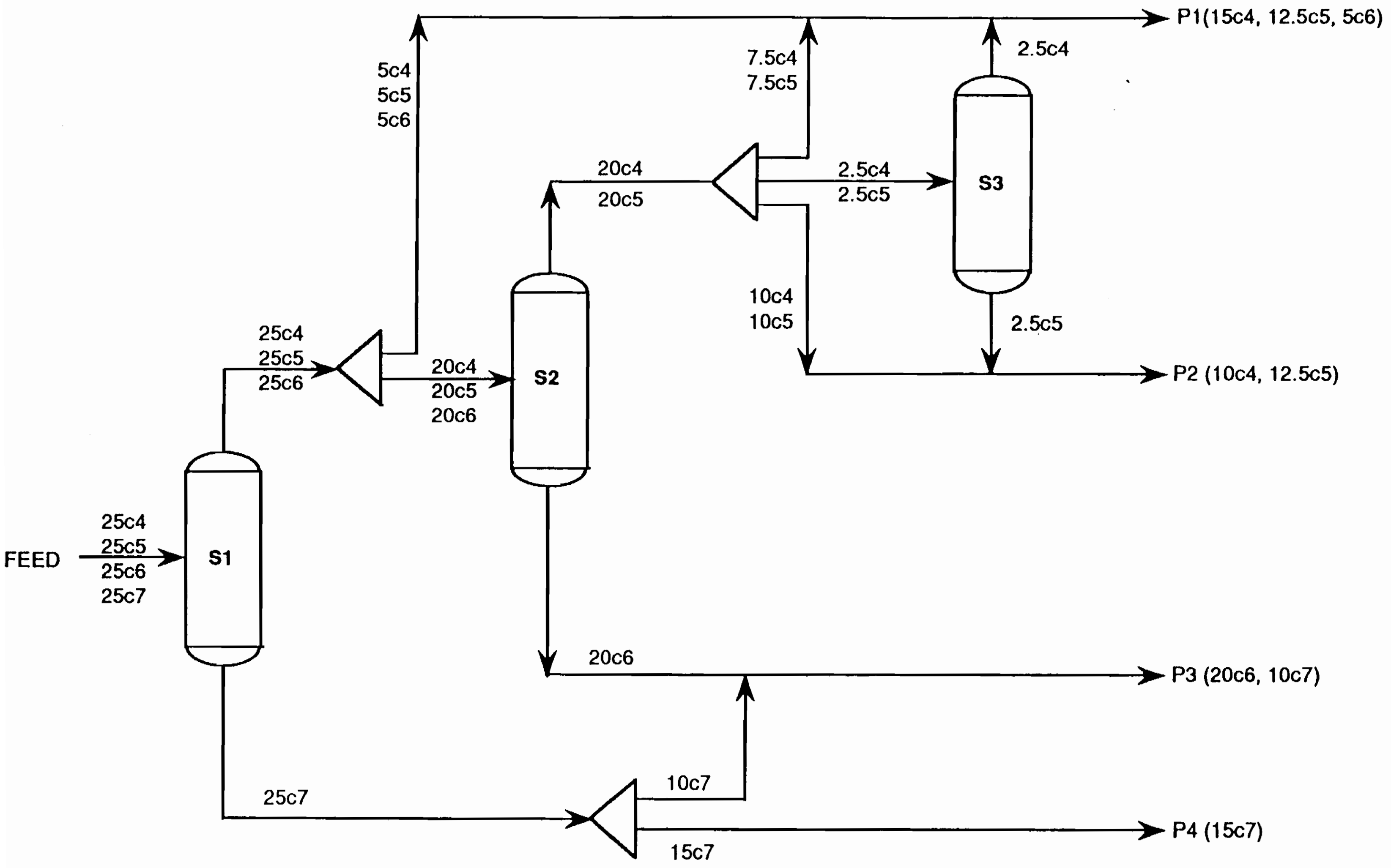


Figure 14.27. Flowsheet of second sequence for example one.

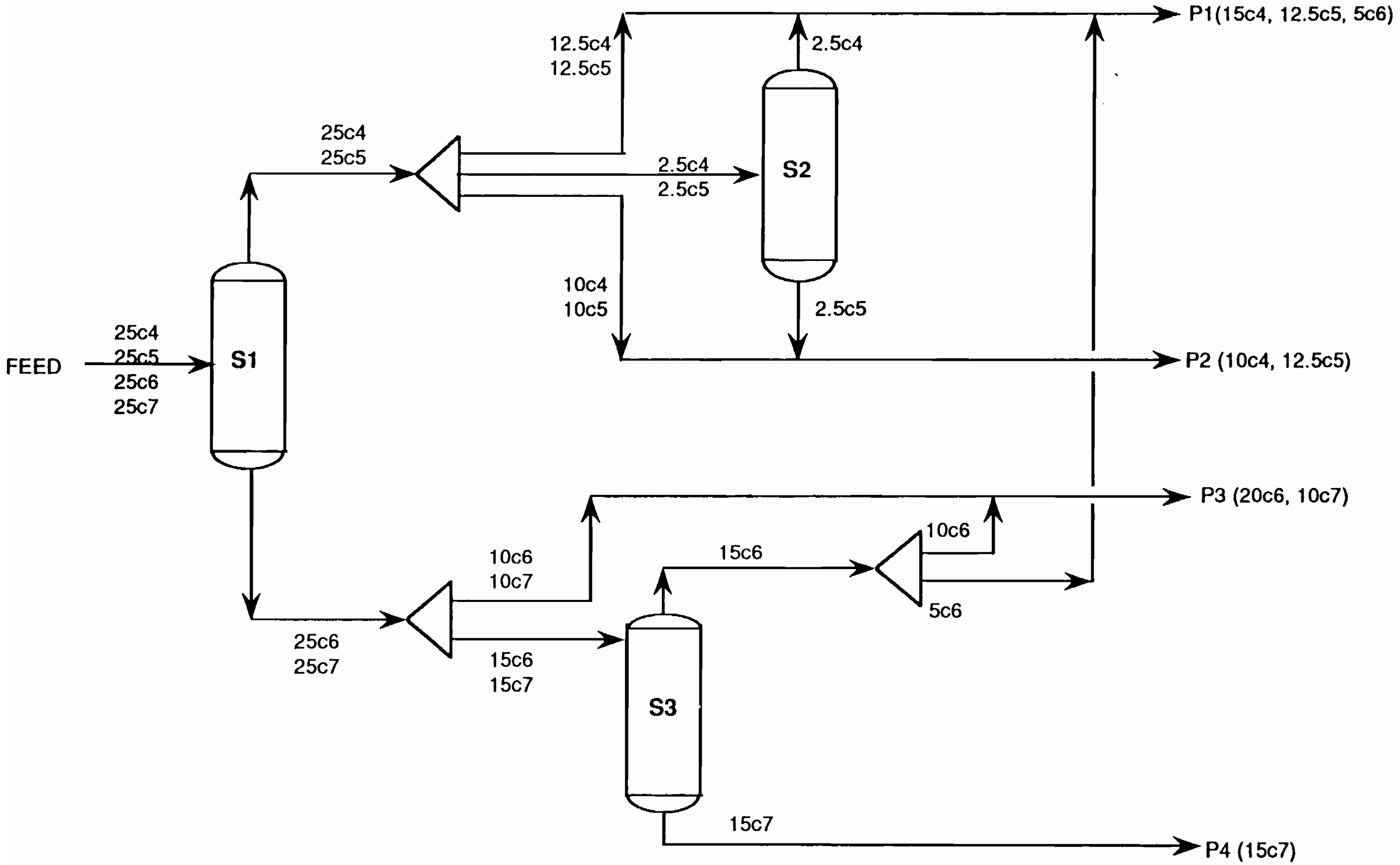


Figure 14.28. Alternate flowsheet for example one.

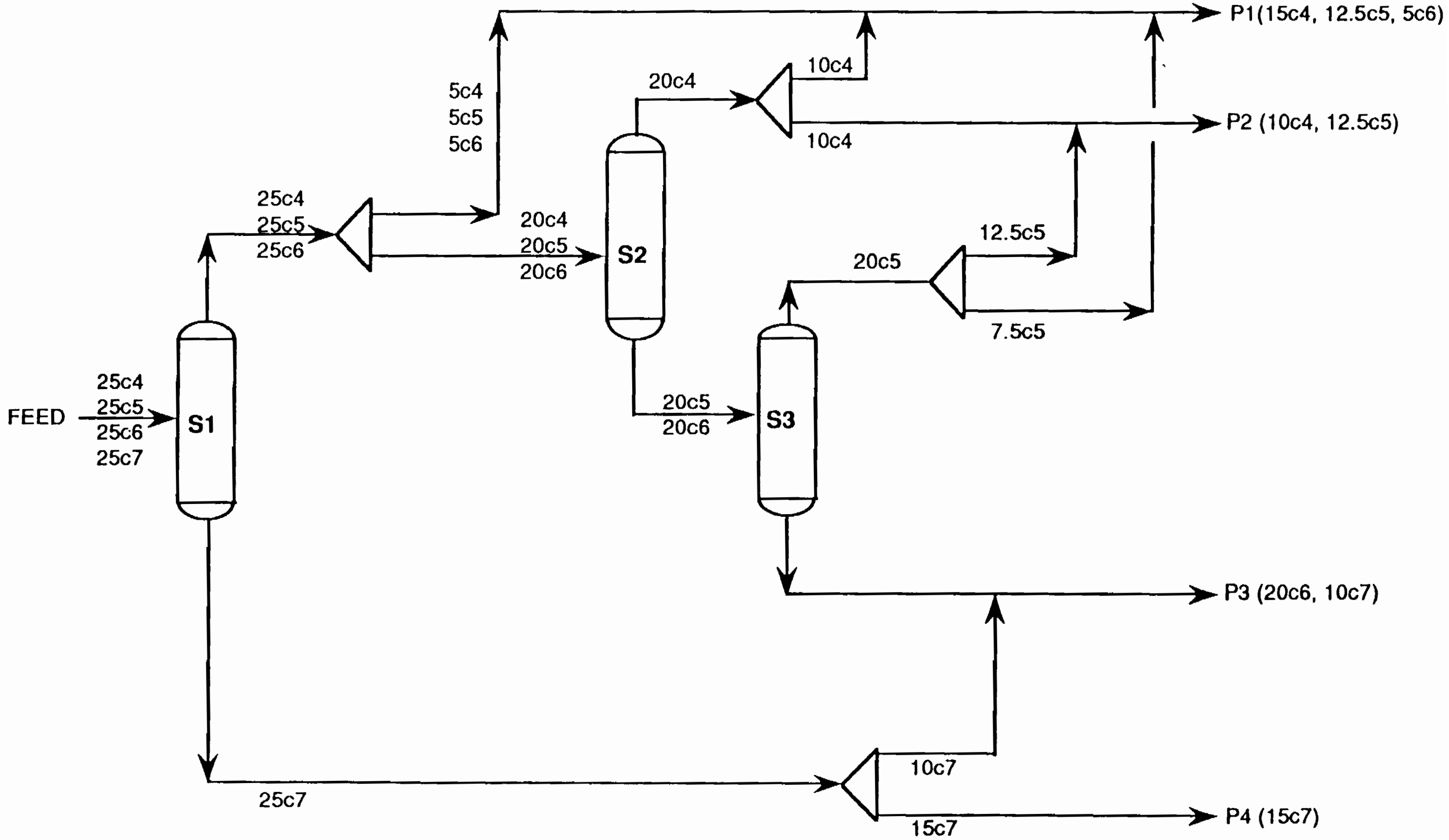


Figure 14.29. Alternate flowsheet for example one.

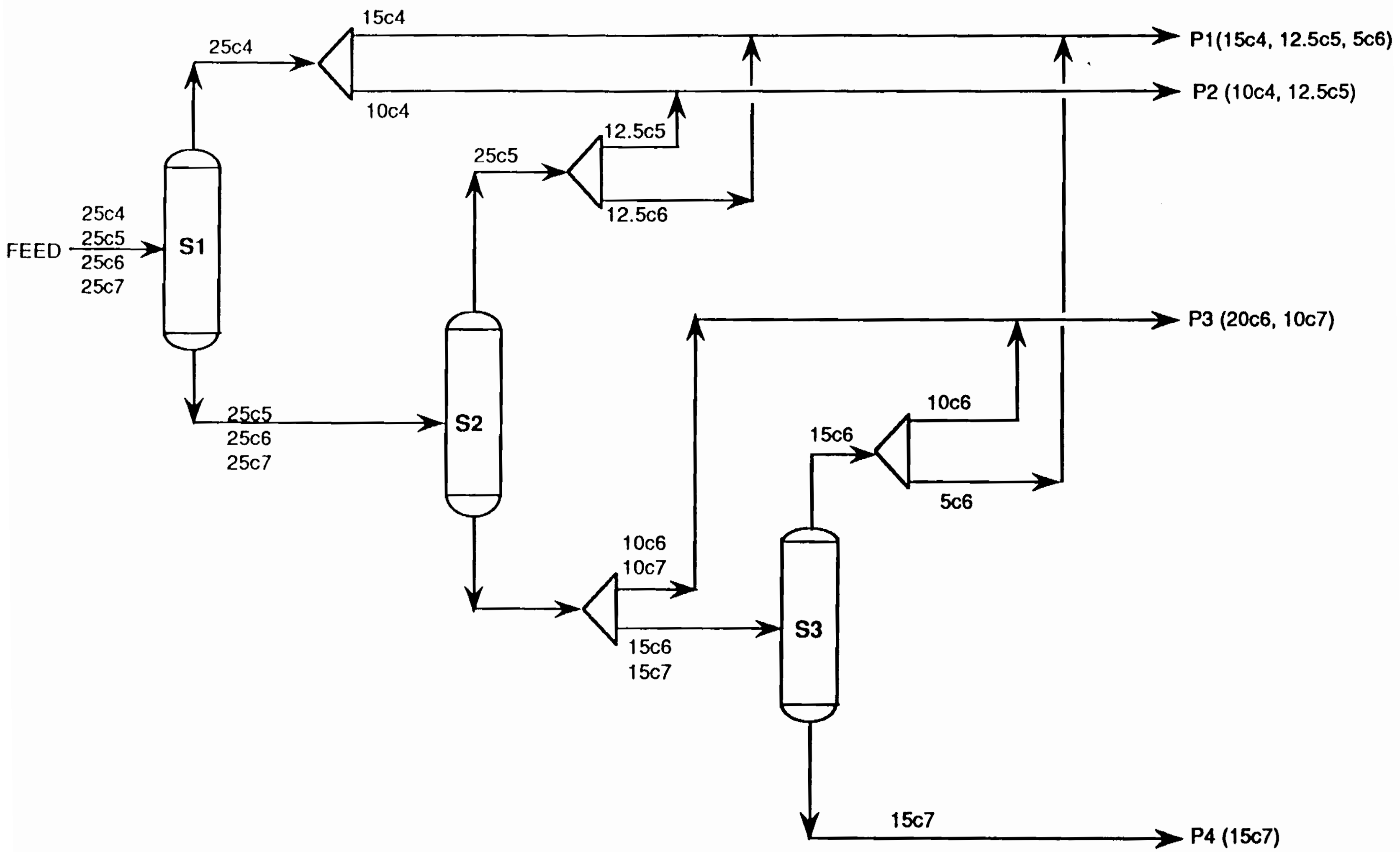


Figure 14.30. Alternate flowsheet for example one.

manual split, EXSEP continues analyzing the rest of the problem using the heuristics.

To end the EXSEP session, we hit the **Ctrl-Break** keys simultaneously.

14.4 ADDITIONAL EXAMPLE PROBLEMS

A. Example 2: Example One Modified to Include Pseudoproducts

In chapter 13, section 13.5 shows the conversion of the CAM of example one to include pseudoproducts. We convert product p1 into two pseudoproducts, p1* and p1′, such that when blended together, they form product p1. Importantly, the pseudoproduct transformation is done such that previously infeasible horizontal splits are made feasible. For example one, our transformed CAM, including pseudoproducts, is:

$$\begin{array}{c|cccc|}
 & c4 & c5 & c6 & c7 \\
\hline
p4 & 0 & 0 & 0 & 15 \\
p3 & 0 & 0 & 20 & 10 \\
p1' & 0 & 12.5 & 5 & 0 \\
p2 & 10 & 12.5 & 0 & 0 \\
p1* & 15 & 0 & 0 & 0
\end{array}$$

Given this CAM, we go through the data-acquisition process with a pseudoproduct. We again enter:

```
A> EXSEP
```

to begin EXSEP, and then highlight **Screen** as our choice for data loading.

The data-acquisition process produces the following dialogue:

```
How many products are in the system?
5
How many components are in the system?
4

What is the name of component 1?
c4
What is the name of component 2?
c5
What is the name of component 3?
c6
What is the name of component 4?
c7

What is the name of product 1?
pl*
What is the name of product 2?
p2
What is the name of product 3?
pl'
What is the name of product 4?
p3
What is the name of product 5?
p4

What is the flow of c4 in pl*?
15
What is the flow of c5 in pl*?
0
What is the flow of c6 in pl*?
0
What is the flow of c7 in pl*?
0

What is the flow of c4 in p2?
10
What is the flow of c5 in p2?
12.5
What is the flow of c6 in p2?
0
What is the flow of c7 in p2?
0

What is the flow of c4 in pl'?
0
```

```
What is the flow of c5 in p1'?
12.5
What is the flow of c6 in p1'?
5
What is the flow of c7 in p1'?
0
```

```
What is the flow of c4 in p3?
0
What is the flow of c5 in p3?
0
What is the flow of c6 in p3?
20
What is the flow of c7 in p3?
10
```

```
What is the flow of c4 in p4?
0
What is the flow of c5 in p4?
0
What is the flow of c6 in p4?
0
What is the flow of c7 in p4?
15
```

```
What is the normal boiling point (°C) of c4?
-0.49
What is the K-value of c4?
1.3
What is the normal boiling point (°C) of c5?
36.08
What is the K-value of c5?
0.39
What is the normal boiling point (°C) of c6?
68.75
What is the K-value of c6?
0.14
What is the normal boiling point (°C) of c7?
98.44
What is the K-value of c7?
0.05
```

```
Are any components hazardous or corrosive?
no
Are there any pseudoproducts?
yes
How many pseudoproducts are there?
1
```

```
What is the name of the actual product?
pl
Is pl* a pseudoproduct of pl?
yes
Is p2 a pseudoproduct of pl?
no
Is pl' a pseudoproduct of pl?
yes
Is p3 a pseudoproduct of pl?
no
Is p4 a pseudoproduct of pl?
no
Do you wish to save this file?
yes
Please enter file name:
a: chl4p.dat
```

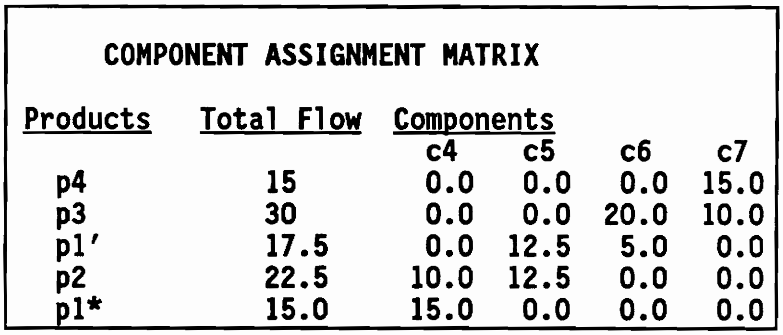
EXSEP saves the file in Prolog clause form, as shown in Figure 14.31.

```
flow(pl*,c4,15).
flow(p2,c4,10).
flow(pl',c4,0).
flow(p3,c4,0).
flow(p4,c4,0).
flow(pl*,c5,0).
flow(p2,c5,12.5).
flow(pl',c5,12.5).
flow(p3,c5,0).
flow(p4,c5,0).
flow(pl*,c6,0).
flow(p2,c6,0).
flow(pl',c6,5).
flow(p3,c6,20).
flow(p4,c6,0).
flow(pl*,c7,0).
flow(p2,c7,0).
flow(pl',c7,0).
flow(p3,c7,10).
flow(p4,c7,15).
k_value(c4,1.3).
k_value(c5,0.39).
k_value(c6,0.14).
k_value(c7,0.05).
boiling_temp(c4,-0.49).
boiling_temp(c5,36.08).
boiling_temp(c6,68.75).
boiling_temp(c7,98.44).
initial_set([pl*,p2,pl',p3,p4]).
initial_components([c4,c5,c6,c7]).
corrosive(none).
pseudoproduct([pl*,pl'],pl).
```

Figure 14.31. File CH14P.DAT after the data-input session.

After saving the file, EXSEP builds and displays the CAM, shown in Figure 14.32.

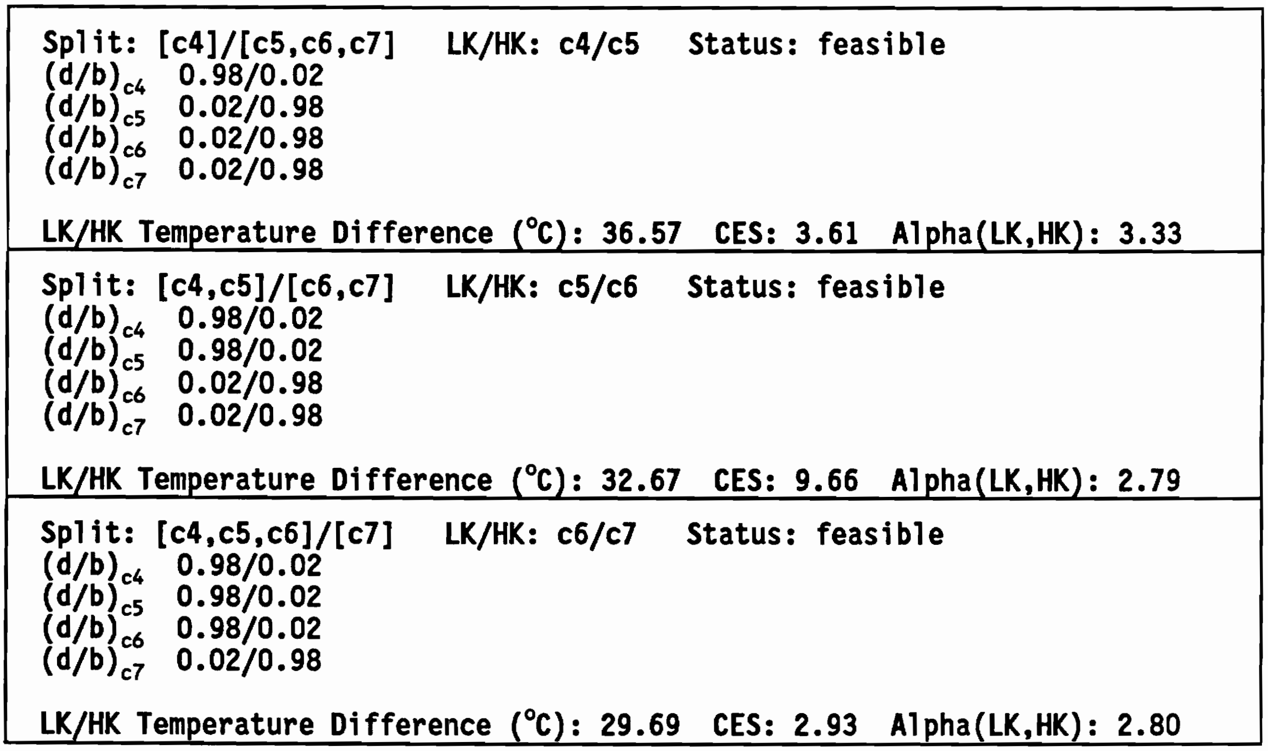
EXSEP then pauses until we hit **ENTER.** It performs bypass analysis, reports that bypass is not possible, and moves on to the separation specification table, shown in Figure 14.33.



COMPONENT ASSIGNMENT MATRIX

Products	Total Flow	Components			
		c4	c5	c6	c7
p4	15	0.0	0.0	0.0	15.0
p3	30	0.0	0.0	20.0	10.0
p1′	17.5	0.0	12.5	5.0	0.0
p2	22.5	10.0	12.5	0.0	0.0
p1*	15.0	15.0	0.0	0.0	0.0

Figure 14.32. Component assignment matrix for example one with pseudoproduct.



Split: [c4]/[c5,c6,c7]   LK/HK: c4/c5   Status: feasible
$(d/b)_{c4}$ 0.98/0.02
$(d/b)_{c5}$ 0.02/0.98
$(d/b)_{c6}$ 0.02/0.98
$(d/b)_{c7}$ 0.02/0.98

LK/HK Temperature Difference (°C): 36.57   CES: 3.61   Alpha(LK,HK): 3.33

Split: [c4,c5]/[c6,c7]   LK/HK: c5/c6   Status: feasible
$(d/b)_{c4}$ 0.98/0.02
$(d/b)_{c5}$ 0.98/0.02
$(d/b)_{c6}$ 0.02/0.98
$(d/b)_{c7}$ 0.02/0.98

LK/HK Temperature Difference (°C): 32.67   CES: 9.66   Alpha(LK,HK): 2.79

Split: [c4,c5,c6]/[c7]   LK/HK: c6/c7   Status: feasible
$(d/b)_{c4}$ 0.98/0.02
$(d/b)_{c5}$ 0.98/0.02
$(d/b)_{c6}$ 0.98/0.02
$(d/b)_{c7}$ 0.02/0.98

LK/HK Temperature Difference (°C): 29.69   CES: 2.93   Alpha(LK,HK): 2.80

```
Split: [p1*]/[p2,p1',p3,p4]   LK/HK: c4/c5   Status: feasible
(d/b)c4  0.60/0.40
(d/b)c5  0.02/0.98
(d/b)c6  0.00/1.00
(d/b)c7  0.00/1.00

LK/HK Temperature Difference (°C): 36.57  CES: 3.46  Alpha(LK,HK): 3.33
------------------------------------------------------------------------
Split: [p1*,p2]/[p1',p3,p4]   LK/HK: c4/c5   Status: feasible
(d/b)c4  0.98/0.02
(d/b)c5  0.50/0.50
(d/b)c6  0.04/0.96
(d/b)c7  0.00/1.00

LK/HK Temperature Difference (°C): 36.57  CES: 12.98  Alpha(LK,HK): 3.33
------------------------------------------------------------------------
Split: [p1*,p2]/[p1',p3,p4]   LK/HK: c5/c6   Status: feasible
(d/b)c4  0.99/0.01
(d/b)c5  0.50/0.50
(d/b)c6  0.02/0.98
(d/b)c7  0.00/1.00

LK/HK Temperature Difference (°C): 32.67  CES: 11.60  Alpha(LK,HK): 2.79
------------------------------------------------------------------------
Split: [p1*,p2,p1']/[p3,p4]   LK/HK: c5/c6   Status: feasible
(d/b)c4  1.00/0.00
(d/b)c5  0.98/0.02
(d/b)c6  0.20/0.80
(d/b)c7  0.00/1.00

LK/HK Temperature Difference (°C): 32.67  CES: 11.66  Alpha(LK,HK): 2.79
------------------------------------------------------------------------
Split: [p1*,p2,p1']/[p3,p4]   LK/HK: c6/c7   Status: infeasible
(d/b)c4  0.98/0.02
(d/b)c5  0.75/0.25
(d/b)c6  0.20/0.80
(d/b)c7  0.02/0.98

LK/HK Temperature Difference (°C): 29.69  CES: 0.00  Alpha(LK,HK): 2.80
------------------------------------------------------------------------
Split: [p1*,p2,p1',p3]/[p4]   LK/HK: c6/c7   Status: feasible
(d/b)c4  1.00/0.00
(d/b)c5  1.00/0.00
(d/b)c6  0.98/0.02
(d/b)c7  0.40/0.60

LK/HK Temperature Difference (°C): 29.69  CES: 2.81  Alpha(LK,HK): 2.80
```

Figure 14.33. Output from the separation specification table.

After completing the SST, EXSEP performs heuristic analysis. Neither heuristic S1 nor S2 applies (i.e., no component is hazardous or corrosive, nor are there any difficult, essential last separations). This result is not surprising, since neither heuristic S1 nor S2 applied in example one when we did not include a pseudoproduct. EXSEP then moves on to heuristics C1 and C2. EXSEP identifies product p3 as the most plentiful, and identifies splits that violate heuristic C1, as shown in Figure 14.34.

```
Most plentiful product is p3. The sloppy
split [p1*]/[p2,p1',p3,p4] with c4/c5 as LK/HK
violates heuristic C1: remove most plentiful
product first. Do you wish to remove this
split from consideration or maintain it?

Most plentiful product is p3. The sloppy
split [p1*,p2]/[p1',p3,p4] with c4/c5 as LK/HK
violates heuristic C1: remove most plentiful
product first. Do you wish to remove this
split from consideration or maintain it?

Most plentiful product is p3. The sloppy
split [p1*]/[p2,p1',p3,p4] with c5/c6 as LK/HK
violates heuristic C1: remove most plentiful
product first. Do you wish to remove this
split from consideration or maintain it?

Most plentiful product is p3. The sharp
split [c4,c5,c6]/[c7] with c6/c6 as LK/HK
violates heuristic C1: remove most plentiful
product first. Do you wish to remove this
split from consideration or maintain it?
```

Figure 14.34. Heuristic C1 analysis.

For each split that violates Cl, EXSEP asks if we wish to remove or maintain this split as a possible choice via the **Remove/Maintain** menu system (Figure 14.16).

We remove a split from consideration if we desire to follow EXSEP's advice and satisfy heuristic Cl. We maintain the splits if we desire to override Cl. Here we will abide by Cl, and therefore choose **Remove** for every split listed in Figure 14.34.

After eliminating splits that violate heuristic Cl, EXSEP applies heuristic C2 (favor a 50/50 split, or the split with the highest CES). Table 14.5 shows the heuristic C2 report.

Table 14.5. EXSEP's report on heuristic C2.

Split	LK	HK	$\alpha_{LK,HK}$	Flow rate		Molar split ratio	CES
				Overhead	Bottoms		
c4/c567	c4	c5	3.33	25.0	75.0	75/25	3.61
c45/c67	c5	c6	2.79	50.0	50.0	50/50	9.66
pl*2/pl'34	c5	c6	2.79	55.0	45.0	55/45	11.66
pl*21'3/p4	c6	c7	2.80	85.0	15.0	85/15	2.81

Splits c45/c67 and pl*p2/pl'p3p4 are both close to 50/50. Since the sloppy split pl*p2/pl'p3p4 has the higher CES, it is the split of choice. Thus, EXSEP reports:

```
Split [pl*,p2,pl']/[p3/p4] with:
    -- components c5/c6 as LK/HK
    -- a CES = 11.66
    -- a molar split ratio = 55/45
    is the recommended split. Do you
    wish to make this split?
```

We accept this recommendation by highlighting **Split** and hitting **ENTER**. EXSEP executes the split, and our first separator, S1, is:

```
             25 c4
             25 c5
              5 c6
      25 c4 ↗
      25 c5
      25 c6
      25 c7 ↘
             20 c6
             25 c7
```

EXSEP now analyzes the overhead from S1. It calculates a new CAM, shown in Figure 14.35.

Looking at the CAM, no product appears to be all-component-inclusive, and bypass is impossible. However, note that products p1* and p1′ both ultimately end up in product p1. The combination of p1* and p1′ *is* all-component inclusive, and this CAM *is* subjected to bypass.

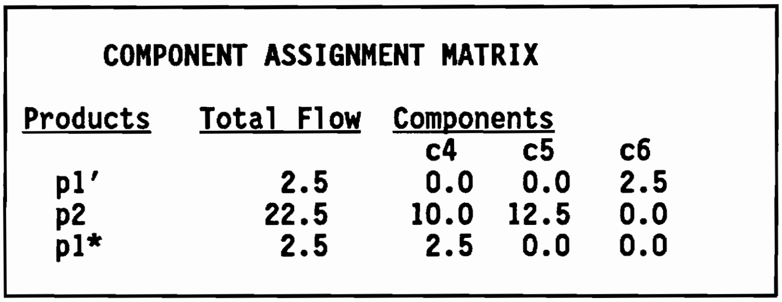
COMPONENT ASSIGNMENT MATRIX

Products	Total Flow	Components c4	 c5	 c6
p1′	17.5	0.0	12.5	5.0
p2	22.5	10.0	12.5	0.0
p1*	15.0	15.0	0.0	0.0

Figure 14.35. Component assignment matrix for overhead from first separator.

EXSEP detects this. We will choose to bypass the maximum amount, and EXSEP reports:

```
Bypass flow to product p1 is:
Component   Flow Rate
   c4         12.50
   c5         12.50
   c6          2.50
```



```
              COMPONENT ASSIGNMENT MATRIX

Products    Total Flow   Components
                          c4     c5     c6
  p1'          2.5        0.0    0.0    2.5
  p2          22.5       10.0   12.5    0.0
  p1*          2.5        2.5    0.0    0.0
```

Figure 14.36. Component assignment matrix for overhead from first separator after bypass.

EXSEP recalculates the CAM (Figure 14.36), develops the SST, and pauses. After allowing us to view the SST, EXSEP goes to the heuristic analysis. The most plentiful product is p2, and consequently, EXSEP recommends the rejection of split c4/c56. We accept EXSEP's recommendation, and highlight **Remove**, and hit **ENTER**. EXSEP removes the split.

Only two possible splits remain: sharp split c45/c6 and sloppy split p1*/p2p1' (with c4/c5 as LK/HK). EXSEP subjects these splits to heuristic C2, and Table 14.6 shows its report.

Table 14.6. EXSEP's report on heuristic C2 for the overhead stream.

Split	LK	HK	$\alpha_{LK,HK}$	Flow rate		Molar split ratio	CES
				Overhead	Bottoms		
c45/c6	c5	c6	2.79	25.0	2.5	91/9	0.97
p1*/p21'	c4	c5	3.33	2.5	25.0	91/9	3.36

EXSEP then recommends split p1*/p21', and we accept this recommendation. Separator S2 is:

```
                2.5 c4
       2.5 c4 ↗
      22.5 c5
       2.5 c6 ↘
              22.5 c5
               2.5 c6
```

COMPONENT ASSIGNMENT MATRIX

Products	Total Flow	Components		
		c4	c5	c6
p1′	2.5	0.0	0.0	2.5
p2	22.5	10.0	12.5	0.0

Figure 14.37. Component assignment matrix for bottoms from second separator.

EXSEP now analyzes the overhead from S2. The overhead is 2.5 mol/hr c4. Since this is a pure stream, no further processing is needed. EXSEP then goes back to analyze the bottoms from S2. Figure 14.37 shows the new CAM. The problem is getting fairly easy. We have only two splits available: c4/c56 and c45/c6. Note that both splits are sharp, and the vertical split c45/c6 is identical to the horizontal split p2/p1′.

EXSEP executes bypass analysis (no bypass is possible), builds the SST, and then implements the heuristics. Heuristics S1 and S2 do not apply. Heuristic C1 *does* apply. Product p2 is most plentiful, and EXSEP points out that split c4/c56 violates heuristic C1, so we remove it from consideration. The only remaining split is c45/c6. Executing this split gives separator S3:

```
            10.0 c4
            12.5 c5
10.0 c4  ↗
12.5 c5
 2.5 c6  ↘
            2.5 c6
```

No further processing of the overhead or bottoms is required from S3, since both streams go directly into products. Therefore, EXSEP backs up, assessing the bottoms stream from separator S1. We see the CAM in

Figure 14.38. EXSEP performs bypass analysis, and reports that product p3 is all-component-inclusive and subjected to bypass. We bypass at 100%, and EXSEP reports:

```
      COMPONENT ASSIGNMENT MATRIX

Products    Total Flow  Components
                          c6    c7
  p4           15.0      0.0  15.0
  p3           30.0     20.0  10.0
```

Figure 14.38. Component assignment matrix for bottoms from S1.

```
Bypass flow to product p3:
Component    Flow rate
   c6           8.0
   c7          10.0
```

EXSEP then displays the new CAM after bypass, as shown in Figure 14.39.

```
      COMPONENT ASSIGNMENT MATRIX

Products    Total Flow  Components
                          c6    c7
  p4           15.0      0.0  15.0
  p3           30.0     12.0   0.0
```

Figure 14.39. Component assignment matrix after bypass.

This CAM is straightforward. The only split possible is the sharp split c6/c7, which EXSEP recommends. We accept the recommendation, and complete the sequence. EXSEP summarizes the entire flowsheet, shown here in Table 14.7.

Table 14.7. Flowsheet of separation sequence with pseudoproduct.

Separator	Bypass around	Stream flow in	Overhead	Bottoms
S1	none	25.0 c4 25.0 c5 25.0 c6 25.0 c7	25.0 c4 25.0 c5 5.0 c6	20.0 c6 25.0 c7
S2	p1: 12.5 c4 12.5 c5 2.5 c6	12.5 c4 12.5 c5 2.5 c6	2.5 c4	10.0 c4 12.5 c5 2.5 c6
S3	none	10.0 c4 12.5 c5 2.5 c6	10.0 c4 12.5 c5	2.5 c6
S4	p3: 8.0 c6 10.0 c7	12.0 c6 15.0 c7	12.0 c6	15.0 c7

Figure 14.40 also presents the flowsheet graphically.

We can develop alternate flowsheets for the pseudoproduct example, too. We can again override heuristics and instruct EXSEP to execute certain splits. We have done this twice, and Figures 14.41 and 14.42 display the results.

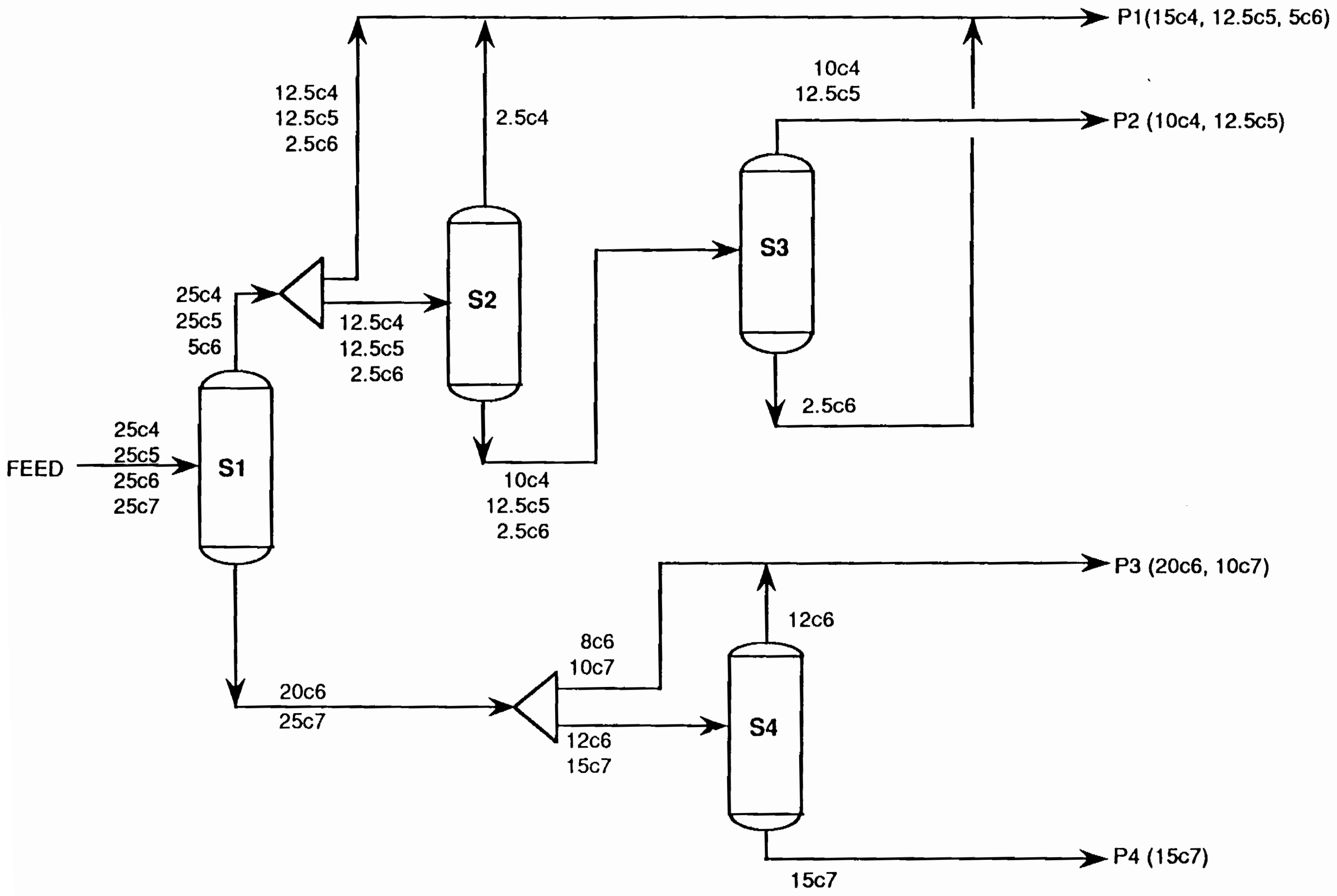


Figure 14.40. Flowsheet of pseudoproduct example.

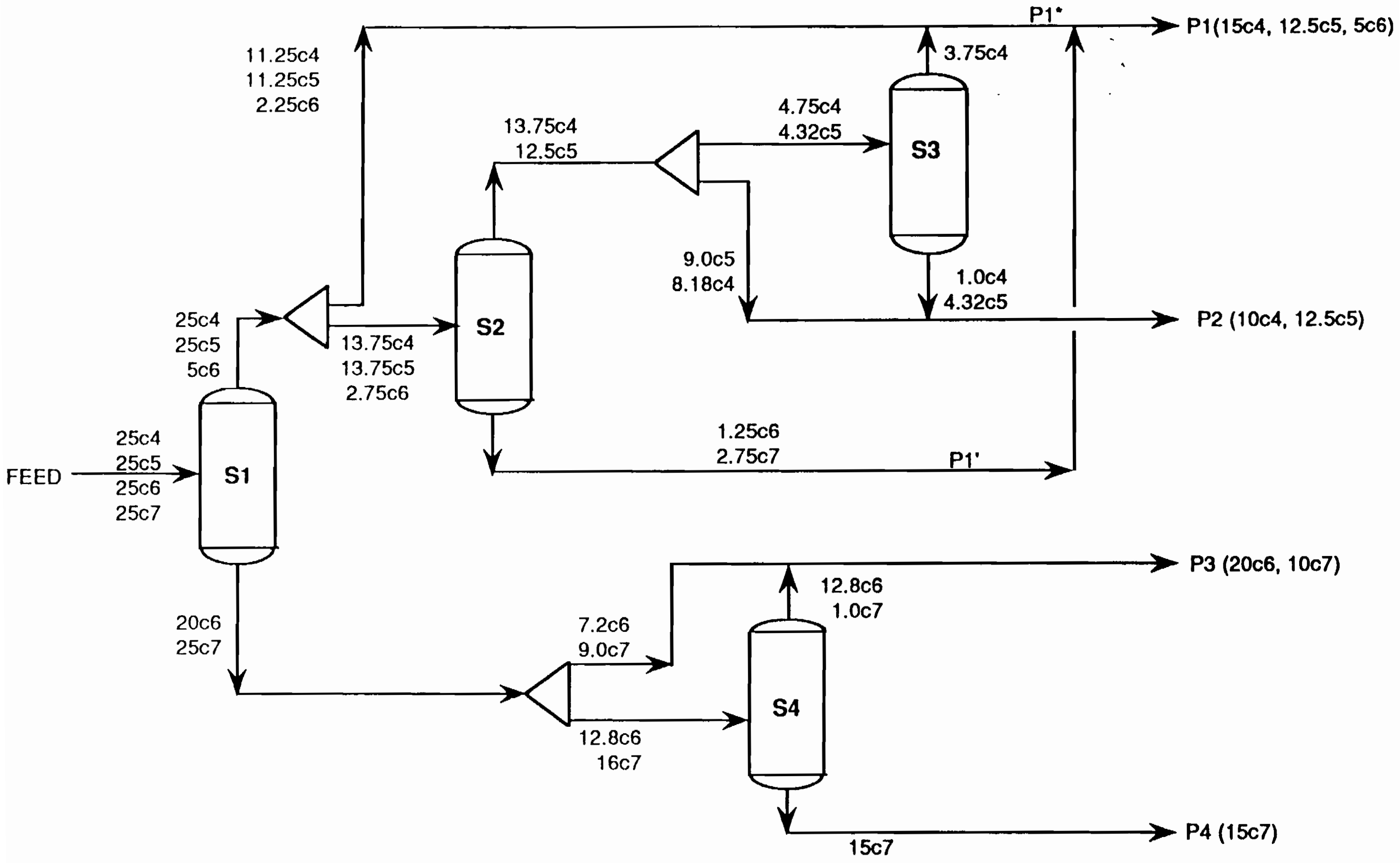


Figure 14.41. Flowsheet of pseudoproduct example.

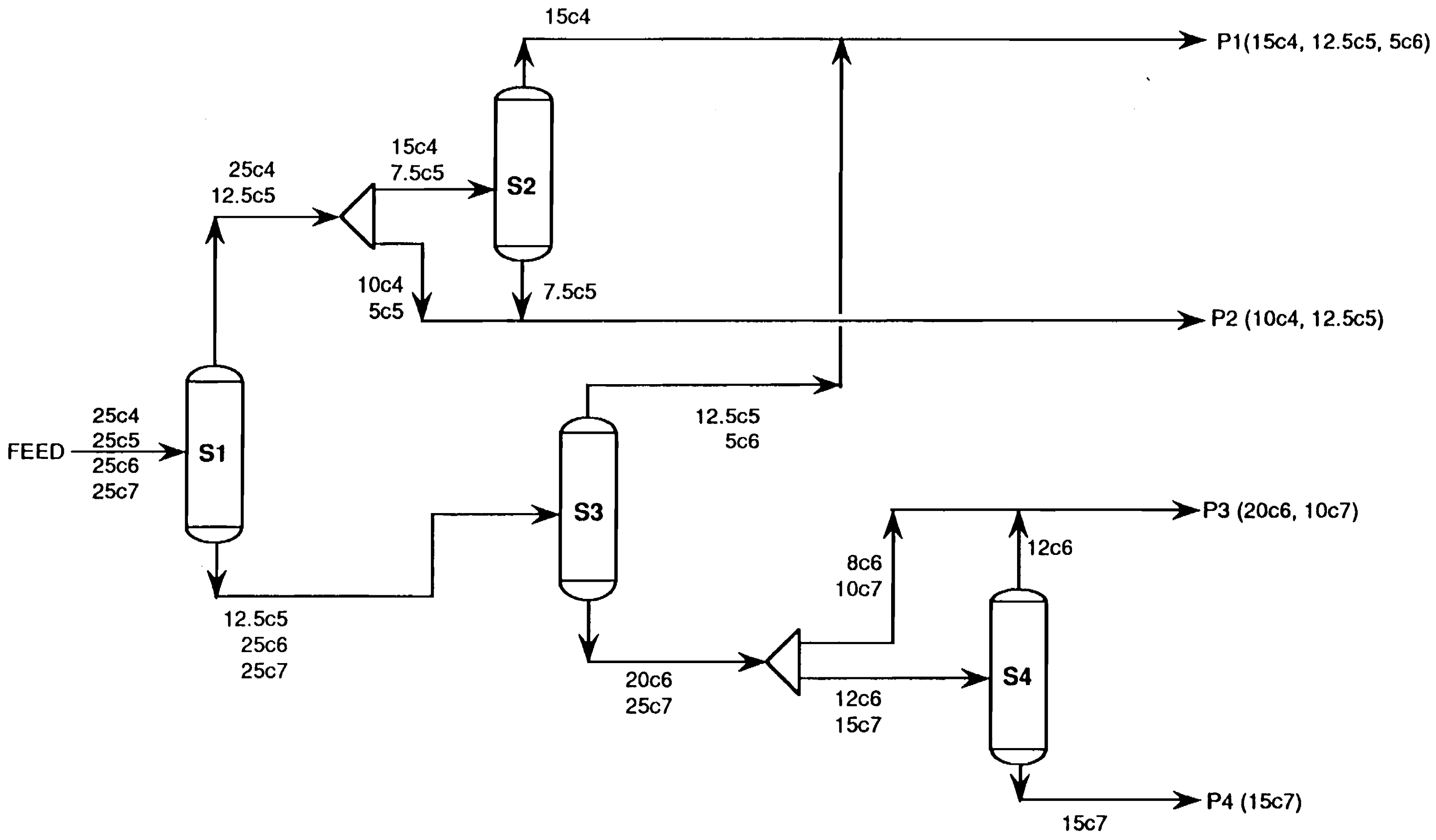


Figure 14.42. Flowsheet of pseudoproduct example.

B. Example 3

Our third example problem addresses a separation problem introduced by Floudas, C.A., and S.H. Anastasiadis, in *Chem. Eng. Sci.*, **43**, 2407 (1988). The system has four components (propane, n-butane, isobutane, and isopentane) and two products, as specified in Table 14.8.

Table 14.8. Feed and product specifications for example problem three.

Desired	Component flow rate (mol/hr)				Product flow
product streams	C3	iC4	nC4	iC5	rate (mol/hr)
P2	5	10	4	10	22.5
P1	10	10	6	5	32.5
Component flow rate (mol/hr)	15	20	10	15	60

To solve this problem using EXSEP, we need the normal boiling point and K-value for each component at feed conditions. The boiling points (°C) are: -42.1, -11.7, -0.5, and 27.8, respectively. The K-values are: 1.3, 0.51, 0.35, 0.14. We need to enter this information into EXSEP. After starting the program and choosing to load data via the **Screen**, we enter the following dialogue:

```
How many products are in the system?
2
How many components are in the system?
4

What is the name of component 1?
c3
What is the name of component 2?
```

```
ic4
What is the name of component 3?
nc4
What is the name of component 4?
ic5

What is the name of product 1?
p1
What is the name of product 2?
p2

What is the flow of c3 in p1?
10
What is the flow of ic4 in p1?
10
What is the flow of nc4 in p1?
6
What is the flow of ic5 in p1?
5

What is the flow of c3 in p2?
5
What is the flow of ic4 in p2?
10
What is the flow of nc4 in p2?
4
What is the flow of ic5 p2?
10

What is the normal boiling point (°C) of c3?
-42.1
What is the K-value of c3?
1.3
What is the normal boiling point (°C) of ic4?
-11.7
What is the K-value of ic4?
0.51
What is the normal boiling point (°C) of nc4?
-0.5
What is the K-value of nc4?
0.36
What is the normal boiling point (°C) of ic5?
27.8
What is the K-value of ic5?
0.14

Are any components hazardous or corrosive?
no
```

Are there any pseudoproducts?
no
Do you wish to save this information to a file?
yes
Please enter file name:
a: ch14ex3.dat

EXSEP saves the file in Prolog clause form as shown in Figure 14.43. Figure 14.44 shows the CAM that EXSEP creates.

```
flow(p1,c3,10).
flow(p2,c3,5).
flow(p1,ic4,10).
flow(p2,ic4,10).
flow(p1,nc4,6).
flow(p2,nc4,4).
flow(p1,ic5,5).
flow(p2,ic5,10).
k_value(c3,1.3).
k_value(ic4,0.51).
k_value(nc4,0.36).
k_value(ic5,0.14).
boiling_temp(c4,-42.1).
boiling_temp(c5,-11.7).
boiling_temp(c6,-0.5).
boiling_temp(c7,27.8).
initial_components([c3,ic4,nc4,ic5]).
initial_set([p1,p2]).
corrosive(none).
```

Figure 14.43. File CH14EX3.DAT after the data input session.

```
        COMPONENT ASSIGNMENT MATRIX

Products    Total Flow   Components
                           c3    ic4    nc4   ic5
  p2            29.0      5.0   10.0    4.0  10.0
  p1            31.0     10.0   10.0    6.0   5.0
```

Figure 14.44. Component assignment matrix for example three.

From the bypass analysis, EXSEP reports that products p1 and p2 are both all-component-inclusive and subjected to bypass. We bypass the maximum amount (100%). EXSEP reports on the screen:

```
Bypass flow to product
p1 is:
Component    Flow Rate
   c3          5.00
  ic4          6.67
  nc4          3.33
  ic5          5.00

Bypass flow to product
p2 is:
Component    Flow Rate
   c3          5.00
  ic4          6.67
  nc4          3.33
  ic5          5.00
```

EXSEP recalculates the CAM (Figure 14.45), and develops the SST (Figure 14.46).

```
     COMPONENT ASSIGNMENT MATRIX

Products    Total Flow  Components
                          c3    ic4    nc4   ic5
  p2            9.0      0.0    3.3    0.7   5.0
  p1           11.0      5.0    3.3    2.7   0.0
```

Figure 14.45. Component assignment matrix for example three after bypass.

```
Split: [c3]/[ic4,nc4,ic5]    LK/HK: c3/ic4    Status: feasible
(d/b)c3    0.98/0.02
(d/b)ic4   0.02/0.98
(d/b)nc4   0.02/0.98
(d/b)ic5   0.02/0.98

LK/HK Temperature Difference (°C): 30.40   CES: 3.00   Alpha(LK,HK): 2.55
---------------------------------------------------------------------------
Split: [c3,ic4]/[nc4,ic5]    LK/HK: ic4/nc4    Status: feasible
(d/b)c3    0.98/0.02
(d/b)ic4   0.98/0.02
(d/b)nc4   0.02/0.98
(d/b)ic5   0.02/0.98

LK/HK Temperature Difference (°C): 11.20   CES: 2.37   Alpha(LK,HK): 1.42
---------------------------------------------------------------------------
Split: [c3,ic4,nc4]/[ic5]    LK/HK: nc4/ic5    Status: feasible
(d/b)c3    0.98/0.02
(d/b)ic4   0.98/0.02
(d/b)nc4   0.98/0.02
(d/b)ic5   0.02/0.98

LK/HK Temperature Difference (°C): 28.30   CES: 2.79 Alpha(LK,HK): 2.57
---------------------------------------------------------------------------
Split: [p1]/[p2]             LK/HK: any/any    Status: infeasible
(d/b)c3   0.98/0.02
(d/b)ic4   0.50/0.50
(d/b)nc4   0.80/0.20
(d/b)ic5   0.02/0.98

LK/HK Temperature Difference (°C):  0.00   CES: 0.00   Alpha(LK,HK): 0.00
```

Figure 14.46. Output from the separation specification table for example three.

After reporting the SST, EXSEP does the heuristic analysis. As in previous cases, neither heuristic S1 (remove corrosive or hazardous components first) nor S2 (perform difficult separations last) apply. Since, product p1 is the most plentiful product, however, heuristic C1 does apply. EXSEP indicates splits that violate C1, and recommends removing them from consideration. We see on the screen:

```
Most plentiful product is p1. The sharp
split [c3]/[ic4,nc4,ic5] with c3/ic4 as LK/HK
violates heuristic C1: remove the most plentiful
product first. Do you wish to remove this split
from consideration or maintain it?
```

```
Most plentiful product is p1. The sharp
split [c3, ic4]/[nc4,ic5] with ic4/nc4 as LK/HK
violates heuristic C1: remove the most plentiful
product first. Do you wish to remove this split
from consideration or maintain it?
```

In response, we recommend removal of both splits by highlighting **Remove** and hitting **ENTER**. After EXSEP removes these splits from the database, only one feasible split remains. EXSEP recommends this split (after applying heuristic C2):

```
Split [c3,ic4,nc4]/[ic5] with:
    -- components nc4/ic5 as LK/HK
    -- a CES = 2.79
    -- a molar split ratio = 75/25
is the recommended split. Do you
wish to make this split?
```

Since we do want this split, we highlight **Split** and hit **ENTER**. EXSEP executes the split, and separator S1 becomes:

```
                5.00 c3
                6.67 ic4
                3.33 nc4
5.00 c3    ↗
6.67 ic4
3.33 nc4
5.00 ic5   ↘
                5.00 ic5
```

(Bypass: 5.0 c3 + 6.67 ic4 + 3.33 nc4 + 5.00 ic5 to p1; 5.0 c3 + 6.67 ic4 + 3.33 nc4 + 5.00 ic5 to p2)

EXSEP immediately begins to analyze the overhead from S1. It builds and displays the CAM of Figure 14.47, and then performs bypass analysis. Product p1 is all-component-inclusive and subjected to bypass. We bypass the maximum amount (100%), and EXSEP reports:

COMPONENT ASSIGNMENT MATRIX

Products	Total Flow	Components		
		c3	ic4	nc4
p2	4.0	0.0	3.33	0.67
p1	10.0	5.0	3.33	1.67

Figure 14.47. Component assignment matrix for example three overhead from S1.

Bypass flow to product p1 is:

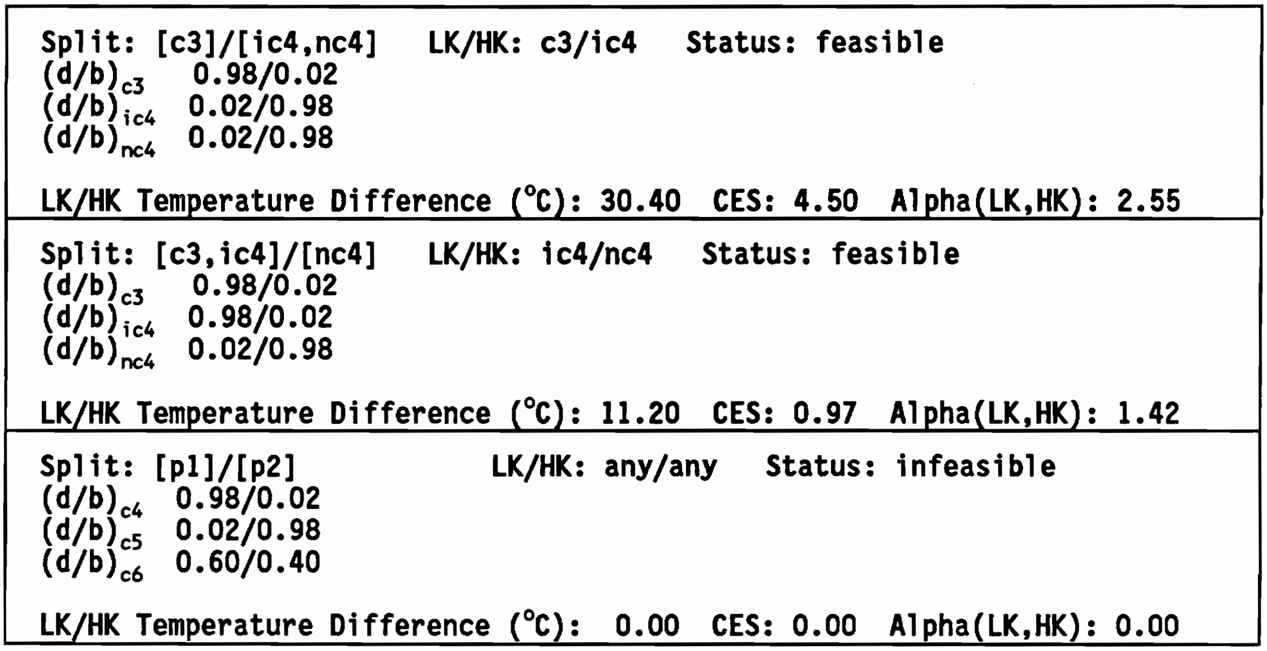
Component	Flow Rate
c3	2.50
ic4	3.33
nc4	1.67

COMPONENT ASSIGNMENT MATRIX

Products	Total Flow	Components		
		c3	ic4	nc4
p2	4.0	0.0	3.33	0.67
p1	3.5	2.5	0.00	1.00

Figure 14.48. Component assignment matrix for example three, S1 overhead after bypass.

EXSEP recalculates and displays the new CAM, shown in Figure 14.48 EXSEP then calculates the SST, shown in Figure 14.49. According to the SST, the sloppy split p1/p2 is again infeasible, while sharp splits c3/ic4nc4 and c3ic4/nc4 are feasible.



```
Split: [c3]/[ic4,nc4]   LK/HK: c3/ic4   Status: feasible
(d/b)_c3    0.98/0.02
(d/b)_ic4   0.02/0.98
(d/b)_nc4   0.02/0.98

LK/HK Temperature Difference (°C): 30.40  CES: 4.50  Alpha(LK,HK): 2.55
```

```
Split: [c3,ic4]/[nc4]   LK/HK: ic4/nc4   Status: feasible
(d/b)_c3    0.98/0.02
(d/b)_ic4   0.98/0.02
(d/b)_nc4   0.02/0.98

LK/HK Temperature Difference (°C): 11.20  CES: 0.97  Alpha(LK,HK): 1.42
```

```
Split: [p1]/[p2]            LK/HK: any/any   Status: infeasible
(d/b)_c4   0.98/0.02
(d/b)_c5   0.02/0.98
(d/b)_c6   0.60/0.40

LK/HK Temperature Difference (°C):  0.00  CES: 0.00  Alpha(LK,HK): 0.00
```

Figure 14.49. Separation specification table for example three feed into separator S2.

EXSEP then performs heuristic analysis. Heuristics S1, S2, and C1 do not apply. Based on heuristic C2, EXSEP reports the following:

```
Split [c3]/[ic4,nc4] with:
    -- components c3/ic4 as LK/HK
    -- a CES = 4.50
    -- a molar split ratio = 67/33
is the recommended split. Do you
wish to make this split?
```

We agree to make the split, highlight **Split** and hit **ENTER**. EXSEP executes the split, and separator S2 is:

```
             2.5 c3
2.5  c3   ↗
3.33 ic4            (Bypass: 2.5 c3 + 3.33 ic4 + 1.67 nc4 to p1)
1.67 nc4  ↘
             3.33 ic4
             1.67 nc4
```

EXSEP then analyzes the S2 overhead stream. The stream flow is 2.5 mol/hr c3. Since the stream is pure, overhead processing is complete. Therefore, EXSEP proceeds to the bottoms from S2. It calculates and reports the CAM of Figure 14.50.

COMPONENT ASSIGNMENT MATRIX

Products	Total Flow	Components	
		ic4	nc4
p2	4.0	3.33	0.67
p1	1.0	0.00	1.00

Figure 14.50. Component assignment matrix for bottoms from S2.

The CAM reveals that product p2 is all-component-inclusive and subjected to bypass. We bypass 100% (the maximum amount), and EXSEP reports:

Bypass flow to product p2 is:

Component	Flow Rate
ic4	1.33
nc4	0.67

EXSEP recalculates the CAM, as shown in Figure 14.51. The horizontal and the vertical splits represent the same sharp split, ic4/nc4. We execute the ic4/nc4 split, and separator S3 becomes:

COMPONENT ASSIGNMENT MATRIX

Products	Total Flow	Components	
		ic4	nc4
p2	2.0	2.00	0.00
p1	1.0	0.00	1.00

Figure 14.51. Component assignment matrix for bottoms from S2 after bypass.

```
            2.0 ic4
2.0 ic4 ↗             (Bypass: 1.33 ic4 + 0.67 nc4 to p2)
1.0 nc4 ↘
            1.0 nc4
```

The sequence is complete. EXSEP summarizes the flowsheet, shown in Table 14.9. Figure 14.52 is the graphical representation of the same flowsheet.

Table 14.9. Flowsheet of sequence for example three.

Separator	Bypass around	Stream flow in	Overhead	Bottoms
S1	pl: 5.00 c3 6.67 ic4 3.33 nc4 5.00 ic5 p2: 5.00 c3 6.67 ic4 3.33 nc4 5.00 ic5	5.00 c3 6.67 ic4 3.33 nc4 5.00 ic5	5.00 c3 6.67 ic4 3.33 nc4	5.00 ic5
S2	pl: 2.50 c3 3.33 nc4 1.67 ic4	2.50 c3 3.33 nc4 1.67 ic4	2.5 c3	3.33 nc4 1.67 ic4
S3	p2: 1.33 ic4 0.67 nc4	2.00 ic4 1.00 nc4	2.0 ic4	1.0 nc4

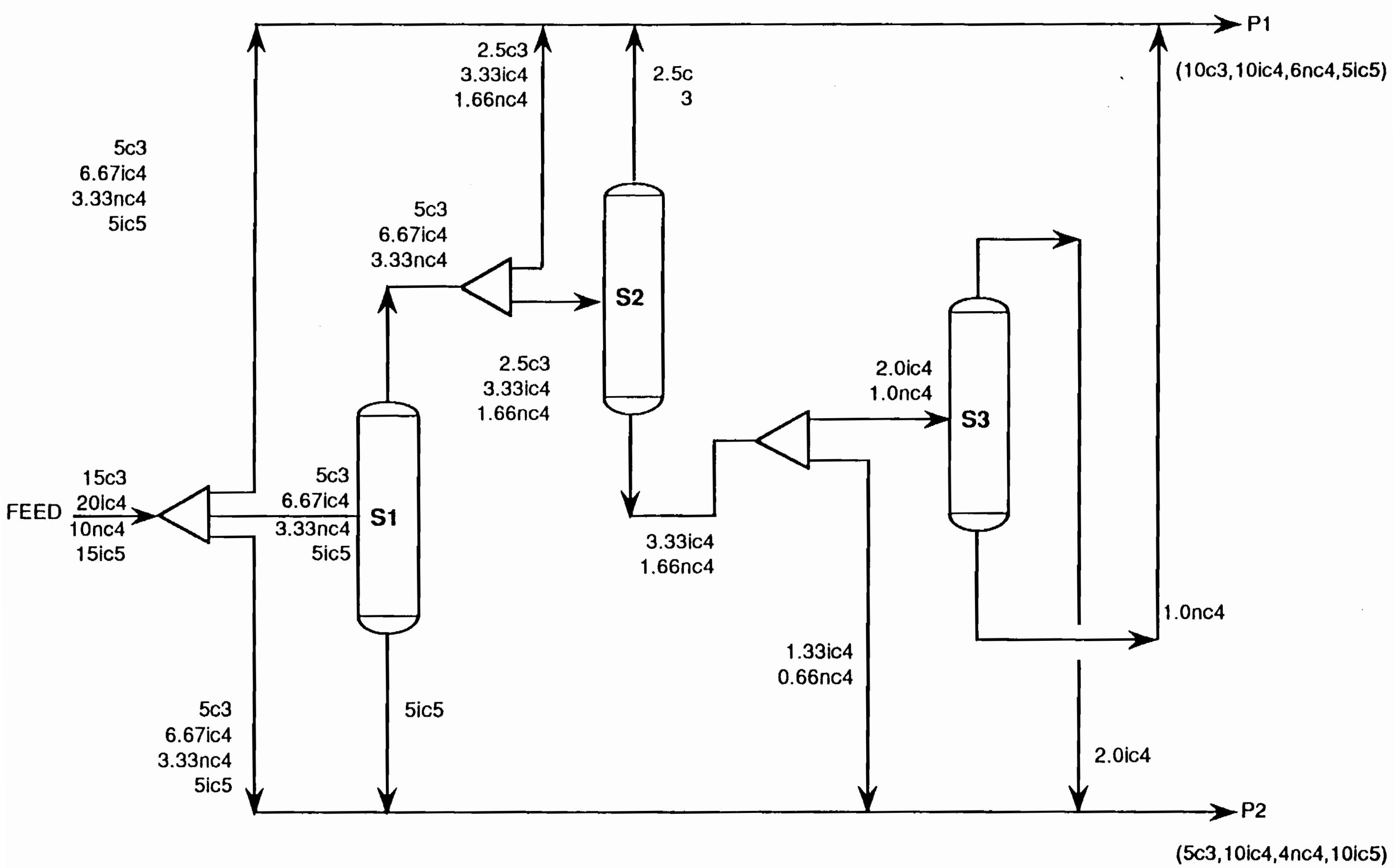


Figure 14.52. Flowsheet for example three.

C. Practice problems

14.4.1 Using EXSEP, develop the alternate flowsheets for example one shown in Figure 14.28, 29, and 30.

14.4.2 Using EXSEP, develop the alternate flowsheets for the pseudoproduct transformation of example one, shown in Figures 14.41 and 42.

14.4.3 In example three, the original component assignment matrix, shown in Figure 14.44, is:

Products	Total Flow	Components			
		c3	ic4	nc4	ic5
p2	29.0	5.0	10.0	4.0	10.0
pl	31.0	10.0	10.0	6.0	5.0

After bypassing the maximum amount, we get the CAM of Figure 14.45:

Products	Total Flow	Components			
		c3	ic4	nc4	ic5
p2	9.0	0.0	3.33	0.67	5.0
pl	11.0	5.0	3.33	2.67	0.0

The horizontal split pl/p2 is infeasible. We now do a CAM transformation to give:

Products	Total Flow	Components			
		c3	ic4	nc4	ic5
p2*	5.67	0.0	0.0	0.67	5.0
pl*	2.67	0.0	0.0	2.67	0.0
p2′	3.33	0.0	3.33	0.0	0.0
pl′	8.33	5.0	3.33	0.0	0.0

Using EXSEP, develop several separation sequences for this transformation.

14.4.4 Consider the products from a thermal cracking unit (Liu et. al., 1987, pp.154-157). The feed mixture consists of:

Species	Flow Rate (mol/hr)	Normal Boiling Point (°C)	K-value
a: Hydrogen	18	-253	23.2
b: Methane	5	-161	4.60
c: Ethylene	24	-104	1.10
d: Ethane	15	-88	0.70
e: Propylene	14	-48	0.214
f: Propane	6	-42	0.174
g: Heavies	8	-1	0.061

We desire six products, five of which are pure: ab, c, d, e, f, g. Use EXSEP to develop two good all-sharp sequences for this separation problem.

14.5 CHAPTER SUMMARY

This chapter introduced EXSEP from the user's perspective. EXSEP, with its menu-driven decision tools, is very easy to use. It generates multicomponent separation flowsheets that are both technically feasible, and economically attractive. Because EXSEP is written in compiled Prolog, it runs very quickly.

We can input data into EXSEP in two ways:

- direct data entry via the screen and keyboard.
- rapid data entry via consulting a file.

One of EXSEP's major advantages is its flexibility. We can choose to override EXSEP's recommendations and implement splits manually to generate a number of competing, economically attractive flowsheets.

A LOOK AHEAD

We have discussed EXSEP from a chemical engineering perspective, and explained in detail how to use the program. In the next chapter, we discuss EXSEP from an AI perspective, examining both its organization and its structure. We also outline the Prolog programming techniques used in the program.

REFERENCES

Cheng, S.H, and Y.A. Liu, "Studies in Chemical Process Design and Synthesis. 8. A Simple Heuristic Method for the Synthesis of Initial Sequences for Sloppy Multicomponent Separations," *Ind. Eng. Chem. Res.*, **27**, 2304 (1988).

Floudas, C.A., "Separation Synthesis of Multicomponent Feed Streams into Multicomponent Product Streams," *AIChE J.*, **33**, 540 (1987).

Floudas, C.A., and S.H. Anastasiadis, "Synthesis of Distillation Sequences with Several Multicomponent Feed and Product Streams," *Chem. Eng. Sci.*, **43**, 2407 (1988).

Liu, Y.A., H.A. McGee, Jr., and W.R. Epperly, Editors, *Recent Developments in Chemical Process and Plant Design*, John Wiley and Sons, New York, NY (1987).

15

AI PERSPECTIVE OF EXSEP

This chapter discusses EXSEP from an AI perspective. It outlines the knowledge representation, the search strategy, the actual structure of EXSEP, and specific Prolog programming techniques.

15.1 KNOWLEDGE REPRESENTATION

A. Logic Structure

1. Predicate Logic

EXSEP is a rule-based expert system written in Prolog. Almost all expert systems written in science or engineering are rule-based, frame-based, or object-oriented. Although frame-based and object-oriented expert systems can be done in Prolog, the language is more suited for rule-based systems using predicate logic.

We have discussed logic and Prolog in *Part One* of this text, but never formally introduced predicate logic. Predicate logic uses logical statements, or predicates, to represent knowledge. It facilitates reasoning based on the knowledge, and provides a way of deducing new conclusions based on old ones. However, formal predicate logic does not possess a decision procedure. Essentially, if we know how to program in Prolog, we know how to use predicate logic; "Prolog" actually stands for *PRO*gramming in *LOG*ic. Prolog models predicate logic very closely. If we want to say "component A is corrosive" in predicate logic, we may represent this knowledge as:

```
Corrosive(A)
```

In Prolog, we write the statement as a fact:

```
corrosive(a).
```

If we want to say, "for all X and for all Y, X greater than Y implies that X is the maximum between X and Y," we write in predicate logic:

```
∀X ∀Y (X > Y) → maximum(X,Y,X)
```

In Prolog, we write:

```
maximum(X,Y,X):- X > Y.
```

Prolog, then, is a backward-chaining replication of predicate logic. As discussed in *Part One*, the :- in Prolog is interpreted as an IF. The symbol was chosen because when Prolog was originally developed, :- was the closest thing to a backward arrow, ←. Some versions of Prolog now use the backward-arrow syntax

```
maximum(X,Y,X) ← X > Y.
```

to further emphasize that Prolog is logically based.

All of the knowledge in EXSEP, including all facts, rules, and heuristics, is represented by predicate logic. Remember that every statement written in Prolog is in clause form. A fact is a clause *without* a body, and a rule is a clause *with* a body.

2. Horn Clauses

Prolog uses a special type of clause know as a *Horn clause*. Only Horn clauses are legal in Prolog; no other type of clause is allowed. A Horn clause has at most *one positive literal*; that is one and only one positive conclusion can be drawn from the clause. Multiple positive conclusions are impossible. If the Horn clause is unsuccessful, we say the clause "fails." We conclude nothing at all from a failed clause; we *cannot* conclude that the statement is false.

The following statement contains *two* predicates on the left:

```
a,b :- c.
```

The comma between **a** and **b** represent an AND. In logic, the above statement says "if c is true, a and b are true." Since the statement has two positive conclusions, it is not a Horn clause and is illegal in Prolog.

The next statement also contains two predicates on the left side. But this time, a semicolon (;) separates them, and in Prolog the ;

represents an OR:

```
a;b :- c.
```

This statement says "if c is true, either a or b is true." Despite the OR, the statement has two possible positive conclusions and is not a Horn clause. The statement, therefore, is illegal in Prolog.

Finally, consider the following illegal statement, containing a **not**,

```
not(a) :- b.
```

which says, "a is not true if b is true." Recall that a Horn clause must have at most one *positive* literal. This statement, however, has only a *negative* conclusion: a is not true. The statement, therefore, is not a Horn clause and not legal in Prolog.

3. Examples of EXSEP Clauses

Now let us consider some actual Prolog statements written for EXSEP. The EXSEP statement:

```
min(X,Y,X):- X < Y.
```

is a Horn clause. It has at most one positive literal: if X < Y, we conclude that **min(X,Y,X)** is true; if X > Y or X = Y, we conclude nothing and the statement fails.

Consider the following clause to calculate the minimum number of stages for a distillation column using the Fenske equation (section 12.3). As a reminder, the Fenske equation is:

$$N_{\min} = \frac{\log\ [\ (\frac{d}{b})_{LK} \bullet (\frac{b}{d})_{HK}\ ]}{\log\ \alpha_{LK,HK}} \tag{15.1}$$

where:

- N_{min} is the minimum number of theoretical stages.
- LK and HK are the light-key and heavy-key components, respectively.
- $(d/b)_{LK}$ is ratio of the recovery fraction in the distillate to the recovery fraction in the bottoms of the light-key component.
- $(b/d)_{HK}$ is ratio of the recovery fraction in the bottoms to the recovery fraction in the distillate of the heavy-key component.
- $\alpha_{LK,HK}$ is the relative volatility between the light key and the heavy key.

The EXSEP clause calculates the minimum number of stages, N_{min}, for a given split. To give the clause a meaningful name, we call it **calc_min_stages**. Its key variables are:

- **Top**: the list of overhead components, e.g., [c3, ic4, nc4].

- **Bot**: the list of bottoms components, e.g., [c5, ic5, c6].
- **LK** and **HK**: the light-key and heavy-key components, respectively.
- **Ratio1** and **Ratio2**: $(d/b)_{LK}$ and $(b/d)_{HK}$, respectively.
- **Alpha**: the relative volatility, $\alpha_{LK,HK}$.
- **TFlow1** and **TFlow2**: distillate molar flow rates (mol/hr) for the LK and HK, respectively.
- **BFlow1** and **BFlow2**: bottoms molar flow rates (mol/hr) for the LK and HK, respectively.

The clause itself is:

```
calc_min_stages(Top,Bot,LK,HK,Nmin,Ratio1,Ratio2):-   %1
     component_flow(Top,LK,TFlow1),                   %2  $D_{LK}$
     TFlow1 > 0.0,                                    %3
     component_flow(Bot,LK,BFlow1),                   %4  $B_{LK}$
     component_flow(Top,HK,TFlow2),                   %5  $D_{HK}$
     component_flow(Bot,HK,BFlow2),                   %6  $B_{HK}$
     BFlow2 > 0.0,                                    %7
     k_value(LK,KLK),                                 %8
     k_value(HK,KHK),                                 %9
     Alpha is KLK/KHK,                                %10 $\alpha_{LK,HK}$
     solve_ratio(TFlow1,BFlow1,Ratio1),               %11 $(d/b)_{LK}$
     solve_ratio(BFlow2,TFlow2,Ratio2),               %12 $(b/d)_{HK}$
     Nmin is log(Ratio1*Ratio2)/log(Alpha),!.         %13 $N_{min}$
```

The information flow in the clause is:

calc_min_stages(*input*,*input*,*input*,*input*,*output*,*output*,*output*)

This clause inputs:

- the list of **Top** components,
- the list of **Bottom** components, and
- the **LK** and **HK** components.

The clause outputs:

- **Nmin**, the minimum number of stages, and
- **Ratio1** and **Ratio2**, which are $(d/b)_{LK}$ and $(b/d)_{HK}$, respectively.

To calculate **Nmin**, some intermediate information is needed. The **component_flow** clause (in lines 2, 4, 5 and 6) obtains the flow rates for the light-key and heavy-key components in the process streams designated by **Top** and **Bot**. For example, in our original CAM of example one of the last chapter (Figure 14.3), if we were calculating **Nmin** for the p12/p34 split with c5/c6 as LK/HK, then **Top** is the list **[c4, c5, c6]** and **Bot** is the list **[c6, c7]**. The statement:

```
component_flow(Top,HK,TFlow2)
```

has the following instantiations when the clause is entered:

```
component_flow([ c4, c5, c6 ], c6, TFlow2)
```

The clause then calculates **TFlow2**, the flow rate of **c6** in the overhead stream composed of components **[c4, c5, c6]**. Based on the CAM in Figure 14.3, we see that **TFlow2** is equal to 20.0, and the **component_flow** clause instantiates **TFlow2** to this value upon successful completion.

We also need the information regarding the K-values of the LK and HK. The **k_value** clause gives us this information by matching with facts in the database. We then use the built-in **is** predicate to calculate **Alpha**.

Finally, before we can explicitly calculate **Nmin**, we need to calculate $(d/b)_{LK}$ and $(b/d)_{HK}$, done by the **solve_ratio** relation. In line 11, the statement

```
solve_ratio(TFlow1,BFlow1,Ratio1)
```

instantiates **Ratio1** to the ratio **TFlow1/BFlow1**. At first glance, **solve_ratio** appears unnecessary. Why not just use the **is** predicate, write

```
Ratio1 is TFlow1/BFlow1
```

and be done with it? There are two reasons. First, if we have **BFlow1** = 0, an error results. The **solve_ratio** clause tests for this condition and responds accordingly, preventing a divide-by-zero error. Secondly, if we have a sharp split (d/b is 0/1 or 1/0), then **solve_ratio** outputs the *limiting ratio*, either 0.02/0/98 or 0.98/0.02, discussed in chapter 13. Knowing the K-values and recovery ratios for the LK and HK, the **calc_min_stages** clause is ready to determine **Nmin** from the Fenske equation, as shown in line 13 of the clause.

The **calc_min_stages** is a Horn clause, because it has at most one positive conclusion. If the clause succeeds, it instantiates N_{min}, $(d/b)_{LK}$ and $(b/d)_{HK}$ to the correct values. Note that although three variables are instantiated, logically the variables are grouped into just one positive conclusion. If the clause fails (i.e., TFlowl = 0, or BFLow2 = 0), it simply has no conclusion.

Predicate logic provides an effective means of implementing rule-based expert systems. However, such systems, compared to frame-based systems, frequently require significant understanding and detailed characterization of the knowledge to work correctly. They can be too "fine-tuned," and as a result unable to grasp overall concepts easily. But when they are working correctly, they are swift and efficient.

B. Problem Decomposition

1. Synthesis versus Decomposition

Challenges arise when developing expert systems for design engineering (in this case, process synthesis and flowsheet development). Expert systems are better at solving problems via *decomposition* (tearing down problems) rather than *synthesis* (building up answers), but flowsheet development is inherently a synthesis task. Synthesis is challenging for expert systems primarily because it provides nothing concrete to analyze until the job is done. Using the synthesis approach to create flowsheets, we develop small, independent unit operations that need to be configured together for the final solution. Frequently, the units may not "fit together" exactly, and the expert system needs to resolve the conflicts. This resolution can be very difficult.

Let us investigate this aspect in more depth. One problem in synthesizing a flowsheet using an expert system is maintaining *continuity*. When independent units are "pasted together" to form the process, the final design must conserve material and energy. As shown in Figure 15.1a-b, if the flow rates, composition, and temperature coming off separator one do not match the anticipated feed conditions for separator two, a conflict develops. Some of these conflicts can be resolved fairly easily. For instance, if the temperatures do not match, we place a heat exchanger in the system. Resolving other conflicts, however, is more difficult. If the flow rates and compositions do not match, we must backtrack and change the design of a separator.

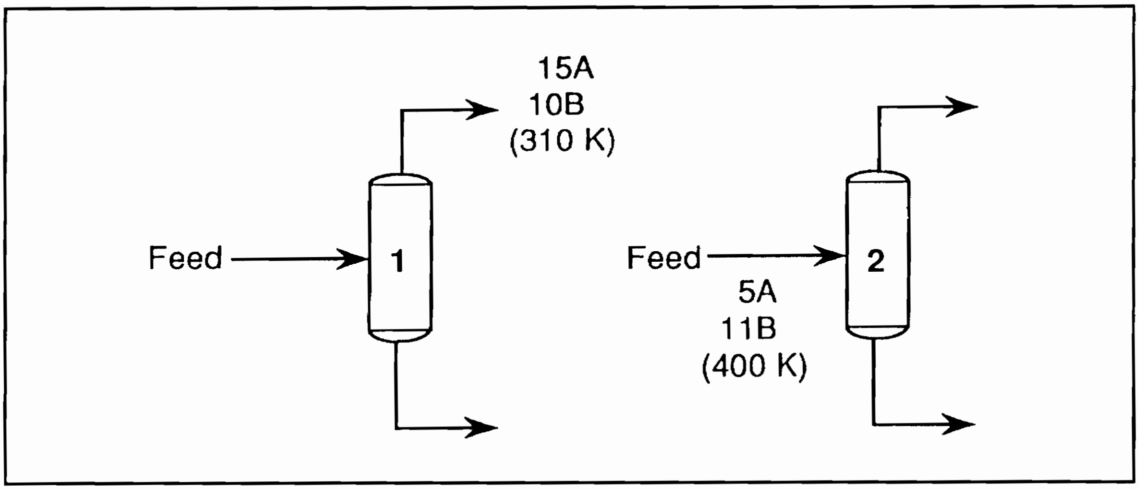


Figure 15.1a. The flowsheet-development problem with a synthesis approach: operations 1 and 2 are separated; no conflict.

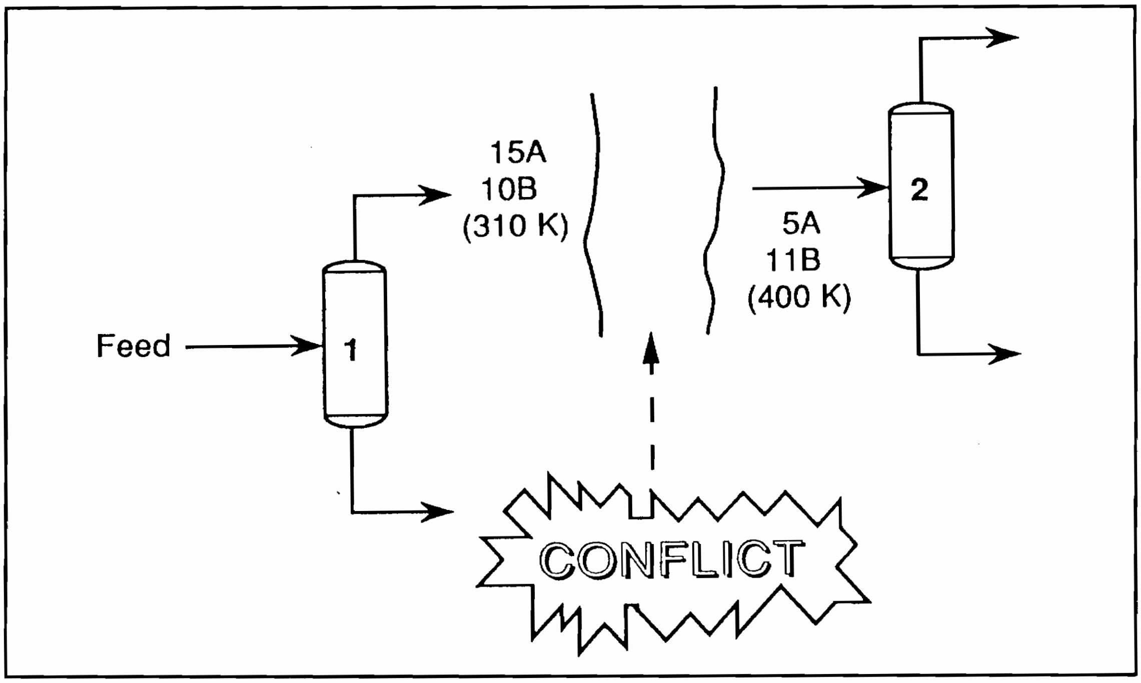


Figure 15.1b. The flowsheet-development problem with a synthesis approach: operation 1 feeds into operation 2; conflict.

Backtracking and changing designs then open up several complications. Is the new design feasible? Can it give us the desired flow rates and compositions? How will the new design affect other units? Will we have to change their design too? All of these questions need to be addressed before we reach a final solution.

It is easier to solve engineering-design problems using the problem-decomposition approach. There already is a concrete, existing framework for the system to analyze. It is easier to implement constraints and maintain continuity. Minimal conflict resolution is required. Many of the problem associated with the synthesis approach do not need to be tackled. The moral of the story is:

> *Expert systems work more efficiently when they implement their inference chain in a problem-decomposition mode rather than a problem-synthesis mode.*

The challenge in engineering-design expert systems is developing the knowledge representation that facilitates problem-decomposition, realizing all the while that design is synthesis task.

2. Decomposition in EXSEP

The knowledge representation in EXSEP works towards developing a flowsheet via systematic problem decomposition. The primary decomposition tool is the *Component Assignment Matrix* (CAM).

Remember that the CAM is a P x C matrix, where P is the number of products and C is the number of components. We write:

$$\text{CAM} = | f_{ij} | \qquad (15.2)$$

where f_{ij} is the flow rate of the j-th component in the i-th product. In a system with four products and four components (example one in chapter 14), our CAM is a 4 x 4 matrix:

$$CAM = \begin{vmatrix} f_{41} & f_{42} & f_{43} & f_{44} \\ f_{31} & f_{32} & f_{33} & f_{34} \\ f_{21} & f_{22} & f_{23} & f_{24} \\ f_{11} & f_{12} & f_{13} & f_{14} \end{vmatrix} \qquad (15.3)$$

To develop a flowsheet via problem decomposition, EXSEP analyzes and split the CAM to produce two smaller sub-matrices. EXSEP then splits these smaller matrices, and this process continues until the split produces either (1) a row matrix, or (2) a column matrix. At that point, the matrix requires no further splitting, because we have isolated either a specific product (row matrix) or a pure component (column matrix).

Let us demonstrate this principle by splitting a 4-component, 4-product (equation 15.2). If EXSEP recommends splitting product one (i.e., $|f_{1j}|$) from the matrix, we get:

$$\begin{vmatrix} f_{41} & f_{42} & f_{43} & f_{44} \\ f_{31} & f_{32} & f_{33} & f_{34} \\ f_{21} & f_{22} & f_{23} & f_{24} \\ f_{11} & f_{12} & f_{13} & f_{14} \end{vmatrix} \rightarrow \begin{vmatrix} f_{11} & f_{12} & f_{13} & f_{14} \end{vmatrix} + \begin{vmatrix} f_{41} & f_{42} & f_{43} & f_{44} \\ f_{31} & f_{32} & f_{33} & f_{34} \\ f_{21} & f_{22} & f_{23} & f_{24} \end{vmatrix} \qquad (15.4)$$

The row matrix $|f_{1j}|$ is the overhead, and meets the specifications for product one. Further splitting of this row matrix is unnecessary.

The 3 x 4 bottoms matrix, however, is neither a row nor a column matrix, and it requires further separation. Suppose EXSEP recommends that we split off component one (the column $|f_{i1}|$). This split is:

$$\begin{vmatrix} f_{41} & f_{42} & f_{43} & f_{44} \\ f_{31} & f_{32} & f_{33} & f_{34} \\ f_{21} & f_{22} & f_{23} & f_{24} \end{vmatrix} \rightarrow \begin{vmatrix} f_{41} \\ f_{31} \\ f_{21} \end{vmatrix} + \begin{vmatrix} f_{42} & f_{43} & f_{44} \\ f_{32} & f_{33} & f_{34} \\ f_{22} & f_{23} & f_{24} \end{vmatrix} \qquad (15.5)$$

The column matrix, $|f_{i1}|$ is a stream of pure component one, and requires no further separation. The bottoms, however, is a 3 x 3 matrix, and *does* require further processing. If EXSEP recommends splitting off product four, we get:

$$\begin{vmatrix} f_{42} & f_{43} & f_{44} \\ f_{32} & f_{33} & f_{34} \\ f_{22} & f_{23} & f_{24} \end{vmatrix} \rightarrow \begin{vmatrix} f_{32} & f_{33} & f_{34} \\ f_{22} & f_{23} & f_{24} \end{vmatrix} + \begin{vmatrix} f_{42} & f_{43} & f_{44} \end{vmatrix} \qquad (15.6)$$

The row matrix, $|f_{42}\ f_{43}\ f_{44}|$, is the bottoms, and requires no further processing. Although it does not encompass *all* of product four, it does contain components two, three, and four at the required flow rates (its fraction of component one, f_{41}, was split off previously and needs to be blended in to fulfill product specifications for product four).

The overhead stream still requires separation. If EXSEP recommends a split between products two and three, we get:

$$\begin{vmatrix} f_{32} & f_{33} & f_{34} \\ f_{22} & f_{23} & f_{24} \end{vmatrix} \rightarrow \begin{vmatrix} f_{22} & f_{23} & f_{24} \end{vmatrix} + \begin{vmatrix} f_{32} & f_{33} & f_{34} \end{vmatrix} \tag{15.7}$$

This split results in two row matrices, $|f_{22}\ f_{23}\ f_{24}|$ as overhead, and $|f_{32}\ f_{33}\ f_{34}|$ as bottoms. Neither requires further processing. The problem decomposition is complete; the CAM has been repetitively split until products or components are isolated. Figure 15.2 shows the entire separation sequence. Items marked with an asterisk, *, are either row or column matrices requiring no further processing.

To summarize, form an AI standpoint, the CAM is a tool to obtain a flowsheet solution via *problem-decomposition*. The CAM provides potential split options that EXSEP evaluates based on certain rules. We repeatedly decompose the CAM until the entire sequence is complete, and only row and column matrices remain. By using a problem-decomposition knowledge representation, we end up with a very fast and accurate reasoning mechanism.

C. Constraints

All engineering problems have constraints. The methods of handling those constraints in an expert system can critically affect the system's

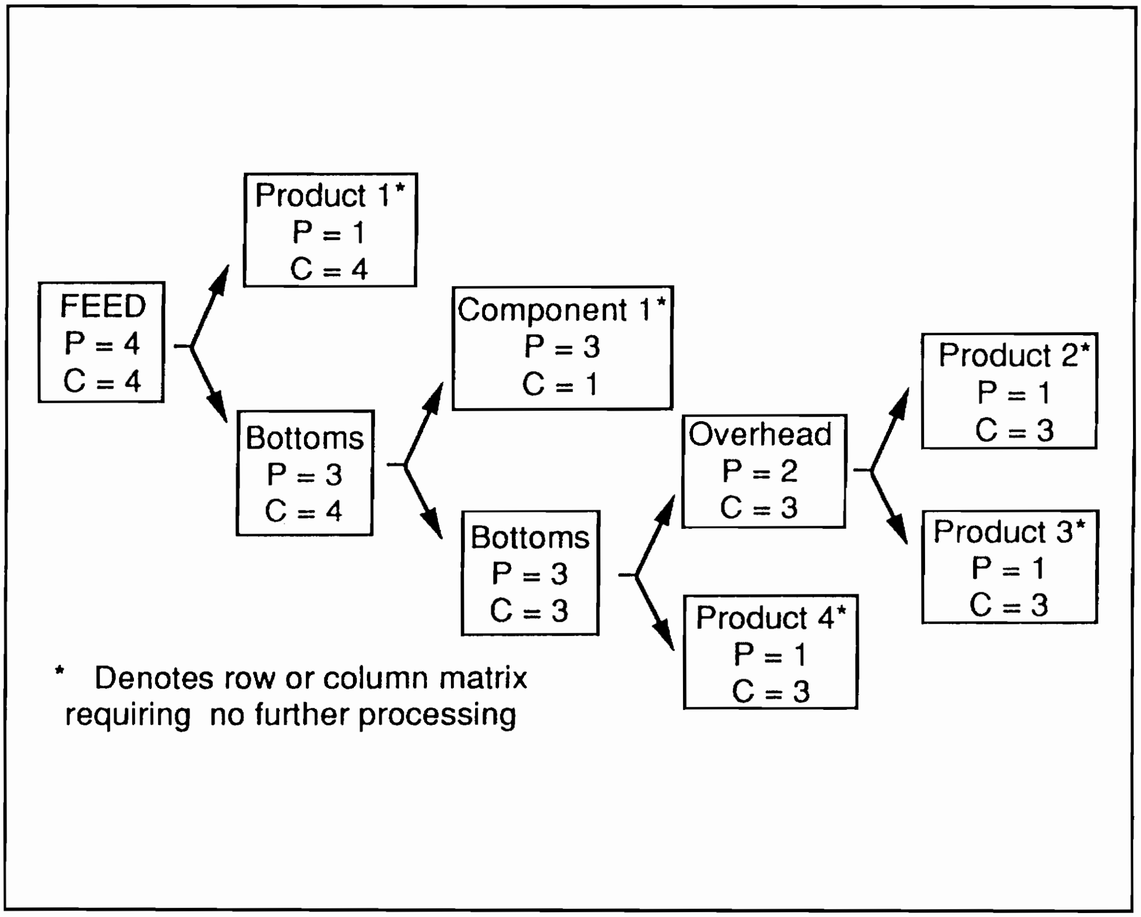


Figure 15.2. Separation sequence a for four-product, four-component separation problem.

accuracy and performance. In expert systems, and in flowsheet development, constraints can be handled *globally* or *locally*.

1. Handling Constraints Locally

Let us consider the material-balance constraint: what goes in, must come out. We may choose to handle this constraint locally. In that case, we first develop a set of separators, each with its own material balance.

We specify the light-key and heavy-key recovery ratios and then determine nonkey-component distributions for each separator. To configure the entire flowsheet, we need to maintain continuity of flow rate and concentration. Nonkey components may distribute in a way that causes a conflict in material balance (as in Figure 15.1b). If a conflict occurs, we need to *backtrack* and adjust the design of some of the separators to alleviate this conflict.

Handling constraints locally rather than globally has some advantages. The expert system is far more flexible and can handle more complex situations. It may even temporarily *violate* a known constraint, develop a flowsheet, and then go back later and correct all violations. This ability allows us to develop unique processes unavailable from an expert system using strictly global constraints.

However, the use of local constraints has its problems. The costs of this added flexibility include:

- *Time is wasted*- every time a conclusion is made, we need to check for conflicts, which consumes valuable computer time.
- *Resolution is not guaranteed*- when a conflict does occur, we cannot guarantee that it can be resolved. The system may have zero degree of freedom, and thus no parameter available for adjustment and resolution.
- *Proper resolution can be arbitrary and difficult*- if degrees of freedom *are* available for adjustment, we still may not be able to

determine the best conflict-resolution option. The resolution may be arbitrary rather than knowledge-based.

2. Handling Constraints Globally

To avoid these problems, EXSEP handles all constraints *globally* through the component assignment matrix. The CAM represents the global allocation of all component flow rates to each product. All separators must follow this allocation.

For example, consider a system with seven products and ten components. We repeatedly split the CAM into smaller sub-matrices, until we see the following CAM with two products and four components:

$$\begin{array}{c c} & \begin{array}{cccc} c3 & c4 & c5 & c6 \end{array} \\ \begin{array}{c} p2 \\ pl \end{array} & \left[\begin{array}{cccc} 0 & 7 & 8.5 & 9 \\ 12 & 10 & 8.5 & 0 \end{array}\right] \end{array}$$

We desire to make the sloppy split pl/p2. With c4 and c5 as the light- and heavy-key components, we calculate nonkey component distributions, and get the following distillate and bottoms products:

- Distillate: 11.9 c3 + 10 c4 + 8.5 c5 + 4.7 c6
- Bottoms: 0.1 c3 + 7 c4 + 8.5 c5 + 4.3 c6

Propane (c3) is a lighter-than-light-key (LLK) component, and the specifications require a recovery ratio of 1.0/0.0. Since $(d/b)_{c3}$ = 0.992/0.008, we recover virtually all of the propane in the overhead, and meet the specification. Therefore, the split is feasible on the LLK side.

Now let us investigate the heavier-than-heavy-key (HHK) component, n-hexane (c6). We desire $(b/d)_{c6}$ = 1.00/0.00. However, from the nonkey component-distribution test, we find that $(b/d)_{c6}$ = 0.61/0.39. This distribution is undesirable because it does not obey the *global* material balance.

Because this split violates the global material balance, we classify it as infeasible and reject it from consideration. If we were handling constraints locally, we *could accept* this split (with its undesirable c6 distribution), and attempt to correct the material balance through backtracking to previously specified separators.

In general, then, if a potential solution violates a global constraint, EXSEP classifies it as infeasible and eliminates it from consideration. Therefore, once the program specifies a separator, it never needs to readdress that separator with backtracking. If EXSEP draws a conclusion, it will never contradict that conclusion later in the program.

By handling constraints globally, the expert system executes very quickly. It does not waste memory retaining previously designed separators for potential backtracking. Nor does it waste time backtracking. And finally, it does not require arbitrary conflict-resolution strategies.

On the down side, handling constraints globally may be restrictive.

We may not be able to develop the unique designs possible with only local constraints. However, this loss of possibilities is a small price. Handling constraints globally provides better program control, and more accurate and efficient problem-solving.

15.2 SEARCH

A general rule of thumb in expert systems is:

The better the knowledge representation, the easier is the search.

EXSEP amply demonstrates this principle. Its search strategy is relatively straightforward and simple because EXSEP benefits from years of detailed chemical engineering research in separation science. Researchers have investigated and debated ways of sequencing separators to give the economically optimal solution. EXSEP relies heavily on the rank-ordered heuristic scheme of Nadgir and Liu (1983), discussed in section 13.6. Because of this extensive research, the knowledge representation in EXSEP (in particular, the heuristics and feasibility analysis) is very good; it makes the search quite manageable.

Good knowledge representation and a manageable search is important in implementing EXSEP on a personal computer. Every step an expert system takes must be maintained in the computer memory to facilitate backtracking. With a large, drawn-out search, memory consumption grows exponentially. An expert system with an inefficient search will run out of memory very quickly on a personal computer. Since EXSEP's search is straightforward, however, it can run on a personal computer easily .

This section details the search itself, and discusses how the knowledge representation streamlines the search.

A. Overall Control Strategy: Plan-Generate-Test

EXSEP's search strategy has a number of facets. It uses a plan-generate-test strategy to:

- *Plan*- trim the state space to a manageable level.
- *Generate*- identify and create feasible splits for further consideration.
- *Test*- evaluate and choose an attractive split from the ones generated.

To implement this strategy, EXSEP uses the following chemical engineering tools:

- Stream-bypass analysis
- Heuristic D1 via the component assignment matrix (CAM)
- Feasibility analysis via the separation specification table (SST)
- Rank-ordered heuristics (M1, S1, S2, C1, C2)

Figure 15.3 represents the overall structure of EXSEP. EXSEP uses the preceding tools to search and determine the best sequence. Bypass analysis and CAM construction (which is an implicit implementation of heuristic D1) are part of the *PLANNER*, the portion that "plans" the search by trimming the state space. Feasibility analysis is part of the *GENERATOR*, which

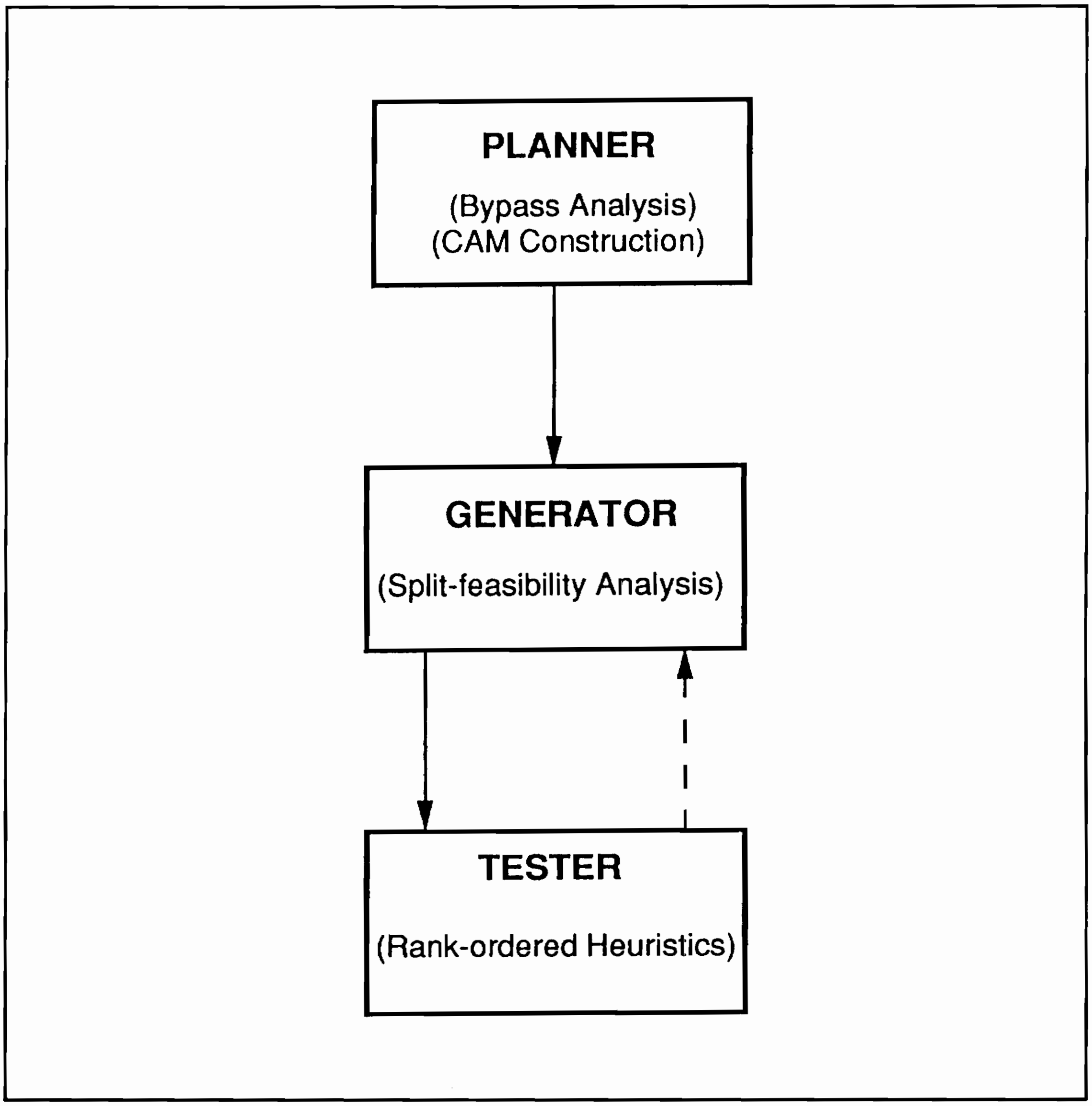


Figure 15.3. EXSEP's plan-generate-test control structure.

"generates" feasible splits from the existing state space and passes these splits on for further analysis. Finally, the *TESTER* uses rank-ordered heuristics to "test" how attractive different feasible splits are and choose the most desirable split.

Now that we have an overall view of the search control strategy, we will discuss each aspect in more detail.

B. Aspects of Search and Control

1. Design Heuristic D1

Design heuristic D1 says to favor the smallest product set. The CAM implements D1 implicitly, and this implementation reduces the search space tremendously.

In separation sequencing, we try to control component concentrations in each process stream. Because concentration is a *continuous variable*, there is, literally, an infinite number of possible configurations.

Heuristic D1, however, immediately reduces number of possibilities and the search space becomes manageable. If a CAM created by heuristic D1 has P products and C components, the number of separation options, $\tilde{O}$, open for that CAM is:

$$(C-1) \leq \tilde{O} \leq P(C-1)$$

Note that the search space is *for this CAM only*, not for the entire sequence. The number of options open to us for the entire sequence is, of course, considerably larger.

To demonstrate the effect of heuristic D1, let us investigate a system with seven products and thirteen components. With P = 7 and C = 13, heuristic D1 reduces the size of the search for the *initial* split (not for the entire sequence) down from infinity to $12 \leq S \leq 84$.

Heuristic D1 is a ***domain-dependent*** search tool. In section 11.1, we discussed ***domain-independent*** tools, such as best-first and depth-first searches. These searches are not problem-specific, and can work in an expert system solving any type of problem, be it medicine, chemistry, engineering, or finance. A domain-dependent search such as heuristic D1, however, *is* problem-specific. It applies to this problem only -- in this case, multicomponent separation sequencing.

After applying heuristic D1, EXSEP then performs a stream-bypass analysis on the CAM. Bypass may or may not reduce the state space. The driving force for bypass is primarily economical; we reduce the mass load of the system, thereby reducing capital and operating costs. In addition, bypass may reduce the size of the state space, and/or make a previously infeasible sloppy split feasible.

2. Feasibility Analysis

Heuristic D1 streamlines the search by generating a set of potential splits, assuming the splits are technically feasible. Feasibility analysis via the separation specification table (SST) goes one step further by analyzing that set of splits for technical feasibility. Based on the Fenske equation and the global material-balance constraints, feasibility analysis eliminates splits that are technically or practically infeasible. EXSEP deems splits infeasible if they do not obey the global material balance.

If a split is feasible, it is goes into a special part of the database and is labelled as such. This step finishes the "generation" portion of the program. We have generated a number of technically feasible splits that obey the global material balance. These generated splits are passed on to the rank-ordered heuristics, the final stage of the search.

3. Rank-Ordered Heuristics

When EXSEP implements the rank-ordered heuristics, it has already generated a set of technically feasible splits. The program chooses the best split based on the heuristics of Nadgir and Liu (1983), recommends the split, and asks for the user's approval.

The expert system goes through the following steps in determining the split of choice:

> *Step 1: Implement Heuristic M1*- Heuristic M1 says to favor ordinary distillation over extractive distillation or processes requiring a mass-separating agent. To satisfy heuristic M1, EXSEP compares the relative volatility between the light-key and heavy-key components (i.e., $\alpha_{LK,HK}$) for every available split. If $\alpha_{LK,HK} < 1.10$, then EXSEP recommends

extractive distillation. If $\alpha_{LK,HK} \geq 1.10$, EXSEP recommends ordinary distillation.

Heuristic M1 is a state-space pruning tool. It identifies splits where normal distillation is inappropriate.

Step 2: Implement Heuristic S1- Heuristic S1 says to remove corrosive or hazardous components first. EXSEP checks to see if any components are corrosive or hazardous. If they are, their removal is an essential first split, and EXSEP recommends the split that removes them.

Heuristic S1 is a rule that actually chooses and executes the split, rather than simply pruning the state space.

Step 3: Implement Heuristic S2- Heuristic S2 says to perform difficult separations last. Thus, while heuristic S1 identifies the essential *first* splits, heuristic S2 identifies the essential *last* splits. EXSEP tests for difficult splits, that is, splits where the boiling point difference or relative volatility is close, but where normal distillation is still appropriate. If the LK/HK boiling point difference is < 10°C, EXSEP labels the split "difficult," and does not consider it until the very end.

Heuristic S2 is a state-space pruning tool. It identifies which splits to save until last.

Step 4: Implement Heuristic C1- Heuristic C1 says to remove the most plentiful product first. A product is deemed "most plentiful" in EXSEP if it: 1) has the highest flow rate of all products in the CAM, and 2) is at least 20% higher in flow rate than the product with the second highest flow rate. If a product meets these conditions, EXSEP eliminates splits that fail to isolate it.

Heuristic C1 is a state-space pruning tool to eliminate splits that fail to isolate the most plentiful product.

Step 5: Implement Heuristic C2- Heuristic C2 says to favor a 50/50 split. If two or more splits are fairly close to 50/50 (i.e., between 60/40 and 40/60), then choose the split with the highest coefficient of ease of separation (CES).

As with heuristic S1, heuristic C2 executes a split (rather than reducing the state space).

These rank-ordered heuristics will always recommend at least one split; they will never fail on their own. The user can reject the recommended split and force backtracking. When this happens, the EXSEP moves into the manual mode where the user can choose a split. If the user fails to choose a split, EXSEP begins the heuristic analysis over again.

15.3 PROLOG PROGRAMMING

This section covers the actual Prolog programming done in EXSEP. We introduce the overall program driver and work through each module. We then examine some specific Prolog programming techniques used in the actual program (listed in Appendix B).

A. Overview

The EXSEP disk for distribution is written in Turbo Prolog, available from Borland International, 4585 Scotts Valley Dr., P.O. Box 660001, Scotts Valley, CA 95066-9938. We chose Turbo Prolog for three reasons:

(1) The compiled program executes quickly.

(2) The software is inexpensive (available for less than $100 in 1990).

(3) The package has a good development interface for the programmer.

Turbo Prolog is different from more "standard" versions of Prolog, and does have limitations, including:

- *Compiled-only language*- Turbo Prolog does not come with an interpreter. All programs must be compiled.
- *Data typing*- Turbo Prolog uses data types. All data types used in

a predicate must be declared in a domain section.

- *Programmer memory management*- Turbo Prolog has no "built-in" memory management or optimization. All memory management is up to the programmer.
- *Assert and retract on database facts only*- all assertions and retractions must be done on facts that are designated as database types. Rules cannot be asserted or retracted.
- *No nested parentheses*- Turbo Prolog forbids nested parentheses. The only exception is in asserting and retracting facts.
- *The = predicate replaces* ***is***- in Turbo Prolog arithmetic, the **=** predicate is built-in and is equivalent to the **is** predicate, covered in our previous discussion of Edinburgh Prolog.
- *No meta-logical functions*- Turbo Prolog has no direct meta-logical predicates such as **var**, **nonvar**, **functor**, **arg**, etc., making it difficult to implement a truly "accessible" expert system.
- *Multiple-solution clauses*- Turbo Prolog contains the **findall** clause, but does not possess the **bagof** nor the **setof** clause. Consequently, we must implement some forms of iteration (i.e., forced backtracking through multiple solutions) through the **assert-retract-fail** techniques discussed in section 8.3.F.2.

We are also in the process of preparing a version of EXSEP for distribution for Arity Prolog users. Arity Prolog is more powerful than Turbo Prolog. All the limitations of Turbo Prolog, listed above, do not exist with Arity Prolog.

B. Program Driver

The overall program driver in EXSEP is the clause **run**. In EXSEP, the clause appears as:

```
run:-
     repeat,
     initialize(List,CList),
     develop_sequence(List,CList),
     report,
     fail.
```

This clause is a **repeat-fail** combination (section 8.3.F.1) where we first **initialize** the input. The Prolog variable **List** is the list of products in the component assignment matrix (CAM). The variable **CList** is the list of components. The **develop_sequence** clause does just that-- it develops the actual flowsheet. Finally, the driver calls **report** to present the results, and then the clause fails. When the clause fails, it backtracks to the **repeat** clause and is ready to input another problem.

The **develop_sequence** clause is perhaps more informative than the **run** clause. It the "heart" of EXSEP, and is written as:

```
develop_sequence([_],_):-!.          % base case: a row matrix CAM
develop_sequence(_,[_]):-!.          % base case: a column matrix CAM
develop_sequence(List,CList):-
        plan(List,CList),
        generate(List,CList),
        test(List,CList,TopList,TopCList,BotList,BotCList),
        develop_sequence(TopList,TopCLIst),
        develop_sequence(BotList,BotCList).
```

The **develop_sequence** clause is the actual flowsheet developer. It follows the plan-generate-test strategy discussed in section 15.2. (See sections 15.2 and 11.3C for more details on plan-generate-test.)

The **develop_sequence** clause first *plans* by setting up the CAM and performing the bypass analysis. It then *generates*, building the separation specification table and retaining feasible splits. The program then *tests* potential splits and recommends the desired split. When executing a split, EXSEP splits the CAM into two smaller sub-matrices, one for the overhead and the other for the bottoms.

The variables **List** (list of products) and **CList** (list of components) define the original CAM. After the split, the variables **TopList** and **TopCList** define the CAM for the overhead stream, while **BotList** and **BotCList** define the bottoms CAM. After obtaining the distillate and bottoms streams from the separator, **develop_sequence** goes into a recursive loop to further analyze and separate those streams.

The **develop_sequence** clause is recursive. The base cases are a one-product or a one-component CAM (i.e., either a row or column matrix). No more processing occurs once the base case is reached. The clauses **run**, **initialize**, **develop_sequence**, and **report** never fail, which makes the program more robust.

C. Program modules

EXSEP has five program modules. They are:

- *MAIN MODULE* - contains central driver clauses and program control.
- *BYPASS MODULE*- performs complete bypass analysis.
- *SST MODULE*- performs feasibility analysis and builds separation specification table. Passes feasible splits to the SPLIT MODULE.
- *SPLIT MODULE*- implements rank-ordered heuristics on the potential splits passed from the SST module.
- *UTILITY MODULE*- perform list processing and other utilities for all the modules.

The complete listing of the program appears in Appendix B. In this section, we discuss how each module is organized and what each achieves.

1. MAIN MODULE

The MAIN MODULE drives the entire program. Figure 15.4 shows the flowchart for the central activities of the MAIN MODULE.

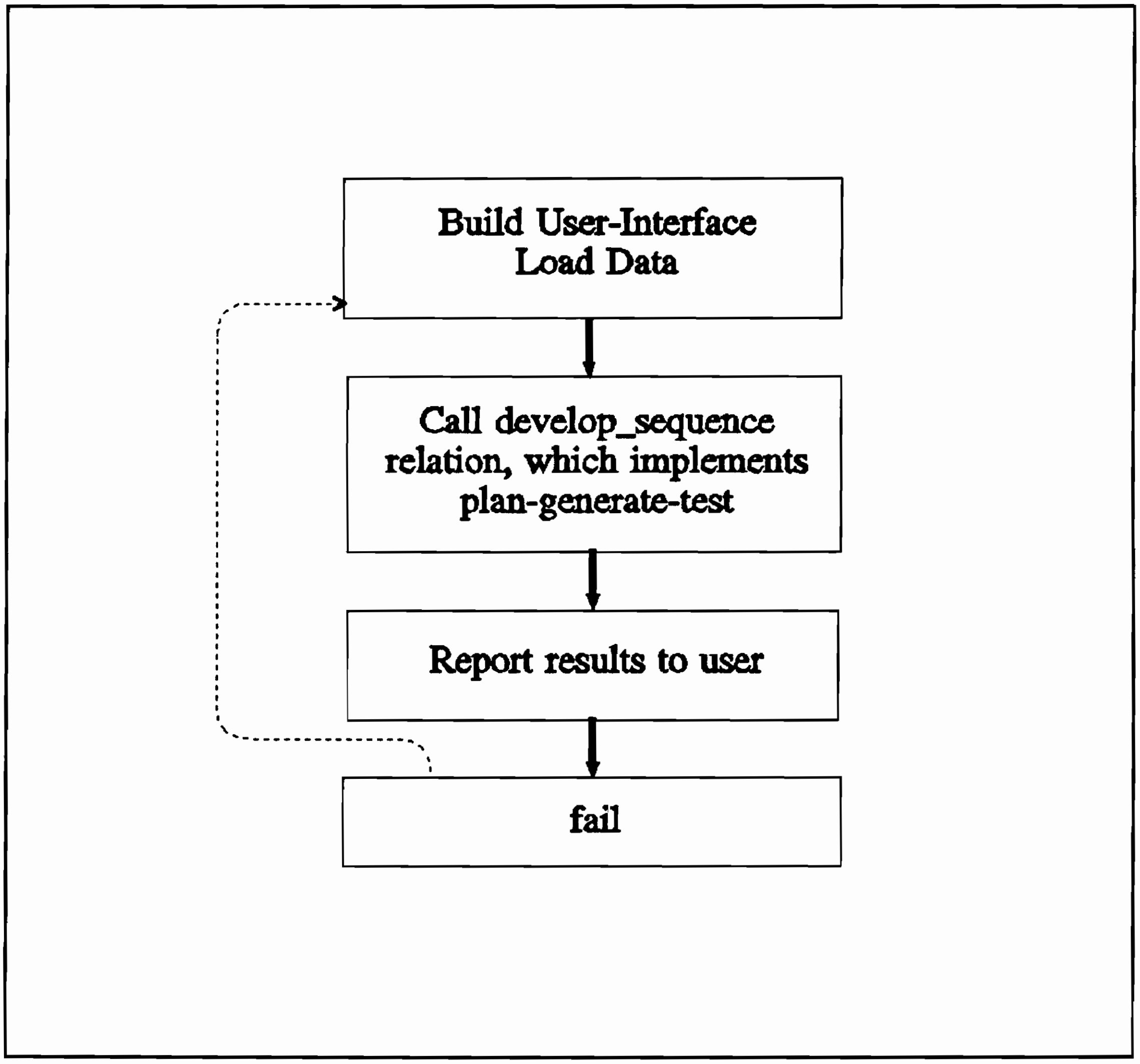

Build User-Interface
Load Data
Call develop_sequence relation, which implements plan-generate-test
Report results to user
fail

Figure 15.4. Flowchart of central activities of MAIN MODULE.

The MAIN MODULE contains the central program driver, the **run** clause. It also contains the central flowsheet-development clause,

develop_sequence. Table 15.1 lists the objectives achieved by the key clauses in the MAIN MODULE.

Table 15.1. Objectives of key clauses in the MAIN MODULE.

CLAUSE	PURPOSE
run	Drives program; coordinates entire program.
initialize	Develops EXSEP's window and menu system; inputs data via keyboard or file.
develop_sequence	Calls the plan-generate-test routine to develop the entire sequence.
report	Reports the final flowsheet results to the user.

plan	Prints the Component Assignment Matrix (CAM);
	calls the BYPASS MODULE;
	reprints the CAM if bypass has occurred.

generate	Calls the SST MODULE used for feasibility analysis.
test	Calls the SPLIT MODULE used to implement the rank-ordered heuristics and recommend a split.

2. BYPASS MODULE

The BYPASS MODULE performs the complete bypass analysis and recalculates the material balance if necessary. The BYPASS MODULE is called from the **plan** clause in the MAIN MODULE. The key sub-goal in the **plan** clause that executes the BYPASS MODULE is **Bypass(List,CList)**. **List** again represents

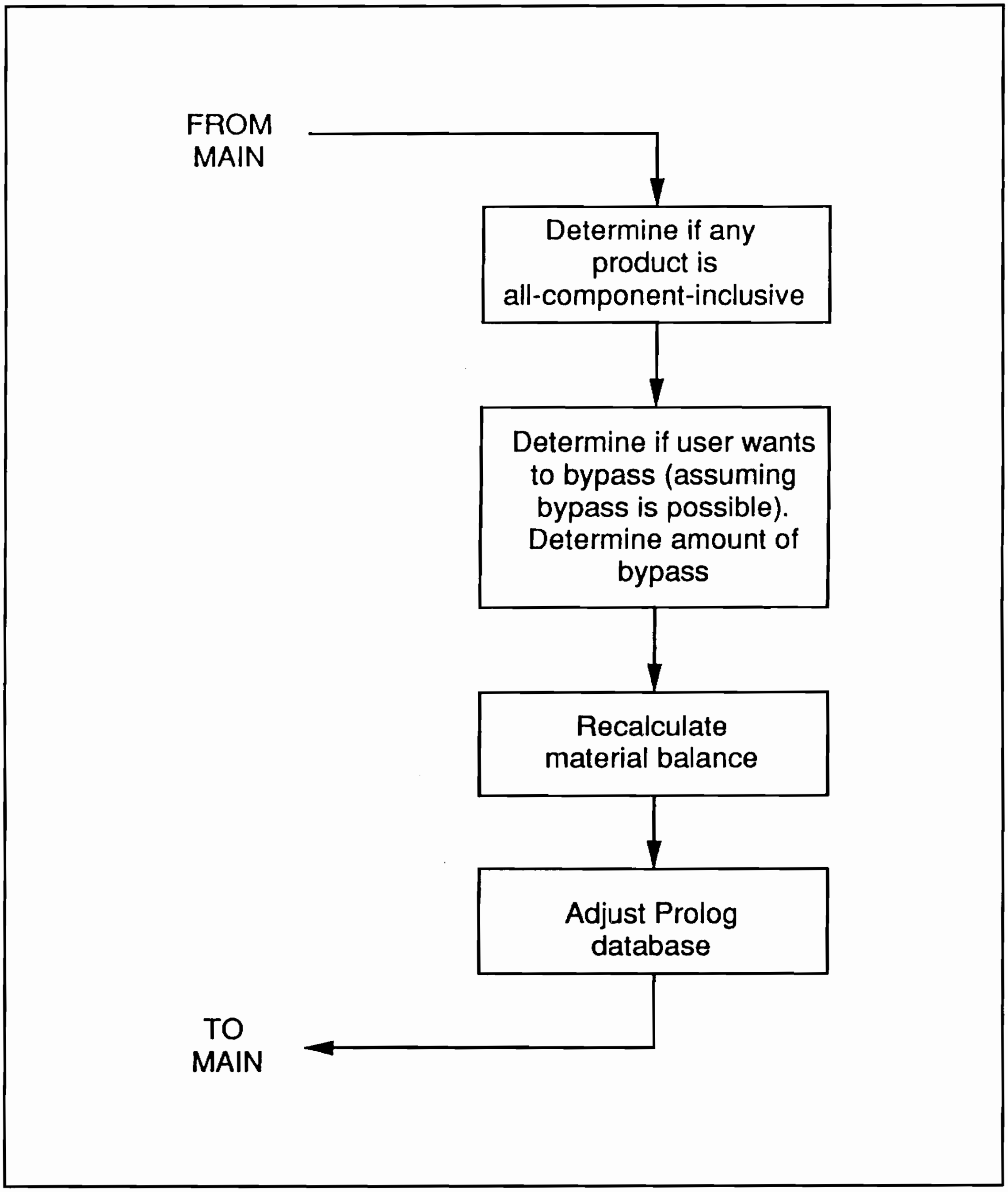

FROM
MAIN
Determine if any product is all-component-inclusive
Determine if user wants to bypass (assuming bypass is possible). Determine amount of bypass
Recalculate material balance
Adjust Prolog database
TO
MAIN

Figure 15.5. Flowchart of central activities of BYPASS MODULE.

the list of products, and **CList** represents the list of components, and together these variables specify the CAM.

The flowchart for the central activities of the BYPASS MODULE appears in Figure 15.5. The BYPASS MODULE contains three key clauses -- **bypass**, **check_bypassing**, and **solve_bypass_flow** -- their purpose is summarized in Table 15.2.

Table 15.2. Objectives of key clauses in the BYPASS MODULE.

CLAUSE	PURPOSE
bypass	Drives the BYPASS MODULE; switches program over to bypass window; calls main bypass-analysis clauses; cleans up database after analysis.
check_bypassing	Determines if bypass is possible; if possible, asks user for amount of bypass desired.
solve_bypass_flow	Recalculates CAM if bypass is done; adjusts database flow rates based on new material balance; reports results of bypass to user; also reports if bypass is not possible (as determined from **check_bypassing**).

3. SST MODULE

The SST module is the *generator* of the program, called from the MAIN MODULE by the **generate** clause. It does the feasibility analysis and generates a set of feasible splits for further testing. The ultimate purpose of the SST MODULE is to assert a fact into the database for every split given to it by the *planner* (CAM construction and bypass analysis).

This fact uses **dsst** (database separation specification table) as the functor, and is:

dsst(List,Clist,Split,Top,Bot,LK,HK,Delta,Listofd,Listofb,Status,Ease,CES)

where:

List = list of products, e.g., **[pl,p2,p3,p4]**.

CList = list of feed components, e.g., **[cl,c2,c3,c4]**.

Split = "sharp" or "sloppy," depending on the split.

Top = list of top products, e.g., **[pl,p2]** for a sloppy split, or **[cl,c2]** for a sharp split.

Bot = list of bottoms products, e.g., **[p3,p4]** for a sloppy split, or **[c6,c7]** for a sharp split.

LK = light-key component in the split

HK = heavy-key component in the split

Delta = normal boiling-point temperature difference between the light-key and heavy-key components, in °C.

Listofd = list of overhead recovery fractions for each component in **CList**, e.g., for **[c4,c5,c6,c7]**, a **Listofd** could be **[0.98,0.98,0.60,0.02]**.

Listofb = list of bottoms recovery fractions for each component in **CList**, e.g., for **[c4,c5,c6,c7]**, a **Listofb** could be **[0.02,0.02,0.40,0.98]**.

Status = "feasible" or "infeasible," depending on component-recovery specification test and nonkey-component distribution test.

Ease = "difficult" or "easy," depending on the temperature difference between the LK and HK components. This parameter applies to heuristic S2. If a split is labelled "difficult", it must be an essential last according to S2.

CES = coefficient of ease of separation for the split.

Figure 15.6 shows the flowchart for the central activities of the SST MODULE, and Table 15.3 summarizes the key clauses.

Table 15.3. Objectives of key clauses in the SST MODULE.

CLAUSE	PURPOSE
separation_ specification_ table	Drives the SST MODULE; switches program over to sst window; calls main feasibility analysis clauses; cleans up database after analysis.
sharp_split_sst_ driver	Develops sst for every sharp split given by the planner; uses the Prolog **assertz** predicate to update the database with the **dsst** fact for every sharp split.
sloppy_split_sst_ driver	Does feasibility analysis for every sloppy split given it by the planner; uses the Prolog **assertz** predicate to update the database with the **dsst** fact for every sloppy split.

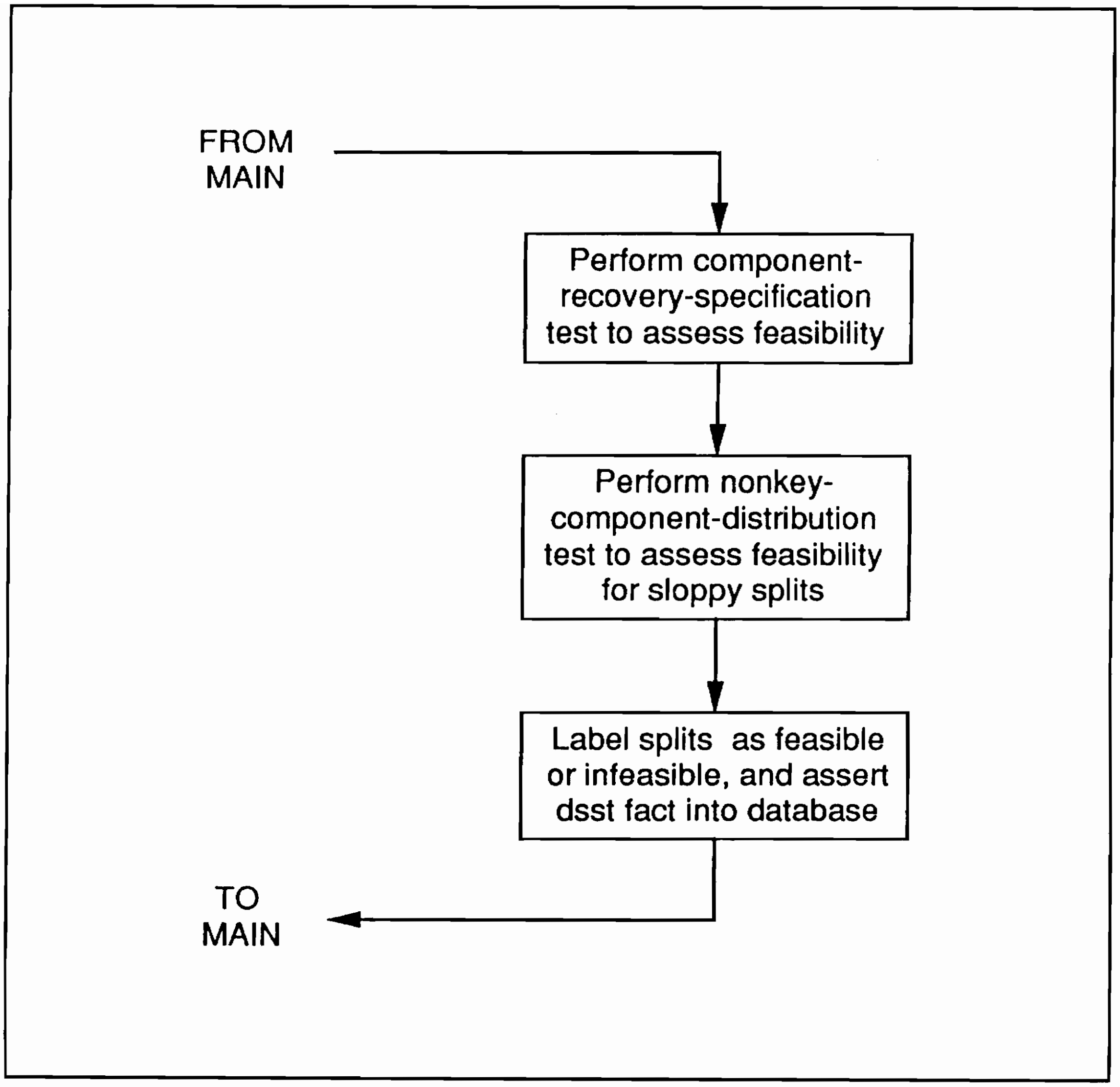

FROM MAIN
Perform component-recovery-specification test to assess feasibility
Perform nonkey-component-distribution test to assess feasibility for sloppy splits
Label splits as feasible or infeasible, and assert dsst fact into database
TO MAIN

Figure 15.6. Flowchart of central activities of SST MODULE.

4. SPLIT MODULE

The SPLIT MODULE, called by the **split** clause in the MAIN MODULE, takes feasible splits generated by the SST MODULE, performs heuristic analysis on them, and recommends the best split.

The **split** clause is the central clause in the module. It has an arity of six, and is defined as:

split(List,CList,TopList,TopCList,BotList,BotCList)

where:

- **List** = list of products
- **CList** = list of components in the feed
- **TopList** = list of products in the overhead
- **TopCList** = list of components in the overhead
- **BotList** = list of products in the bottoms
- **BotCList** = list of components in the bottoms

Essentially, the variables **List** and **CList** define the original input CAM. Splitting the CAM creates two sub-matrices, one for the overhead, and the other for the bottoms. The variables **TopList**, **TopCList**, **BotList**, and **BotCList** define these two sub-matrices.

As mentioned, the central clause in the split module is the **split** clause. The SPLIT MODULE has twenty-two rules with **split** as the head. Table 15.4 summarizes these rules, which are numbered in sequential order in the SPLIT MODULE listing in Appendix B.

Table 15.4. The split clauses used in SPLIT MODULE.

RULE	FUNCTION	DESCRIPTION
1	Base case	Ends recursion if **List** = empty list [ ].
2	Base Case	Ends recursion if **List** = single element list [ ].
3	Base Case	Ends recursion if **CList** = empty list [ ].
4	Base Case	Ends recursion if **CList** = single element list [ ].
5	Heuristic M1	Identifies **dsst** facts from database where relative volatility is too low to use ordinary distillation and extractive distillation is required.
6	Heuristic M1	Restores **dsst** database corrupted by **assert-retract-fail** used in rule 5; fails, forcing application of other heuristics.
7	Heuristic S1	Identifies corrosive component. If it is the most volatile component, recommends splitting it off and isolating it in the overhead.
8	Heuristic S1	Identifies corrosive component. If it is the least volatile component, recommends splitting it off and isolating it in the bottoms.
9	Heuristic S1	Identifies corrosive component. If it is neither the most nor the least volatile component, recommends applying other heuristics.
10	Heuristic S1	Determines that no corrosive component exists, and fails, forcing application of other heuristics.
11	Heuristic S2	Identifies difficult separations due to low boiling-point differences; labels them as essential last splits; fails, forcing application of other heuristics.
12	Heuristic S2	Determines that no split is difficult and heuristic S1 does not apply; fails, forcing application of other heuristics.
13	Heuristic S2	Restores **dsst** database corrupted by **assert-retract-fail** used in rule 11; fails, forcing application of other heuristics.
14	Heuristic C1	Identifies that no product is most plentiful; fails, forcing application of other heuristics.

15	Heuristic C1	Identifies that no product is most plentiful; fails, forcing application of other heuristics.
16	Heuristic C1	Identifies the most plentiful product; detects that this product is not all-component-inclusive; retracts all splits that violate heuristic C1; fails, forcing application of other heuristics.
17	Heuristic C1	Identifies most plentiful product; detects that this product is all-component-inclusive, and does not apply heuristic C1; fails, forcing application of heuristic C2.
18	Heuristic C2	Identifies splits closest to 50/50 split ratio; decides whether to use CES; recommends split based on split ratio and CES.
19	Heuristic S2	Implements difficult separation when no other split remains; chooses split with highest CES.
20	Program control	Detects that no split has been done; asks if user desires to split manually or heuristically; implements manual split; fails if user desires heuristic search.
21	Program control	Restores **dsst** database corrupted by **assert-retract-fail** used in rule 16; fails, forcing application of heuristic analysis again.
22	Program control	Recursively calls the same **split** clause, since no split was executed. This effectively begins heuristic analysis all over again.

5. UTILITY MODULE

The UTILITY MODULE houses helpful tools, or utilities, for all the modules. Unlike other modules, the UTILITY MODULE is never explicitly called. Relations inside the UTILITY MODULE are utilized on an as-needed basis. The UTILITY MODULE includes some of the typical list-processing clauses, such as **reverse**, **length**, **member**, **delete**, etc.

D. Application of Prolog Techniques

1. Memory Usage

Turbo Prolog is inexpensive, simple, and easy to use. One area we must watch very closely when using it, however, is memory-management. As previously mentioned, Turbo Prolog has no built-in memory management capabilities; memory management and control rests almost entirely in the programmer's hands. In addition, the entire program must utilize less memory than the "MS DOS barrier" of 640 KB.

Memory usage is *the* key problem in implementing expert systems on microcomputers. As an AI program executes (in Prolog or LISP), the system must remember the inference chain to facilitate backtracking. Because the computer adds each step that the program takes to its memory, complex programs quickly consume enormous memory space.

To exemplify how much memory an expert system can require, we mention that Gold Hill Computer Company sold a LISP package known as Gold Hill Common LISP (GCLISP). The company recommended purchasing an add-on board ranging from 8 MB to 24 MB in extended RAM to develop expert systems. That is a lot of memory for a personal computer. EXSEP was implemented on a microcomputer with only 640 KB of RAM. Clearly, getting EXSEP to run on a standard PC required a lot of "squeezing" (although EXSEP has one advantage-- compiled Prolog consumes less RAM than interpretive LISP).

Probably the most frequently used memory-conservation technique in EXSEP is the **fail** predicate (see section 6.3.C). Instead of using recursion to obtain multiple solutions, EXSEP uses backtracking. The **fail** predicate forces Prolog to backtrack, which uninstantiates variables. As EXSEP generates multiple solutions, it continuously instantiates and uninstantiates variables. Consequently, the memory load remains constant, whereas in recursion, the load on the computer memory grows steadily.

As an example, let us consider the **sharp_split_sst_driver** clause in the SST MODULE:

```
sharp_split_sst_driver(List,CList):-
   process_keys(LK,HK),
   sharp_split_sst(List,CList,LK,HK),
   retract(process_keys(LK,HK)),
   fail.
sharp_split_sst_driver(_,_):- !.
```

If we have a four-component system (c4, c5, c6, c7), before calling **sharp_split_sst_driver**, EXSEP will assert the following facts into Prolog's database:

```
process_keys(c4,c5).
process_keys(c5,c6).
process_keys(c6,c7).
```

When called, **sharp_split_sst_driver(List,CList)** has **List** and **CList** as previously instantiated input variables. Assume that Prolog makes the following call:

sharp_split_sst_driver([p1,p2,p3,p4],[c4,c5,c6,c7])

In English, the clause says "develop the SST for sharp splits with a CAM defined by products p1, p2, p3, and p4, and components c4, c5, c6, and c7."

The **sharp_split_sst_driver** clause goes through the following steps:

Step 1: Prolog matches **process_keys(LK,HK)** with the fact **process_keys(c4,c5)**, instantiating the light key, **LK**, to **c4** and the heavy key, **HK** to **c5**.

Step 2: Prolog calls **sharp_split_sst([p1,p2,p3,p4], [c4,c5,c6,c7], c4, c5)**. This clause assesses the feasibility of sharp split c4/c567. Sharp splits are always feasible. The clause calculates the Coefficient of Ease of Separation, and asserts the essential split information into the database. Importantly, the **sharp_split_sst** clause is *deterministic*. It has only one solution; backtracking into it results in a **fail**.

Step 3: Prolog now retracts the fact **process_keys(c4,c5)** from the database, and two facts remain:

```
process_keys(c5,c6).
process_keys(c6,c7).
```

Step 4: Prolog now hits the **fail** predicate. It immediately fails, and backtracks attempting to satisfy the **sharp_split_sst_driver** goal. Both the **sharp_split_sst** and the **retract** clauses are deterministic. Prolog backtracks over them, and goes to the next nondeterministic clause, **process_keys**.

Step 5: Both variables **LK** and **HK** are uninstantiated and become free when Prolog backtracks into **process_keys**. The goal becomes **process_keys(LK,HK)**, and now Prolog matches it with the fact **process_keys(c5,c6)**. The variables **LK** and **HK** are instantiated to **c5** and **c6**, respectively, and the process begins all over again.

Step 6: The new call is **sharp_split_sst([pl,p2,p3,p4], [c4,c5,c6,c7], c5, c6)**. EXSEP analyzes the c45/c67 split and asserts into the database. Then Prolog retracts **process_keys(c5,c6)**, leaving only **process_keys(c6,c7)** in the database.

Step 7: Prolog again **fails**, and backtracks until matching **process_keys(c6,c7)**. The new call becomes **sharp_split_sst(**

[p1,p2,p3,p4], [c4,c5,c6,c7], c6, c7). EXSEP analyzes the split c456/c7, asserts it into the database, and then retracts **process_keys(c6,c7)** from the database.

Step 8: Prolog hits the **fail** predicate again. Because no **process_keys** fact remains, the first **sharp_split_sst_driver** clause fails. Prolog backtracks to the second **sharp_split_sst_driver** clause.

Step 9: The second clause is :

```
sharp_split_sst_driver(_,_):- !.
```

Having two anonymous variables, this clause matches and takes no action except the cut (this prevents any backtracking into the clause). As a result, the **sharp_split_sst_driver** succeeds and Prolog moves on to other sub-goals.

The **sharp_split_sst_driver** clause analyzes all three splits (c4/c567, c45/c67, and c456/c7) using backtracking instead of recursion. Using the **fail** predicate thus saves computer memory.

2. Cut

EXSEP uses the cut (sections 4.1 to 4.4) extensively as a means of improving program efficiency. In scientific and engineering expert systems, we frequently run into situations that are mutually exclusive. *Mutual exclusivity is an ideal place to use the cut.*

As a simple example of mutual exclusivity, we consider the **assess_feasibility** clause:

```
assess_feasibility(feasible, feasible, feasible):- !.
assess_feasibility(_,_,infeasible):- !.
```

We can view this clause as:

assess_feasibility(*LLK side*,*HHK side*,*Answer***):- !.**

This clause assesses the feasibility of nonsharp splits after completing the nonkey-component distribution test. The first clause says, "a split is feasible if the nonkey-component distributions are feasible on both the LLK and HHK sides." The second clause says "if any other situation arises (the split is infeasible on either the LLK or HHK side), then the entire split is infeasible."

Clearly, feasibility is mutually exclusive. Either a split is feasible or it is not. When mutual exclusivity occurs, we can freely use the cut-- we know that the other clauses do not apply. Cuts used in mutually exclusive situations are examples of *green cuts*.

As a more complex example of mutual exclusivity, let us look at the **test_LLK_feasibility** clause, shown in Figure 15.7. The clause compares $(d/b)_{desired}$ and $(d/b)_{calculated}$ recursively for all components. If the base case occurs, the split is feasible. If a fail occurs, the last clause marks the split as infeasible. Note that cuts follow both the feasible situation (the base case) and the infeasible situation (the "fail" case, or, the last clause) -- their results are mutually exclusive.

```
/*********************************************************
 * The test_LLK_feasibility clause has four cases:       *
 * First Clause: the base case; if it gets here, the split*
 *   is feasible.                                        *
 * Second Clause: sharp split desired on LLK; split is   *
 *   is feasible if calculated (d/b)LLK >= 0.93/0.07     *
 * Third Clause: nonsharp split desired on LLK; split is *
 *   feasible if calculated (d/b)LLK is within +/- 20%   *
 *   of desired (d/b)LLK.                                *
 * Fourth Clause: split is infeasible if it fails clauses *
 *   two and three above. No more recursion needed.      *
 *********************************************************/

   test_LLK_feasibility(_,_,[],[],"feasible"):-!.

   test_LLK_feasibility(Top,Bot,[HC|TC],[HR|TR],LStatus):-
        component_flow(Top,HC,TFLow),
        component_flow(Bot,HC,BFLow),
        solve_ratio(TFLow,BFLow,Desireddoverb),
        Desireddoverb >= 49.0,
        HR >= 13.28571429,
        test_LLK_feasibility(Top,Bot,TC,TR,LStatus).

   test_LLK_feasibility(Top,Bot,[HC|TC],[HR|TR],LStatus):-
        component_flow(Top,HC,TFLow),
        component_flow(Bot,HC,BFLow),
        solve_ratio(TFLow,BFLow,Desireddoverb),
        Lowerbound= 0.8*Desireddoverb,
        Lowerbound <= HR,
        Upperbound = 1.2*Desireddoverb,
        Upperbound >= HR,
        test_LLK_feasibility(Top,Bot,TC,TR,LStatus).

   test_LLK_feasibility(_,_,_,_,"infeasible"):-!.
```

Figure 15.7. The test_LLK_feasibility clause.

In EXSEP, we generally avoid the use of the *negation-as-failure* (section 4.3.C) achieved explicitly through the *cut-fail* combination. Programs are more readable using the **not** predicate instead (see section 6.3.E). For instance, consider the **split** clause applying heuristic S1, shown in Figure 15.8.

```
   split(List,CList,_,_,_,_):-
        not(ddsst(List,CList,_,_,_,_,_,_,_,_,"feasible",
"difficult",_)),
        write("No splits are particularly difficult."),nl,
        write("Heuristic S1 does not apply, and no splits\nare
identified as essential last splits."),nl,
        shiftwindow(N),
        pause(N),
        fail.
```

Figure 15.8. Use of the not predicate.

The statement **not(ddsst(List,CList,_,_,_,_,_,_,_,_,"feasible", "difficult",_))** is a test in this clause done before any action is taken to ensure that no split exists that is technically feasible, but practically difficult. If such a split exists, EXSEP can report that heuristic S1 does not apply. The **not** predicate implements this heuristic easily. We *could* have used **cut-fail**, however, as shown in Figure 15.9.

You may wish to trace through both of these and verify that they are

```
     split(List,CList,_,_,_,_):-
          test_split,
          write("No splits are particularly difficult."),nl,
          write("Heuristic S1 does not apply, and no splits\nare
  identified as essential last splits."),nl,
          shiftwindow(N),
          pause(N),
          fail.

     test_split:-
          ddsst(List,CList,_,_,_,_,_,_,_,_,"feasible","difficult",_),
          !,fail.
     test_split.
```

Figure 15.9. Explicit implementation of cut-fail on the split clause.

procedurally identical.

3. Other Efficiency Techniques

EXSEP is in no way "fully optimized"; we welcome comments from EXSEP users about possible improvements for this prototype expert system. A number of other improvements can still be made to the system. For instance, the **reverse** relation in EXSEP has an arity of two, and is defined by:

```
     reverse([],[]).
     reverse([Head|Tail],List):-
          reverse(Tail,List1),
          append(List1,[Head],List).
```

A more efficient version (see section 3.4.B) uses an accumulator and eliminates the use of the **append** relation. This version is written as:

```
     reverse(L1,L2):-
          reverse_help(L1,[],L2).

     reverse_help([],L,L).
     reverse_help([H|T],L2,L3):-
          reverse_help(T,[H|L2],L3).
```

This definition illustrates the common optimization technique of *using more variables rather than more clauses* (see section 8.3.B), a technique used in a number of important areas in EXSEP.

Consider the assertion of a line of the SST into the database. EXSEP uses the **dsst** functor with an arity of thirteen:

```
     dsst(List,CList,Type,Top,Bot,LK,HK,Delta,Listofd,Listofb,
     Feasibility,Ease,CES),
```

We illustrate this functor with the following fact:

```
     dsst([p1,p2,p3,p4],[a,b,c,d],"sharp",[a,b],[c,d],"b","c",
18.2,[0.98,0.98,0.02,0.02],[0.02,0.02,0.98,0.98],"feasible","easy",8.5).
```

This fact, when matched, contains an enormous amount of information:

- **List**- the list of products in the CAM, **[pl,p2,p3,p4]**.
- **CList**- the list of components in the CAM, **[a,b,c,d]**.
- **Type**- the type of split (either sharp or sloppy), **"sharp."**.
- **Top**- the list of overhead products, **[a,b]**.
- **Bot**- the list of bottoms products, **[c,d]**.
- **LK**- the light-key component, **"b."**
- **HK**- the heavy-key component, **"c."**
- **Delta**- the normal boiling-point temperature difference between the light-and heavy-key components, **18.2** K.
- **Listofd**- the list of overhead recovery fractions for each component in **CList**, **[0.98,0.98,0.02,0.02]**.
- **Listofb**- the list of bottoms recovery fractions for each component in **CList**, **[0.02,0.02,0.98,0.98]**. (Note that d + b = 1.)
- **Feasibility**- whether the split is feasible or infeasible (using ordinary distillation), or whether another process such as extractive distillation should be used. This split is **"feasible."**
- **Ease**- whether the split is difficult or easy based on heuristic S2. In this case, it is **"easy."**
- **CES**- the coefficient of ease of separation for the split, **8.5**.

Using more variables than clauses promotes more rapid processing of knowledge, but generally makes the program more difficult to read.

As another example of an efficiency technique, EXSEP follows the principle "fail as soon as possible" (section 8.3.E). This principle prevents wasting time on actions that are doomed to failure. Consider the implementation of heuristic M1 from the SPLIT MODULE in EXSEP, given below:

```
/***************************************************************
 * Heuristic M1: Favor ordinary distillation and avoid MSA's.*
 * However, if the relative volatility between the LK and HK *
 * components is < 1.10, do not use distillation. This clause*
 * asserts "extractive distillation" for status if the       *
 * relative volatility is < 1.10.                            *
 ***************************************************************/
    sharp_split_sst(List,CList,LK,HK):-
       k_value(LK,KLK),
       k_value(HK,KHK),
       Alpha = KLK/KHK,
       Alpha <= 1.10,       % this is the key test for the clause.
       split_above(CList,LK,TopComponents,BotComponents),
       sharp_split_dandb(CList,TopComponents,BotComponents,Listofd,
Listofb),
       get_delta_temp(LK,HK,Delta),
       assertz(ddsst(List,CList,"sharp",TopComponents,BotComponents,LK,
HK,Delta,Listofd,Listofb,"extractive distillation","neither",0),table),
       !.
```

This clause attempts to fail as soon as possible. It gets the K-values for the light- and heavy-key components and immediately calculates the relative volatility, **Alpha**. If **Alpha** is greater than 1.10, the clause does not apply. It poses this test as soon as possible. If the clause fails, we have not wasted any time performing the **split_above**, **sharp_split_dandb**, or **get_delta_temp** clauses.

E. Developing Prolog Relations

For those accustomed to programming in procedural languages such FORTRAN, writing complex relations in Prolog can be difficult. This is one case where someone with prior computer experience may be at a disadvantage to the novice. Frequently, a competent FORTRAN programmer needs to "unlearn" some habits before he can program well in Prolog.

1. Evolutionary Development

Prolog program development, and particularly writing relations requiring rules and clauses, is an evolutionary process. After we have written a rule, we may need to change it later to accommodate some unforeseen situation. Such changes can be unsettling to those ingrained in the well-defined, algorithmic development process typical of FORTRAN.

Writing Prolog relations takes planning. We must first decide what we want each clause in the relation to achieve. We do not want an individual clause to achieve too much. If this happens, the clause will get complex, difficult to understand, and tough to debug or adjust.

Let us develop the relation for determining the separation specification table for horizontal splits. This relation must:

(1) be applied to all horizontal splits in the CAM.

(2) execute both the component-recovery specification test and the nonkey-component distribution test.

(3) determine what LK/HK combinations are possible for each horizontal split to do the nonkey-component distribution test.

(4) calculate the coefficient of ease of separation if the split is feasible.

(5) calculate the (d/b)'s for all nonkey components and retain this information in the database.

(6) assess whether the split is feasible or not, and retain this information in the database.

The above list includes only the objectives we can identify now. We will find more objectives as we develop the relation and get into the "nitty-gritty" detail. The above set is a good start, however.

2. Writing Clauses

The basic rule of thumb followed in EXSEP is: *write one clause for each objective*. This rule keeps program development focused and goal-oriented. Let us begin with the first objective: the relation must apply to all horizontal splits.

Horizontal splits are product splits (pl/p234, p21/p34, p123/p4, etc.). An astute programmer will realize that using recursion on the variable **List** (the list of products) will achieve objective one. However, to save computer memory, we want to use backtracking instead of recursion. Backtracking will require some type of **assert-retract-fail** technique.

What variables do we need to enable this clause to properly assess the feasibility of sloppy splits? What arity should the clause have? We are not sure at this point. We know that we at least need the variables **List** and **CList** to define the CAM. We can start with those two variables, and add more later as we need them. We first define the following clauses:

```
sloppy_split_sst_driver(List,CList):-
     process_product(Prod),
     sloppy_split_sst(List,Prod,CList),
     retract(process_product(Prod),table),
     fail.

sloppy_split_sst_driver(_,_):-!.
```

The above clauses avoid recursion and perform iteration, thereby saving computer memory. We have not yet defined the functors **sloppy_split_sst** and **process_product**, taking a "leap of faith" that we can properly define those relationships later so the entire relation will work. We have designed the clause to:

(1) pull a **process_product** fact out of the database; and

(2) use it to calculate the **sloppy_split_sst**, retract the fact, then iteratively repeat the process with another fact.

To complete the **sloppy_split_driver** relation, we need to first attack the **process_product** relation. The clause as written assumes **process_product** facts are in the database. We need to put those into the database *before* **sloppy_split_driver** is called. Therefore, the parent clause must maintain the following goal-ordering:

```
. . .,
initialize_products(List),
sloppy_split_sst_driver(List,CList),
. . .
```

The **initialize_products** clause will assert **process_product** facts into the database before the **sloppy_split_sst_driver** clause takes over. Now we need to define **initialize_products**:

```
initialize_products([_]):-!.
initialize_products([H|T]):-
     assertz(process_product(H),table),
     !,initialize_products(T).
```

Objective one has been achieved! The system will assert all pertinent **process_product** facts into the database before calling the **sloppy_split_sst_driver** clause. When Prolog does call **sloppy_split_sst(List,Prod,CList)**, the program will use the variable **Prod** to assess all horizontal splits in the CAM.

Now we need to define **sloppy_split_sst(List,Prod,CList)**. Our plan is as follows: to assess the horizontal split feasibility, we first need to know what overhead and bottoms products result from splitting the CAM horizontally. The variable **List** is the list of products. We can generate the overhead and bottoms products by cutting **List** into two sublists at the variable **Prod.** For example, if we have:

```
sloppy_split_sst([p1,p2,p3,p4],[a,b,c,d],p1)
```

we cut **List** at **p1**, **[p1 | p2, p3, p4]**. Product **p1** is the overhead, and products **p2**, **p3**, and **p4** are the bottoms. If we have:

```
sloppy_split_sst([p1,p2,p3,p4],[a,b,c,d],p2)
```

we cut **List** at **p2**, **[p1, p2 | p3, p4]**. Products **p1** and **p2** are overhead, and products **p3** and **p4** are bottoms. To generate these streams, we introduce **split_above**, the first sub-goal in the clause:

```
sloppy_split_sst(List,Prod,CList):-
     split_above(List,Prod,Top,Bot), ...
```

The **split_above** clause will split **List** at product **Prod**, giving the overhead stream **Top** and the bottoms stream **Bot**. The **sloppy_split_sst** clause is not done yet; we have addressed only the first sub-goal. We have introduced the relation **split_above** such that

```
split_above([p1,p2,p3,p4],p1,Top,Bot)
```

gives

```
Top = [p1]
Bot = [p2,p3,p4]
```

Likewise, the statement,

```
split_above([p1,p2,p3,p4],p2,Top,Bot)
```

gives

Top = [p1,p2]

Bot = [p3,p4]

Now we have our overhead and bottoms products resulting from the split. The streams are instantiated to the variables **Top** and **Bot**, respectively. The next issue is split feasibility. Let us first apply the component-recovery specification test. We introduce the clause:

```
component_recovery_specification_test(Top,Bot,CList,Status)
```

to assess feasibility. The input variables are **Top**, **Bot**, and **CList**. The output variable is **Status**, which can be instantiated to the atom "feasible" or the atom "infeasible." Our **sloppy_split_sst** clause is now:

```
sloppy_split_sst(List,Prod,CList):-
   split_above(List,Prod,Top,Bot),
   component_recovery_specification_test(Top,Bot,CList,Status),
   ...
```

Note that the **spilt_above** and **component_recovery_specification_test** relations remain undefined. We know *what* we want these clauses to achieve -- we have their objectives -- but we have not yet defined *how* to achieve those objectives. For the **sloppy_split_sst** clause to work properly, we will have to define that "how."

For now, let us make a mental note of the objectives of these undefined clauses, and remember that we need to work on them. But to maintain continuity, we will proceed with the development of the **sloppy_split_sst** clause. We can define the other clauses later.

Now that the component-recovery specification test is complete, we are at a decision point. If **Status** is "infeasible," then the split is infeasible and that fact needs to be asserted into the database. If **Status** is "feasible," we still need to subject the split to the nonkey-component distribution test. We can accomplish the "**Status** = infeasible" case with the following clause:

```
    sloppy_split_sst(List,Prod,CList):-
       split_above(List,Prod,Top,Bot),
       component_recovery_specification_test(Top,Bot,CList,Status),
       Status = "infeasible",
       component_recovery_dandb(CList,Top,Bot,Listofd,Listofb),
       assertz(ddsst(List,Clist,"Sloppy",Top,Bot,"any","any",0,Listofd,
Listofb,"infeasible", "difficult",0),table),!.

     sloppy_split_sst(List,Prod,CList):-
         split_above(List,Prod,Top,Bot), . . .
```

The first **sloppy_split_sst** clause is complete! It performs the following steps:

- splits the CAM horizontally at **Prod**, giving the **Top** and **Bot** products.
- performs the component-recovery specification test on the split to determine feasibility.
- calculates the recovery ratios for the split, instantiating **Listofd** and **Listofb**, if **Status** is infeasible.
- asserts the split characteristics into the database; in this case, the split is infeasible.
- fails and backtracks to the next clause if **Status** is feasible (note that when a clause fails, variables become uninstantiated).

We now subject the split to the nonkey-component distribution test. In the next clause, then, we need to call the nonkey-component distribution test. However, before we can carry out the test, we need to develop potential light- and heavy-key components.

Our second **sloppy_split_sst** clause, although not complete yet, is beginning to take shape. For now, we may write it as:

```
sloppy_split_sst(List,Prod,CList):-
   split_above(List,Prod,Top,Bot),
   determine_potential_LKs(Top,Bot,CList,CList,ListofLKs),
   . . .
```

Note that we do not need to call **component_recovery_specification_test** in the clause. The previous clause performed that check (the split must be feasible to reach this point). The **determine_potential_LKs** clause develops the list of potential light-key components for the horizontal split identified by **Top** and **Bot**. We can now move on to the nonkey-component distribution test.

A review of our objectives reveals that we again want to implement recursion on the **ListofLKs** so that all possible LK/HK combinations can undergo the nonkey-component distribution test. But again, we prefer backtracking instead of recursion. Consequently, we need to initialize the database with facts for the nonkey-component distribution test to retract and thus iterate. Therefore, we define the following relations:

```
sloppy_split_sst(List,Prod,CList):-
   split_above(List,Prod,Top,Bot),
   determine_potential_LKs(Top,Bot,CList,CList,ListofLKs),
   initialize_keys_for_nonkey_test(ListofLKs,CList),
   nonkey_component_distribution_test_driver(List,CList,Top, Bot),!.
```

We have finished our second **sloppy_split_sst** clause (not including the supporting clauses such as **split-above**, **determine_potential_LKs**, etc.). A potential problem exists, though, that we did not think of when we wrote our original objectives.

What if a horizontal split we are analyzing is actually sharp? Then the nonkey-component distribution test is unnecessary. We can add an additional objective to the **determine_potential_LKs** clause to remedy the situation: **determine_potential_LKs** must give us LK/HK options for *sloppy* splits only. If the split is sharp, we will not get that LH/HK pair. Instead, for a sharp split, **ListofLKs** will equal **[ ]**. We can take advantage of this in the following relation:

```
    sloppy_split_sst(List,Prod,CList):-
         split_above(List,Prod,Top,Bot),
         determine_potential_LKs(Top,Bot,CList,CList,ListofLKs),
         not(equal(ListofLKs,[])),
         initialize_keys_for_nonkey_test(ListofLKs,CList),
          nonkey_component_distribution_test_driver(List,CList,Top,
Bot),!.

    sloppy_split_sst(_,_,_):-!.
```

Here, if **ListofLKs** = **[ ]**, the clause will fail and go down to the third clause, **sloppy_split_sst(_,_,_):-!.** When this happens, the horizontal split is actually sharp and the nonkey-component distribution test is unnecessary. In fact, the clause does not need to take any action; the **sharp_split_sst** portion of the program handles all sharp splits.

In summary, we have defined the following goal-ordering:

```
    . . .
    initialize_products(List),
    sloppy_split_sst_driver(List,CList),
    . . .
```

coupled with the following clauses:

```
    initialize_products([_]):-!.

    initialize_products([H|T]):-
       assertz(process_product(H),table),
       !,initialize_products(T).

    sloppy_split_sst(List,Prod,CList):-
       split_above(List,Prod,Top,Bot),
       component_recovery_specification_test(Top,Bot,CList,Status),
       Status = "infeasible",
       component_recovery_dandb(CList,Top,Bot,Listofd,Listofb),
       assertz(ddsst(List,Clist,"Sloppy",Top,Bot,"any","any",0,Listofd,
Listofb,"infeasible", "difficult",0),table),!.
```

```
sloppy_split_sst(List,Prod,CList):-
   split_above(List,Prod,Top,Bot),
   determine_potential_LKs(Top,Bot,CList,CList,ListofLKs),
   not(equal(ListofLKs,[])),
   initialize_keys_for_nonkey_test(ListofLKs,CList),
   nonkey_component_distribution_test_driver(List,CList,Top, Bot),!.

sloppy_split_sst(_,_,_):-!.
```

We now have a better "feel" of how to develop clauses in Prolog. Developing Prolog clauses can sometimes be unsettling. We must take "leaps of faith," introducing relations not yet fully specified. When we develop the overall relations, we simply recognize that we will have to specify some supporting clauses later.

Frequently, clauses need to be adjusted to account for unforeseen situations. With **sloppy_split_sst** above, we have only guessed at what variables the relation requires. As we develop the supporting clauses such as **nonkey_component_distribution_test**, the specific variables required becomes clearer. As we define these supporting clauses, their arity may change and the program will evolve.

In the above example, we must realize that we have still not fully defined the **sloppy_split_sst** relation, since the clauses **split_above**, **component_recovery_specification_test**, **component_recovery_ dandb**, **determine_potential_LKs**, **initialize_keys_for_nonkey_test** and

nonkey_component_distribution_test are undefined. We define these supporting clauses *after* we work out the entire strategy for the parent relation, **sloppy_split_sst_driver**.

This example shows how voluminous knowledge-engineering can be. We set out to develop one relation, **sloppy_split_sst_driver** to calculate the separation specification table for horizontal splits. In the end, we need over ten supporting clauses.

15.4 CHAPTER SUMMARY

- EXSEP is written in Prolog and is a rule-based expert system using predicate logic.
- EXSEP develops flowsheets via *problem decomposition*. In addition, EXSEP applies material-balance constraints in a *global* fashion *only*.
- *Search* in EXSEP is straightforward. *Design heuristic D1* and *feasibility analysis* generate the search space and reduce it down to a few choices. *Rank-ordered heuristics* then determine the best split from the generated options.
- The *plan-generate-test* strategy controls EXSEP; at the end, **report** driver clauses convey results. Technically, EXSEP has five modules: MAIN, BYPASS, SST, SPLIT, and UTILITY.
- EXSEP uses a number of programming-efficiency techniques. The primary one is favoring forced backtracking via the **fail** predicate rather than recursion.
- Other efficiency techniques include: 1) the cut, 2) accumulators (more variables) instead of more clauses, and 3) operating on the principle *fail as soon as possible*.
- To develop Prolog relations, we first need to understand the objectives. As we write the relation, we take "leaps of faith" and introduce supporting clauses as the relation develops. These new clauses start out undefined; we know *what* we want the clause to achieve, but we have yet to written the Prolog code for them. After

fully specifying the parent relation, we move on to define the required supporting clauses.

A LOOK AHEAD

This chapter concludes Part Two of the book. We now have an understanding of not only how to program in Prolog, but also how to implement an expert system. We now move on to Part Three, where we shall introduce and discuss some expert systems developed and used in chemical engineering, and we shall also introduce a growing important topic in AI called artificial neural networks (ANNs).

REFERENCES

Cheng, S.H., and Y.A. Liu, "Studies in Chemical Process Design and Synthesis. 8. A Simple Heuristic Method for the Synthesis of Initial Sequences for Sloppy Multicomponent Separations", *Ind. Eng. Chem. Res.*, **27**, 2304 (1988).

Liu, Y.A., T.E. Quantrille, and S.H. Cheng, "Studies in Chemical Process Design and Synthesis. 9. A Unifying method for the Synthesis of Multicomponent Separation Sequences with Sloppy Product Streams", *Ind. Eng. Chem. Res.*, **29**, 2227 (1990).

Nadgir, V.M. and Y.A. Liu, "Studies in Chemical Process Design and Synthesis: Part V. A Simple Heuristic Method for Systematic Synthesis of Initial Sequences for Multicomponent Separations", *AIChE J.*, **29**, 926 (1983).

16

OVERVIEW OF KNOWLEDGE-BASED APPLICATIONS IN CHEMICAL ENGINEERING

16.1 INTRODUCTION

Knowledge-based systems, also known as expert systems, have begun to penetrate chemical engineering. With the growing availability of powerful and inexpensive hardware, using an expert system is now within the grasp of the practicing engineer.

Compared to typical chemical engineering computing, knowledge-based systems are unique. Instead of "crunching numbers," they contain chemical engineering knowledge, i.e., facts, rules, and heuristics. Having this knowledge, they have the ability to reason and solve problems.

In this chapter, we devote section 16.2 to expert-system project management, a section that answers questions such as: when is an expert system justified? What types of chemical engineering problems are suited for expert-system development? What is required for a successful expert system? Answering these questions is critical before initiating an expert-system project.

Sections 16.3 through 16.8 discuss actual AI applications in chemical engineering. AI software has begun to prove itself in the market, but there are also some challenges. We discuss both the benefits and limitations of AI techniques in chemical engineering, and finish with some of the challenges that lie ahead.

There have been six main application areas of knowledge-based systems in chemical engineering. They are:

- *Fault diagnosis*- process troubleshooting, i.e., determining the origins of process problems and recommending solutions.

- *Process control*- improving process control through utilization of qualitative process information, trend analysis, neural networks, etc.

- *Process design*- designing processes, selecting and estimating physical and thermodynamic properties, developing flowsheets, specifying equipment, ensuring process safety, assessing process flexibility, and estimating cost.

- *Planning and operations*- scheduling, developing procedures, assessing safety concerns, executing complex inter-related procedures, and aiding maintenance.

- *Modeling and simulation-* using qualitative reasoning and symbolic computing to model and simulate chemical processes.

- *Product design, development, and selection-* recommending chemical formulations, compositions, materials, process procedures, etc., required to design, develop, or select a new or existing product that achieves specified objectives.

AI applications in each of these areas have added a new dimension to chemical engineering. Although much of the work is still in the developmental stages, AI is sure to affect each of these areas in the future. Some areas could be completely revolutionized. The wise engineer will know both the strengths and limitations of AI techniques (Mathis, 1986).

REFERENCES AND FURTHER READING

Listed below are some general references on AI applications in chemical engineering.

Bartl, J.P., "Current Research in Computer Science at MIT," *Chem. Eng. Prog.*, **83** (9), 60 (1987).

Beltramini, L. and R.L. Motard, "Expert Systems in Chemical Engineering," *Chem. Biochem. Eng. Quarterly*, 2 (3), 199 (1988).

Ferrada, J.J., and J.M. Holmes, "Developing Expert Systems," *Chem. Eng. Prog.*, 86 (4), 34 (1990).

Gevarter, W.B., "Introduction to Artificial Intelligence," *Chem. Eng. Prog.*, 83 (9), 21 (1987).

Gray, N.A.B., "Artificial Intelligence in Chemistry," *Anal. Chim. Acta*, 210 (1), 9 (1988).

Mavrovouniotis, M.L., Editor, *Artificial Intelligence in Process Engineering*, Academic Press, San Diego, CA (1990).

Mathis, J.F., "Chemical Engineers Meet a Changing World," *Chem. Eng. Prog.* 82 (7), 17 (1986).

Pierce, T.H., and B. A. Hohne (Eds.), "Artificial Intelligence Applications in Chemistry," *ACS Symposium Series*, 306, American Chemical Society, Washington, DC (1986).

Russo, M.F. and R.L. Peskin, "Knowledge-Based Systems for the Engineer," *Chem. Eng. Prog.*, 83 (9), 38 (1987).

SanGiovanni, J.P and H.C. Romans, "Expert Systems in Industry: A Survey," *Chem. Eng. Prog.*, 83 (9), 52 (1987).

Stephanopoulos, G., "A Research Program in Artificial Intelligence," *Chem. Eng. Educ.*, p.182, Fall (1986).

Stephanopoulos, G., "The Future of Expert Systems in Chemical Engineering," *Chem. Eng. Prog.*, 83 (9), 44 (1987).

Stephanopoulos, G., J. Johnston, T. Kriticos, R. Lakshmanan, M. Mavrovouniotis, and C. Siletti, "DESIGN-KIT: An Object-Oriented Environment for Process Engineering," *Comput. Chem. Eng.*, 11 (6), 655 (1987).

Stephanopoulos, G., "Artificial Intelligence in Process Engineering - Current State and Future Trends," *Comput. Chem. Eng.*, 14 (11), 1259 (1990).

Stephanopoulos, G. and J.F. Davis (Editors), *Artificial Intelligence in Process Systems Engineering*, Vol. I, II, and III, CACHE Monograph Series, CACHE Corp., Austin, TX (1990).

Venkatasubramanian, V., "A Course in Artificial Intelligence in Process Engineering," *Chem. Eng. Edu.*, p.188, Fall (1988).

Wong, W.G., "PROLOG: A Language for Artificial Intelligence," *PC Magazine*, Oct. 14, pp. 247 (1986).

Special issues on expert systems and artificial intelligence:

Chem. Eng. Prog., 83 (9), 21-75 (Sept., 1987).

Comp. Chem. Eng., 12 (9-10), 853-1074 (1988).

Corrosion Reviews, 7 (2-3), 125-299 (1987).

16.2 DEVELOPMENT OF EXPERT SYSTEMS

Expert systems have received a lot of attention from the press in the 1980's. Unfortunately, much of the writing sensationalized the field. Expectations rose dramatically as some people, fueled by public speculation, began to overpromise. Misconceptions about what AI can and cannot do arose and they persist today. Many rushed into the field in search of quick answers and quick profits. Many AI researchers saw what was happening and feared a backlash once all the excitement wore off.

In 1988-90, things did begin to change. Some of the realities and limitations of AI techniques became evident. An AI backlash has resulted to a certain extent, but fortunately, it has not been wide-scale. Instead, the optimism remains, with a better sense of realism than before. Both the benefits and limitations are better appreciated.

When is an expert-system development warranted? This question must be answered before we initiate an expert-system project. D. Waterman (1986) gives some good guidelines on when we should consider using expert systems. Expert systems can be a very powerful asset. However, they also require significant investment in development cost. An expert system should be considered only when development is:

- Possible;
- Appropriate; and
- Justified.

The next sub-sections discuss what is meant by "possible," "appropriate," and "justified."

A. Identifying Expert-System Possibilities

Not all problems can be solved by expert systems. Before undertaking an expert-system project, we must identify the problem and project characteristics to ensure that an expert system is indeed possible. What are the characteristics of a problem that could possibly be solved using expert systems? Figure 16.1 summarizes the necessary requirements.

For a knowledge-based system to be possible, an expert on the field of application, called the *domain expert*, must exist. In addition, the expert must be able and willing to dedicate time to the project. Finally, the expert must have the ability to articulate his methods to the point where the *knowledge engineer* can correctly capture it into the knowledge base. The knowledge engineer is the AI programmer who takes the problem, casts it into an AI framework, and programs the computer to achieve objectives.

Besides requiring the available human expertise, the nature of the problem also affects whether an expert system is possible or not. First, the problem cannot be too difficult; we must be able to characterize it in some way. Secondly, the problem must not be poorly understood, else the developed knowledge base will be inaccurate and unreliable. Third, the problem must require cognitive skills rather than physical skills. Expert

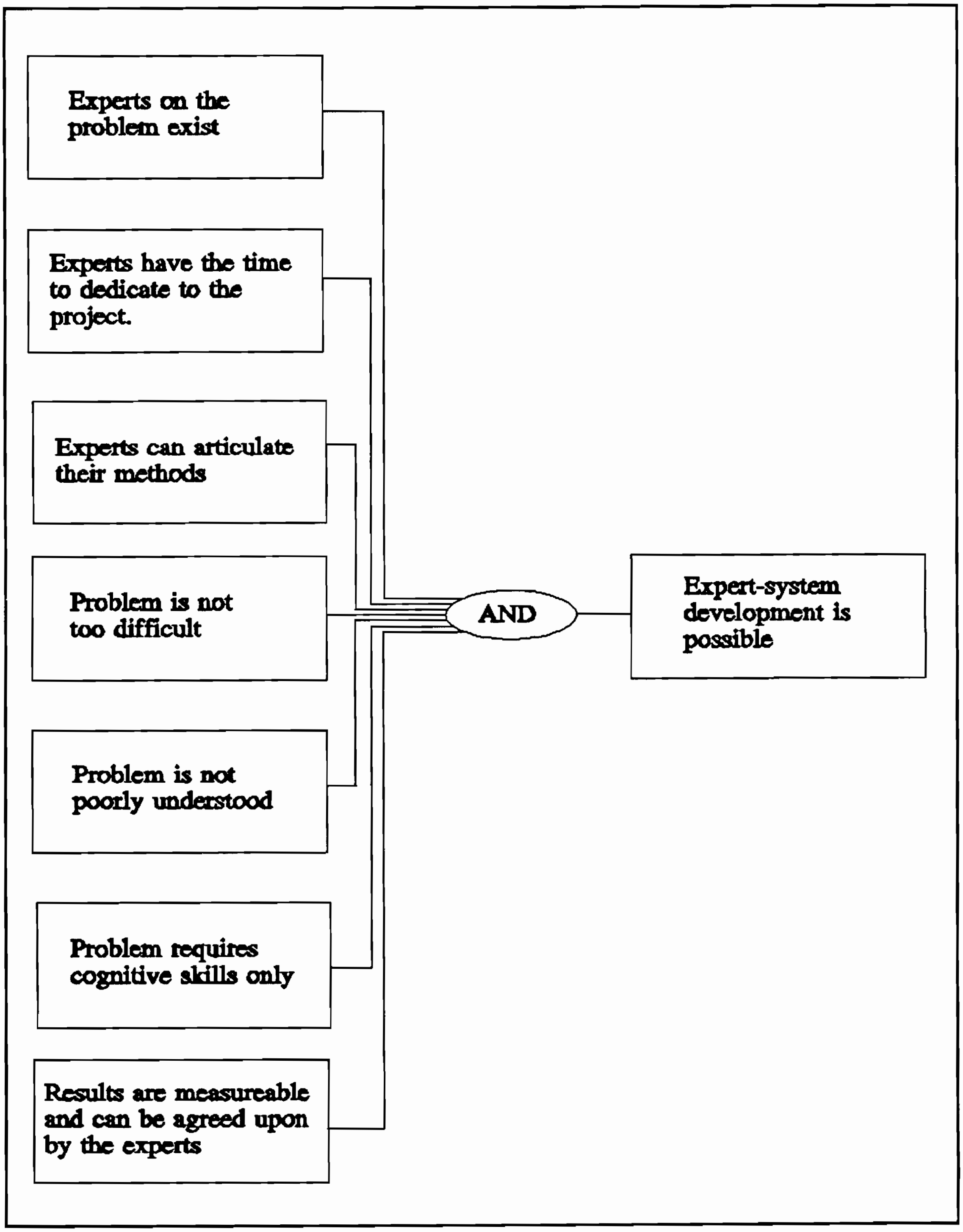

Experts on the problem exist
Experts have the time to dedicate to the project.
Experts can articulate their methods
Problem is not too difficult
Problem is not poorly understood
Problem requires cognitive skills only
Results are measureable and can be agreed upon by the experts
AND
Expert-system development is possible

Figure 16.1. Necessary requirements for expert-system development.

systems are computer-based tools that cannot perform any physical activity. Note however, that an expert system *can* be linked with something that does have physical skills. This happens frequently in robotics. Finally, the solution to the task at hand must be measurable and can be agreed upon by experts. This allows us to test and guide the expert-system development, and gives us a means of "quality control."

Note that in Figure 16.1, *all* of the defined criteria must be met for an expert system to be possible. If all the criteria are not fulfilled, we must seriously question if expert-system development is indeed possible. What is the most common pitfall in expert system development? Underestimating the complexity of the problem. Thus, of all the criteria specified in Figure 16.1, there are two guidelines that we must consider carefully:

(1) The problem must not be too difficult; and
(2) The problem must not be poorly understood.

When an expert system fails to meet its objectives, almost invariably one of these two criteria is violated. These criteria are violated because, at the beginning of the project, we tend to underestimate the complexity of the problem.

B. Determining the Appropriateness of Expert-System Development

Just because an expert system is possible, it may not be appropriate. Once we have identified an application that is possible, we need to determine if it is indeed appropriate to develop the system. This step can sometimes be the most difficult. Figure 16.2 shows the criteria on the appropriateness of an expert-system application.

Clearly, the problem must involve symbolic processing and the use of heuristics. If this is not the case, then mathematical modeling may be more appropriate. Another point to evaluate is how difficult the problem is. If the solution to the problem is too easy, another application, such as a sophisticated spreadsheet or database, may be better suited for the job.

Although the problem must be of sufficient complexity to utilize an expert system, it cannot be too complex. If the project is overly ambitious, it will get bogged down in the knowledge-engineering and prototyping stages. The knowledge base may "balloon" to an unmanageable size. The expert system may not run fast enough to be practical. People will lose interest, and the project may fail. Expectations can run high, so we must be careful in choosing a project with the right complexity.

For chemical engineering applications, how do we determine if an application is complex enough but not too complex? One rule of thumb we may use is that if it takes less two or three hours for an engineer to solve, the problem is too easy. If the problem takes more than two days, it is too difficult. Of course, this rule depends on the nature of the problem, and is a moving target. Computer-hardware improvements are

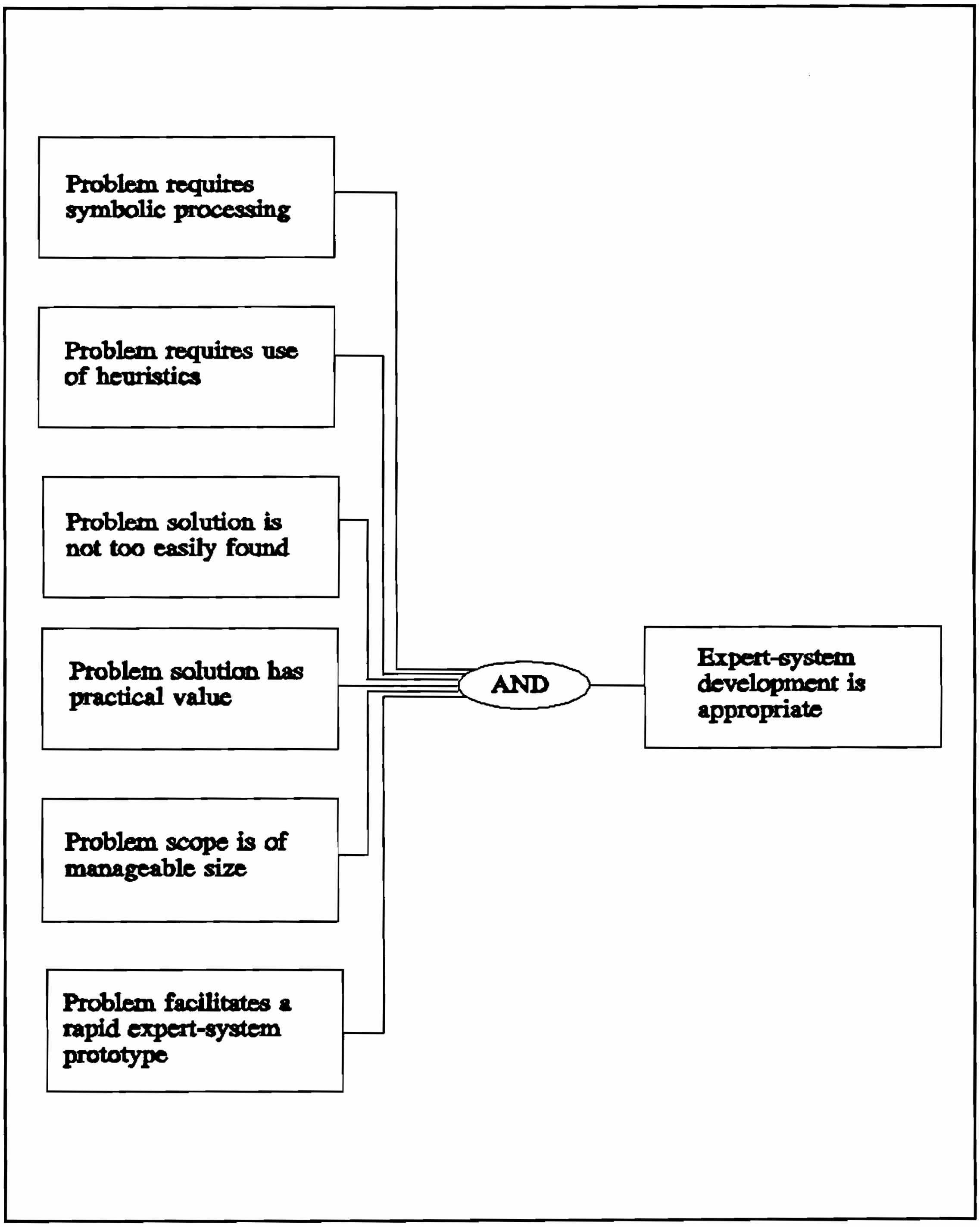

Problem requires symbolic processing
Problem requires use of heuristics
Problem solution is not too easily found
Problem solution has practical value
Problem scope is of manageable size
Problem facilitates a rapid expert-system prototype
AND
Expert-system development is appropriate

Figure 16.2. Evaluating the appropriateness of an expert-system application.

steadily converting previously impossible tasks into possible ones. In addition, if the application is to be used repeatedly (as in chemical process design), it may pay to develop an expert system for the application.

We strongly recommend that *expert systems be developed on problems that allow the system to be prototyped rapidly* (in one to six months depending on the nature of the project and the business). A rapid prototype has a number of advantages:

(1) Project momentum is maintained once the system is up and running.

(2) When developing the system, the following statement becomes acutely true:

> *You don't know what you don't know*

What does this statement mean? We may think that we have the knowledge correctly characterized, but invariably, holes and contradictions in the knowledge appear. System development is therefore an evolutionary process. The sooner we get a prototype running, the sooner we can test and modify it to perform accurately in real-world situations.

Early prototyping allows the expert system to have practical impact as soon as possible. Without rapid prototyping, we could be waiting a long time for any results.

If a problem does not facilitate prototyping, it may not be an appropriate expert-system application. Typical problems that fall into this category are those requiring "perfect" performance. Expert systems use knowledge to enable computers to perform like human experts. If we use a human expert to solve a problem, there is no guarantee that this expert will be able to find an acceptable solution. The same is true for expert systems. If "perfect" performance is required, an algorithmic approach may be more appropriate.

As an example, let us consider an expert system designed to handle space-shuttle docking. Space-shuttle docking requires perfection; if we mess up, we may lose a life or a $ 700 million satellite. Space-shuttle docking control is not a good expert-system application. It is impossible to prototype the system correctly on the earth, and therefore the initial execution would probably be unreliable. Unfortunately, there is no "tweaking" it to improve the system performance after the system is placed into service; it must perform perfectly immediately after installation or disaster can result.

C. Justifying Expert-System Development

Even though an expert-system approach is possible and appropriate, the project may not be justified. Expert systems require a significant investment of time and money. To justify an expert-system project, there must be a definite need with payback that outweighs the required investment. Figure 16.3 shows some of these needs that justify the expert-system development.

Some of the primary justifications of the expert-system development relate to solving the problem at hand. Expert systems are justified when:

(1) solution of a single problem has a high payoff (such as in oil exploration); or

(2) solution of a complex problem rapidly and repetitively enhances productivity (such as in chemical process design).

Other justifications also exist. If human expertise is going to be lost, is generally unavailable, or is needed in multiple locations, then the expert-system development is justified. An example is fault diagnosis, where during a critical malfunction in a plant, the process expert may not be available on-site.

Note that Figure 16.3 is not all-inclusive. The figure is meant to display some of the more common justifications for expert system development. A specific problem may have a unique justification of its own.

To summarize, we must make sure that an expert system is possible,

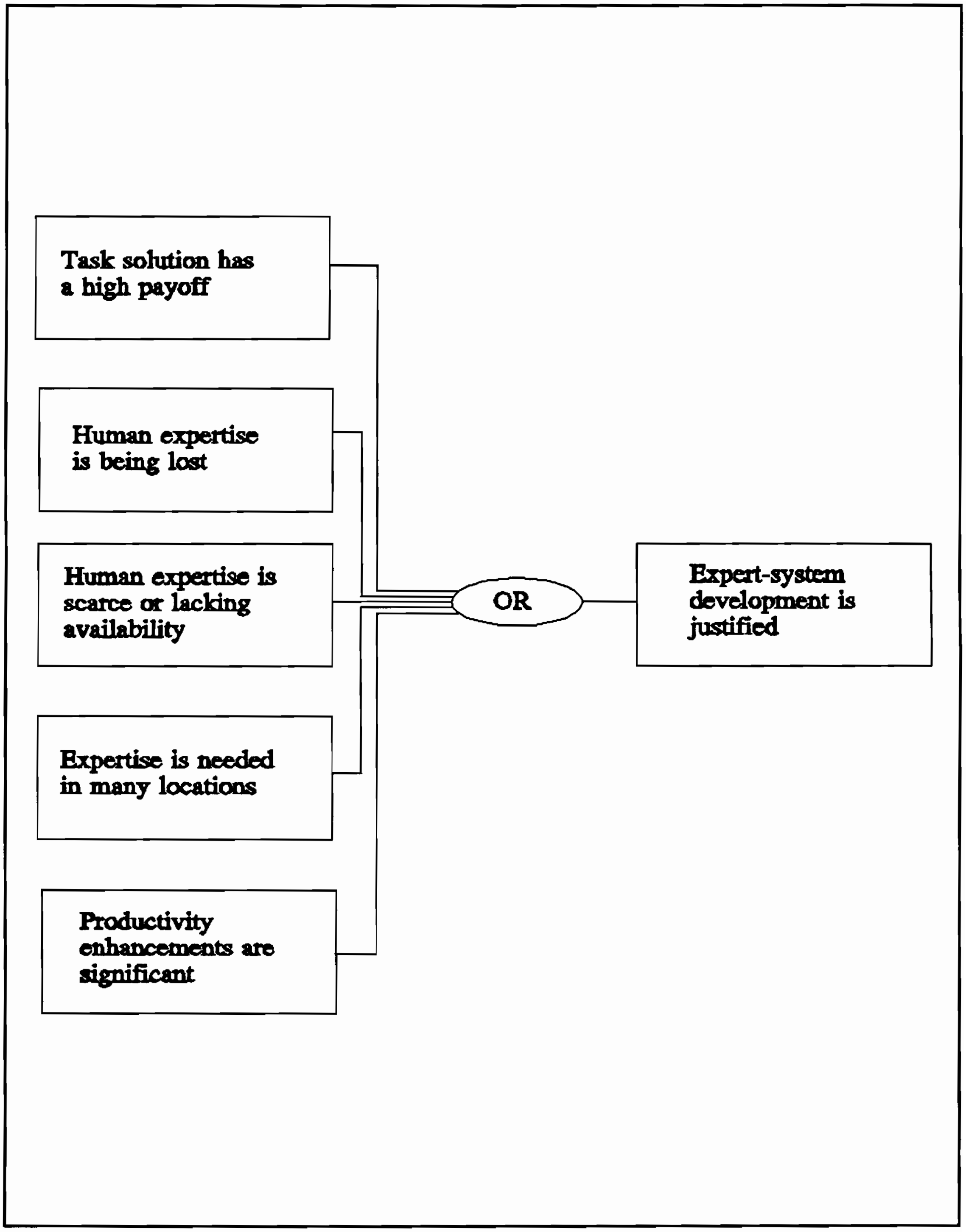

Task solution has a high payoff
Human expertise is being lost
Human expertise is scarce or lacking availability
Expertise is needed in many locations
Productivity enhancements are significant
OR
Expert-system development is justified

Figure 16.3. Justifications for expert-system development.

appropriate, and justified before starting a project. A number of classifications of problems have been found to be appropriate for the expert-system development, as mentioned in section 16.1. These application areas are:

- Fault diagnosis
- Process control
- Process design
- Planning and operations
- Modeling and simulation
- Product development

We discuss these applications in the following sections. Note that we *do not* present an exhaustive discussion of every published paper. We only highlight representative works. However, we have made an attempt to include essentially as many chemical engineering publications as possible up to late 1990 as references for further reading.

D. References and Further Reading

Harmon, P., and D. King, *Expert Systems: Artificial Intelligence in Business*, John Wiley & Sons, New York, NY (1985).

Hayes-Roth, F., D.A. Waterman, and D.B. Lenat, *Building Expert Systems*, pp.3-29, Addison Wesley, Reading, MA (1983).

Jackson, P., *Introduction to Expert Systems*, Addison-Wesley, Reading, MA (1986).

Pierce, T.H., and B. A. Hohne (Eds.), *Artificial Intelligence Applications in Chemistry*, ACS Symposium Series, 306, American Chemical Society, Washington, DC (1986).

Waterman, D.A., *A Guide to Expert Systems*, pp.135-161, Addison Wesley, Reading, MA (1986).

16.3 APPLICATIONS TO PROCESS-FAULT DIAGNOSIS

A. Introduction

If expert systems have a historical orientation, it would be towards diagnosis. The classic expert system MYCIN is a rule-based system for medical diagnosis. It takes in knowledge about the patient history, symptoms, and laboratory test results and then diagnoses the cause of infectious blood diseases. It also recommends treatments and remedies. Developed at Stanford University, MYCIN was the first expert system to perform at the level of, or better than, a human expert (in this case, a physician whose expertise is in infectious blood diseases).

In a similar fashion, knowledge-based systems have been used for diagnosis of chemical processes. Some of the key objectives in process-fault diagnosis are:

(1) infer system malfunctions from observable information;

(2) identify potential problems *before* they actually happen; and

(3) troubleshoot the process and recommend remedies.

Expert systems can be applied in two different ways to fault diagnosis:

(1) *on-line fault diagnosis*, where the expert system runs in real time; and

(2) *off-line fault diagnosis*, where real-time constraints do not exist.

1. On-Line Fault Diagnosis

Real-time fault diagnosis is a major challenge for process operators. When a problem occurs, it usually manifests itself in many ways. Multiple alarms may fire. Information overload can result, and it may be difficult for operators to deduce root causes of the fault. Also, low-probability malfunctions can occur where the operator has no prior experience. Plant engineers and experts may not be available to analyze the physical and chemical principles underlying anomalous behavior. When malfunctions occur, a timely response is absolutely essential. Both physical and financial disaster can result if the problem is not remedied quickly *and* correctly. A computer-based expert system assisting operators in fault diagnosis and recommending remedies can be very valuable.

While the need for on-line fault diagnosis using expert systems is great, this area is very challenging, and the consensus today is that expert systems are too slow to respond in real-time. Thus, the potential benefits of real-time systems have not been attained and must be considered an ideal. Nevertheless, a few real-time fault-diagnosis expert systems have been attempted. One is FALCON (Fault AnaLyzer CONsultant), which we discuss in section 16.3B.

2. Off-Line Fault Diagnosis

Off-line fault diagnosis does not have the constraint of real-time response, and therefore, is much easier to implement. There are many off-line fault-diagnosis systems in operation today. If a complex or chronic problem exists, an operator can go to the off-line expert system for guidance. For example, General Electric Company has a commercial, off-line fault-diagnosis system to aid in the repair of diesel locomotives used in railroads. A mechanic inputs symptoms that the locomotive is suffering, and the expert system alerts the mechanic for areas of concern and recommends corrective actions.

Virtually all commercial expert system implemented today are off-line. Consequently, these systems are restricted to problems where an in-depth analysis, rather than a rapid response, is in order.

B. FALCON

1. Background

FALCON (Fault AnaLyzer CONsultant) is a fault-diagnosis expert system for a commercial-scale chemical plant. FALCON officially began in 1984 as a joint development project between Du Pont, Foxboro, and The University of Delaware. It was funded at a total cost of approximately $ 1 million. Its primary goal was to assess the technical feasibility, develop, and implement an on-line system that could detect process faults and determine originating causes for a commercial chemical process.

The plant of choice for FALCON was Du Pont's adipic acid plant in Victoria, Texas. The plant was chosen because:

(1) it is sufficiently complex such that successful implementation would confirm the system's feasibility on a commercial scale;
(2) real operational hazards exist; performing fault diagnosis would have practical value for plant personnel;
(3) the process is well understood, so a knowledge base could be adequately constructed; and
(4) the process is relatively non-proprietary, a requirement since the project was developed in a university environment.

2. Qualitative Reasoning

In the initial development of FALCON, a "conventional" rule-based, qualitative knowledge representation was used. Since fault diagnosis is data-driven, FALCON was developed as a forward-chaining rule-based system. A causal model of the process was developed, i.e., rules were developed to deduce conclusions through cause-and-effect behavior observed in the process.

After much time and effort, it was concluded that the traditional rule-based approach was insufficient. The system was unreliable, and there were too many "holes" in the knowledge base. The heuristic rules developed were insufficient to reliably characterize the process.

Why was the rule-based approach insufficient? MYCIN, the first successful expert system to match or exceed human expertise, was developed as a rule-based system. Just like FALCON, MYCIN is a diagnostic system--MYCIN diagnoses infectious blood diseases. The similarities between MYCIN and FALCON are clear. Why then, was MYCIN a success, while this attempt with FALCON was not?

The answer lies in the characteristics of the problem. FALCON diagnoses a *dynamic* process, while MYCIN diagnoses a *static* one. For complex dynamic processes, it is very difficult to develop a knowledge base with adequate depth and accuracy.

FALCON lesson #1: For dynamic systems, it is very difficult to develop a qualitative knowledge base that has adequate depth and accuracy.

FALCON's original knowledge base had what is known as "shallow knowledge." Shallow knowledge knows the rules (i.e, heuristics), but knows nothing about the physical principles behind these rules. When dynamic situations arose where the knowledge had not been explicitly encoded into the knowledge base, FALCON became unreliable. Something beyond shallow knowledge was required if FALCON was to achieve its objectives.

Based on this learning, FALCON changed gears. In discussions with process engineers and operators, some interesting information was recognized as needing more attention. Engineers were quick to use both material and energy balances in their diagnostic procedure. These balances are *quantitative* analyses. The conclusion? Quantitative information was available and should be used. The remedy for FALCON? Successful process troubleshooting ought to include both qualitative and quantitative information.

FALCON lesson #2: A successful fault-diagnosis expert system will do both quantitative and qualitative analysis.

FALCON used a forward-chaining causal-inference strategy. If process conditions matched the conditions programmed into a rule, the rule would "fire." These rules develop a cause-and-effect inference chain, ultimately leading to the origin of the problem. The forward-chaining inference typically leads to numerous competing explanations as to the origin of the process problem. Resolution of these competing explanations is required. A hierarchical rule system was developed to enable FALCON to resolve conflicts.

Unfortunately, the hierarchical resolution system did not meet expectations. There was no guarantee that the system could handle all situations. When new situations arose, the system was unreliable. Clearly, a straight qualitative approach was not going to work; quantitative reasoning from first principles was required.

3. Quantitative Reasoning from First Principles

Since the quantitative information is readily available, we should be as opportunistic as possible and utilize it. FALCON's knowledge base was adjusted to include the following quantitative information (in the form of process models):

- *Material-balance equations*- written in terms of measured variables around desired control volumes.
- *Energy-balance equations*- also written in terms of measured variables around desired control volumes.
- *Empirical equations*- written to relate measured variables.
- *Controller equations*- written for PI controllers.
- *Valve-curve equations*- written to correlate measured flow to controller outputs.
- *Heat-transfer equations*- written to determine heat-transfer coefficients.

Instead of exclusively having arbitrary heuristics (which is what we see in shallow-knowledge systems), the inclusion of fundamental models from physical principles began to "deepen" FALCON's knowledge base. Shallow knowledge contains experiential, arbitrary knowledge about systems. Shallow systems know the heuristics, but do not understand the fundamental physical principles behind them. If a situation arises that is not explicitly encoded into the knowledge base, a shallow-knowledge system will fail.

In contrast to shallow-knowledge systems, there are "deep knowledge" systems. Deep-knowledge systems have the heuristics, but they also have the ability to reason from more fundamental principles (e.g., the conservation laws). This capability allows the systems to handle novel situations, enhancing both the accuracy and the robustness of the system. If the heuristics fail, reasoning at deeper levels of abstraction may still be able to do the job. With the inclusion of quantitative reasoning, FALCON was making the turn from a shallow- to a deep-knowledge system.

When a malfunction occurs, engineers are quick to hypothesize and test the accuracy and validity of the information being received from the control instrumentation. This important point was not addressed in FALCON's initial execution, but was realized later when quantitative reasoning was included in the system.

FALCON lesson #3: A successful fault-diagnosis expert system will be able to identify control-system malfunctions (e.g., sensor failure, valve saturation, and grossly inaccurate measurements).

Control-equipment failure is tested in FALCON using the quantitative equations described under "lesson #2." For example, valve-curve and controller equations are used to model the control-valve behavior. If the observed behavior deviates significantly from the equations, there is a control-instrumentation malfunction. When troubleshooting, process engineers frequently use this information to determine if any difficulties occur in the control equipment itself.

FALCON was placed through a rigorous "quality-control" analysis before it was implemented. Actual plant data were used. The advantage of using plant data is clear: FALCON can be tested in the true environment that it must perform in. Unfortunately, the plant data include only a small fraction of the range of possible faults that could potentially occur. Thus, testing with plant data is not enough-- in the future, the system may encounter a novel fault-situation, and not perform acceptably. Serious consequences could result.

Because a wide range of potential operating conditions was needed to test FALCON, a rigorous simulation of the process was developed. The simulation had over 200 differential equations and 1100 variables. The simulation was so large that even on a VAX-11/780, it could not run in real time. The program was run over the weekend, time histories were saved in files, and those files were fed into FALCON to test its fault-diagnosis ability.

FALCON lesson #4: For complex fault-diagnosis expert systems it is almost essential to have a good simulation of the process to help develop the knowledge-base.

Overall, the FALCON project was the first attempt to test the feasibility of and build a large expert system for fault diagnosis. It reached most, but not all of its objectives. The results were somewhat disappointing, given the initially high expectations. It took about 12 man-years of time to develop the system. In the development of the project, some challenges arose that were not entirely obvious at the project's inception. *Unexpected and unforeseen challenges that tend to "pop up" in testing are very common in the expert-system development.*

On-line fault diagnosis is more challenging than medical diagnosis. For this reason, the shallow-knowledge approach is acceptable for medical diagnosis in MYCIN, but is not acceptable for dynamic, on-line fault diagnosis. One key lesson from FALCON is that some form of deep knowledge is required for expert systems to properly diagnose process malfunctions.

C. CATDEX

CATDEX (Venkatasubramanian, 1988) stands for CATalytic cracking unit Diagnostic EXpert. It is simple a fault-diagnosis expert system for fluidized catalytic cracking (FCC) process. FCC units in oil refineries "crack" higher-molecular-weight gas oils into lower-molecular-weight materials such as gasoline and aviation fuel. These units are typically very profitable, since the demand for fuels is much higher than that for gas oils. Diagnosing and correcting problems quickly is essential in FCC units.

CATDEX is a very small prototype system, developed by K. Zilora in a few months as part of an AI course project at Columbia University. CATDEX identifies FCC process problems in three areas:

- excessive catalyst loss;
- poor catalyst circulation; and
- yield loss.

These problem areas do not cause immediate shutdowns. However, if not corrected, they result in reduced profits, poor process performance, imminent shutdown, and even irreparable equipment damage.

CATDEX is a backward-chaining rule-based system. It uses a depth-first search strategy, and is written in OPS5 (Brownston et. al., 1985). The language OPS5 is more suitable for forward chaining, so implementation of backward chaining in the program is a little awkward. Prolog, with built-in backward chaining and depth-first capabilities, would probably have been the better language choice. CATDEX has approximately 100 production rules.

CATDEX uses an AND-OR problem-decomposition strategy (section 9.4A) to locate the origins of problems. It uses "exact" reasoning, incorporating a "yes-no" approach. The system asks the engineer questions that can be answered "yes" or "no". Based on these answers, CATDEX recommends the probable source of the problem. Statistical reasoning, uncertainty, and inexact reasoning strategies are not used in CATDEX.

As mentioned previously, CATDEX attempts to isolate FCC problems associated with excessive catalyst loss, poor catalyst circulation, and yield loss. It identifies problems such as:

- cyclone damage
- changing particle size distribution
- standpipe level fluctuations
- partial bed defluidization
- thermal degradation of catalyst
- high regenerator velocity
- holes in reactor plenum
- pressure fluctuations
- catalyst attrition
- catalyst poisoning
- plugged lines
- change in feed quality

CATDEX also has an explanation facility. When the system asks a question, the user can respond **why**. This triggers the explanation mechanism. CATDEX will give a brief explanation of why the information is needed.

CATDEX is a good example of the broad and practical applications that expert systems can have in process engineering. Although its knowledge base is limited, CATDEX is also a good example of how quickly one can get a prototype system up and running.

D. BIOEXPERT

BIOEXPERT (LaPointe et al., 1989) is a fault-diagnosis expert system for wastewater treatment. Its focus is on failure detection and diagnosis in anaerobic wastewater-treatment plants. In the development of BIOEXPERT, emphasis was placed on overcoming some of the weaknesses of previous fault-diagnosis systems. Specifically, BIOEXPERT focused on delivering in four important areas:

- *Process generality-* using "deep knowledge" modeling to enable the system to handle general process situations.
- *Fault diversity-* identifying a minimum fault set that the system must be able to handle, i.e., the minimum collection of faults required for the system to operate correctly.
- *Reasoning transparency-* keeping information accessible such that the user can ask the system why and see the reasoning pathway.
- *Reliability and graceful degradation-* if the system fails to identify the fault, BIOEXPERT will give a summary of the different diagnostic steps to assist the engineer in determining causes.

BIOEXPERT does not do on-line fault diagnosis. Its knowledge base consists of:

- *Plant general knowledge-* encompasses the physical layout of the plant, including its structure, components' behavior, and components' functions.
- *Diagnostic knowledge-* includes diagnostic techniques used by process engineers, such as observing symptoms, generating

hypotheses, and testing these hypotheses versus available data.

The diagnostic section includes shallow knowledge, i.e., heuristic "if-then" rules that the system can use to quickly isolate common problems. If the shallow-knowledge approach fails, the system goes to a deeper level of abstraction. While no explicit numerical model exists for the biochemical process of wastewater treatment, BIOEXPERT does perform some deep-knowledge procedures such as the mass balance, qualitative relationships between biological parameters, and the energy balance in the cooling water. The overall structure of the knowledge base is shown in Figure 16.4.

BIOEXPERT is written in Prolog. It uses backward chaining and is a frame-based system. Frames can call other frames in an attempt to fill slots. In addition, each frame has access to diagnostic procedures. For example, the pump frame contains the following information in slots:

- pump switch and its state (on or off)
- float switch and its state (at the bottom or not at the bottom)
- pump motor and its state (correct or incorrect)
- level-control signal and its state (high, normal, and low)
- input flow rate, Q_{in} (numerical value)
- output flow rate Q_{out} (numerical value)
- hydraulic links
- electrical links

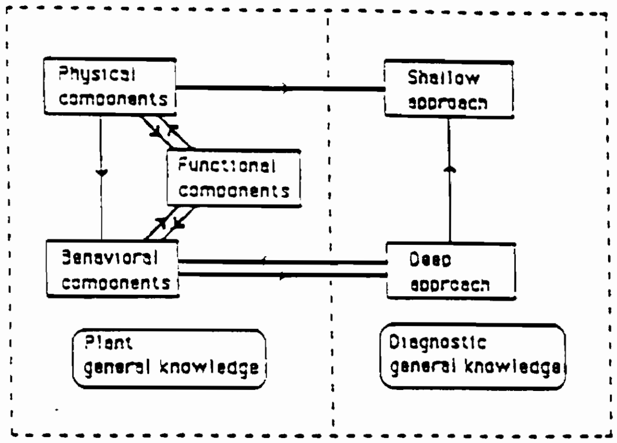


Figure 16.4. BIOEXPERT knowledge base.

- chemical links
- various procedures for filling slots and diagnosing problems.

If all slot values are in their proper ranges, BIOEXPERT reports that the pump is operating correctly. If a slot value is considered improper, a diagnostic search will be executed to determine why.

BIOEXPERT's overall structure consists of:

- *User interface*- facilitates data input and explanation.
- *Blackboard*- collects the facts fed in by the user, or deduced from

qualitative and quantitative analysis on the program.

- *Knowledge base*- develops the inference chain using backward chaining depth-first search with both plant and diagnostic knowledge.

The overall structure of BIOEXPERT is shown in Figure 16.5.

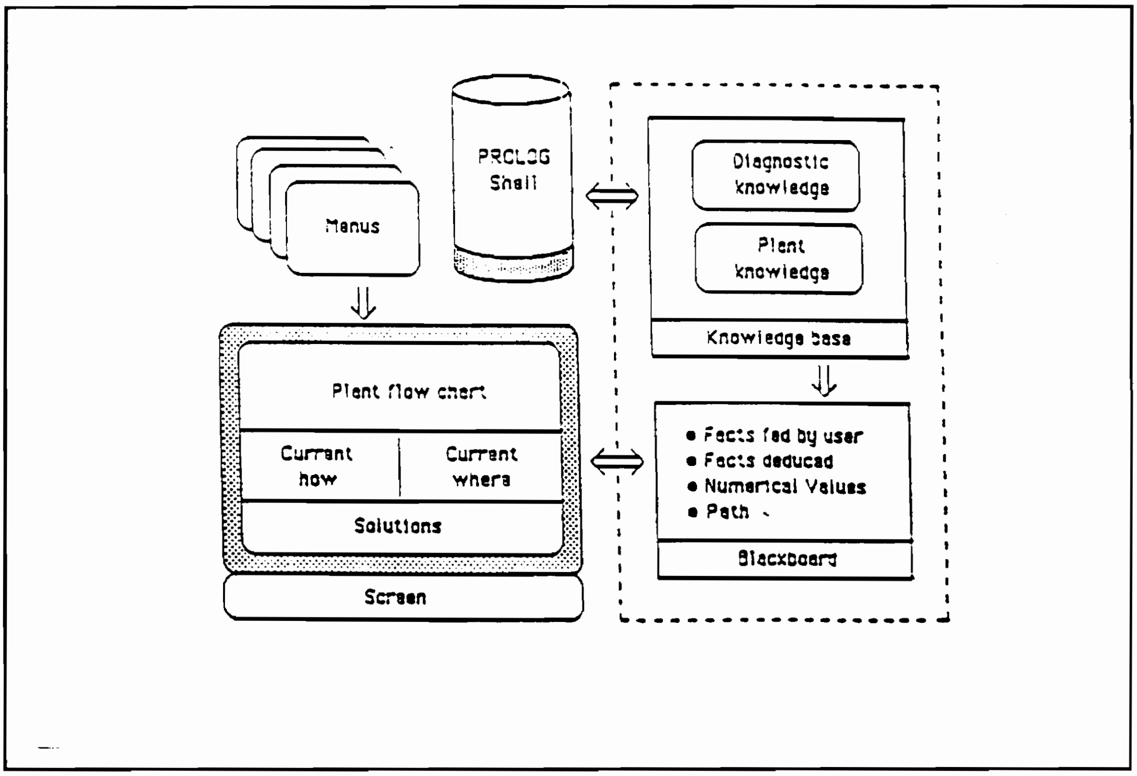


Figure 16.5. Overall structure of BIOEXPERT.

BIOEXPERT uses a *blackboard* (discussed in section 10.2H), a globally accessible database able to serve the entire system. A blackboard receives knowledge from many sources and disseminates this knowledge to many receivers. Communication with a blackboard can be done by numerous field experts, the expert-system inference engine, the expert-system user, and even plant instrumentation. With BIOEXPERT, facts deduced, user information, and results from qualitative and quantitative reasoning are placed on the blackboard.

Since BIOEXPERT does not do on-line fault diagnosis, it can use a menu system to facilitate data input. The user can enter a problem, and BIOEXPERT will respond with questions. Hypotheses are generated and tested. A cause-and-effect chain develops, which in the end, will hopefully lead to the original source of the problem. If BIOEXPERT fails to find an explanation, it reports its reasoning to the user.

E. Problem-Solving Approaches in Fault Diagnosis

We have a feel for, and have seen some examples of, chemical-process fault diagnosis. Because chemical-process fault diagnosis has gotten so much attention, some systematic techniques have been developed to aid in problem-solving. Let us now discuss some of those approaches.

1. Rule-Based Approaches

A rule-based approach is the simplest and most direct form of fault diagnosis. There are two separate rule-based problem-solving strategies: the ***evidential*** approach, and the *causal* approach. The evidential approach is based on evident, experiential knowledge (i.e., compiled shallow knowledge). The causal approach is based on more in-depth models, possibly derived from first principles (i.e., deep knowledge). The causal approach, however, typically uses either qualitative modeling or semi-quantitative modeling. If we desire to use more mathematically rigorous modeling, we typically do not use rule-based approaches. Table 16.1 compares the evidential and causal approaches.

Table 16.1. The evidential versus causal approach to fault diagnosis.

DESCRIPTION	EVIDENTIAL	CAUSAL
Knowledge base	Experiential, shallow, and compiled surface knowledge	Deep-level, possibly using first principles
Knowledge Structure	Hypotheses (causal origins) *explicitly* associated with evidence (symptoms)	Generation of hypotheses using a model to determine the source of faults
	Usually grouped in fault trees, networks, or frames	Classified according to the model used to detect fault
Inference	Usually backward chaining only	Both forward and backward chaining
Reasoning	Usually driven by heuristics	Can be driven by simulations

Which rule-based approach is better, the evidential or causal approach? It depends on the problem complexity. If it is a well-understood process or one that requires no modeling, the evidential approach may be more appropriate (e.g., fault diagnosis of discreet processes, such as assembly and fabrication operations). If the problem is more complex and requires modeling, the causal approach may be more appropriate. Table 16.2 compares the advantages and disadvantages of each approach.

Table 16.2. Advantages and disadvantages of evidential and causal approaches to fault diagnosis.

DESCRIPTION	EVIDENTIAL	CAUSAL
Implementation	Relatively easy to implement	Difficult to implement
Program speed	Fast and efficient	Slow and plodding
Enumeration of hypothesis	Hypotheses must be enumerated *a priori*	Need not enumerate at all
Response in novel situations	Fails when unanticipated or concurrent symptoms occur	Can accommodate unforeseen symptoms and possibilities
Program maintenance	Knowledge base must be rewritten or adjusted if process changes are made	Can accommodate changes with little or no adjustment of knowledge base

2. Model-Based Approaches

Model-based expert systems incorporate in-depth knowledge from first principles. These models are *quantitative*, in contrast to causal rule-based systems that use qualitative or semi-quantitative modeling. Note, however, that the demarcation line between a causal rule-based approach and a model-based approach can be unclear -- that line relates to the depth and numerical nature of the models used.

To improve efficiency of model-based approaches and allow them to integrate better with plant hardware, some more recent model-based systems have been written in C rather than Prolog or LISP. Also, some model-based approaches use a device-centered, object-oriented knowledge representation (Ungar, 1987).

Most model-based approaches use a *generate-and-test* or a *plan-generate-test* technique for problem-solving (discussed in section 11.3C). The generate-and-test approach uses a "scorched-earth policy" that generates *all* potential solutions to the problem before passing the information to the tester. Thus, it is best-suited for a problem where there is a relatively low number of potential faults. The plan-generate-test approach is used on problems where there are many potential faults. Instead of inefficiently generating all potential solutions, the planner uses constraint analysis up-front to prune the search.

In-depth model-based expert systems that rely extensively on first principles are just beginning to develop. Grantham and Ungar (1990) give an example of how to structure the knowledge in such a system. In principle, these model-based approaches offer a number of advantages over traditional rule-based approaches:

- Model-based approaches overcome the brittleness and narrow range of applicability of the rule-based approach.
- Model-based approaches are more robust and better able to handle unique situations.
- Model-based approaches are easily maintained, and do not need major adjustment when the process is altered.

F. Challenges in Process-Fault Diagnosis

As we have seen in the above case studies, fault diagnosis can be a great aid to engineers and operators attempting to troubleshoot processes. But also from the previous summaries we see a number of challenges in the area. If we can overcome these challenges, the utility of fault-diagnosis expert systems will rise dramatically. Kramer (1986) suggests some of these challenges unique to chemical-process fault diagnosis by contrasting the challenges unique to the chemical-process fault-diagnosis problem with the "conventional" diagnosis problem where shallow knowledge is acceptable (an example of the conventional problem is medical diagnosis with the expert system MYCIN, which uses shallow knowledge):

- *Lack of Experts-* in medical diagnosis, there is a large and detailed diagnostic and treatment database. In process diagnosis, there are few experts. In addition, some malfunctions may never have been experienced before. If the plant is new, there may be no experiential knowledge whatsoever.

- *Level of Understanding-* medical diagnosis has an inherent amount of uncertainty due the complexity of biological systems. In process diagnosis, engineering fundamentals (i.e., mass, energy, and momentum conservation, thermodynamics, kinetics, etc.) can be applied for additional guidance. This quantitative knowledge should be used, allowing us to reason with "deeper" levels of knowledge.

- *Lack of Generality*- medical diagnosis treats patients who are physiologically equivalent. In process diagnosis, no two chemical plants are the same. Therefore, the knowledge base for each application needs to be developed from scratch. This is an enormous investment in time and money that could be avoided if generality was attainable.

- *Reliability and Completeness*- testing an expert system performing medical diagnosis is straightforward, since symptoms typically match underlying causes. In process diagnosis, faults occur infrequently. In addition, faults cannot be introduced for the sole purpose of testing the system. For this reason, the shallow-knowledge approach cannot be expected to operate reliably the first time that faults are encountered on-line.

On the more pragmatic side, *fault-diagnosis expert systems for chemical plants do not have the economies of scale that medical diagnosis has.* A medical diagnosis expert system can be distributed to physicians everywhere for use. In contrast, a fault-diagnosis system for a chemical plant *must be built for that specific plant only.* Consequently, development costs are high, and indeed, the expert system may not be justified if the problem-fit is not good.

In addition, compared to medical diagnosis, chemical fault-diagnosis expert systems *need more maintenance*. As modifications to the plant and equipment occur, the knowledge base of the expert system must reflect these changes, or the system could become unreliable. Continual system maintenance is essential.

G. Future Directions

The future of chemical process fault-diagnosis expert systems will have a number of directions. In the software area, the focus is on:

- Combining qualitative and quantitative process knowledge to improve the efficiency of systems;
- Developing tools such as dynamic simulators to test system reliability;
- Automating and generalizing the knowledge-engineering process so that the knowledge base does not need to be developed from scratch for each chemical plant; and
- Implementing learning mechanisms into the expert system to automatically acquire new heuristics when a unique situation is encountered.
- Improving user-interface tools: it is crystal clear that a good

user interface is essential to the success of the system.

- Reducing development costs: it took FALCON twelve man-years of development time. An equivalent system today could be implemented in possibly 3 man-years. A decrease in development costs opens up fault diagnosis to a much wider range of applications.

While the primary emphasis will probably be on the software side, there will also be improvements in hardware. Some areas of focus are:

- Developing controllers that can process qualitative and quantitative information;
- Developing hybrid computer machines that can combine symbolic and numerical processing for on-line diagnosis; and
- Developing controllers and systems that are able to diagnose the control equipment itself.

Finally, we would be remiss in discussing future trends in fault diagnosis if we did not mention *Artificial Neural Networks* (ANNs). Many researchers are dropping symbolic computing approaches and using numerical, connectionist approaches to fault diagnosis using ANNs. We discuss ANNs in-depth in chapter 17.

H. References and Further Reading

Himmelblau (1983) is a good book on the general topic of fault detection and diagnosis in the chemical process industries. A general AI reference is *Artificial Intelligence in Process Engineering*, M.L. Mavrovouniotis, Editor, Academic Press, San Diego, CA (1990). Venkatasubramanian (1988) presents a detailed report for the development of a fault-diagnosis expert system for a fluidized catalytic cracking (FCC) unit. Other publications of particular interest are Calandranis et. al. (1990), Dhjurati et. al. (1987), Kramer and Palowitch (1987), Naito et. al. (1987), and Ramesh et. al. (1989).

Andow, P.K., "Fault Diagnosis Using Intelligent Knowledge-Based Systems," *Inst. Chem. Eng. Symp. Ser.*, 92 (Process Syst. Eng., PSE '85), 145 (1985).

Andow, P.K., "An Integrated Expert System for Real-Time Fault Diagnosis," *AIChE Spring National Meeting*, Orlando, FL, March (1990).

Brownston, L., R. Farrel, E. Kant, and N. Martin, *Programming Expert Systems with OPS5: An Introduction to Rule-Based Expert Systems*, Addison-Wesley, Reading, MA (1985).

Calandranis, J., G. Stephanopoulos, S. Nunokawa, "DiAD-Kit/BOILER: On-Line Performance Monitoring and Diagnosis," *Chemical Engineering Progress*, pp. 60-68, January (1990).

Chang, C.C. and C.C. Yu, "On-Line Fault Diagnosis Using the Signed Directed Graph," *Ind. Eng. Chem. Res.*, **29**, 1290 (1990).

Chen, M., "Fault Diagnostic System for Boiling-Water Reactor: A Signed-Directed Graph Approach," M.S. Thesis, National Taiwan Institute of Technology, Taipei, Republic of China, July (1990).

Chester, D.L., D.E. Lamb, and P.S. Dhurjati, "Rule-Based Computer Alarm Analysis in Chemical Process Plants," *Proc. Microdelcon*, **1**, 22 (1984).

Chester, D.L., D.E. Lamb, and P.S. Dhurjati, "An Expert-System Approach to On-Line Alarm Analysis in Power and Process Plants," *Comput. Eng.*, **1**, 345 (1984).

Choi, K. Y., J.O. Yong, and S.H. Chang, "The Manipulation of Time-Varying Dynamic Variables Using the Rule Modification Method and Performance-Index in NPP (Nuclear Power Plant) Accident Diagnostic Expert Systems," *IEEE Trans. Nucl. Sci.*, **35** (5), 1121 (1988).

Chowdhury, J.,"Troubleshooting Comes On-Line in the CPI", *Chem. Eng.*, Oct. 13, 1986, pp.14-19.

Chung, D.T. and M. Modarres, "A Method of Fault Diagnosis: Presentation of a Deep-Knowledge System," *AIChE Annual Meeting*, New York, NY, November (1987).

Cooper, D.J., and A.M. Lalonde, "Process Behavior Diagnostics and Adaptive Process Control," *Comput. Chem. Eng.*, **14**, 541 (1990).

D'Ambrosio, A., "Modeling Real-World Processes: Deep and Shallow Knowledge Integrated with Approximate Reasoning in a Diagnostic Expert System," in *Artificial Intelligence in Process Engineering*, pp. 189-220, M. Mavrovouniotis, Editor, Academic Press, San Diego, CA (1990).

Dhurjati, P.S., D.E. Lamb, D.L. Chester, "Experience in the Development of an Expert System for Fault Diagnosis in a Commercial-Scale Chemical Process," in *Foundations of Computer-Aided Process Operations*, pp. 589-625, Elsevier Publishers, New York, NY (1987).

Finch, F.E. and M.A. Kramer, "Narrowing Diagnostic Focus Using Functional Decomposition," *AIChE J.*, **34**, 25 (1988).

Grantham, S.D. and L.H. Ungar, "A First Principles Approach to Automated Troubleshooting of Chemical Plants," *Comput. Chem. Eng.*, **14**, 783 (1990).

Hart, P., "Directions for AI in the Eighties," *SIGART Newslet.*, 79 (1982).

Henley, E.J., and Kumamoto, K., "Designing for Reliability and Safety Control," *Inst. Chem. Eng. Symp. Ser.*, **92** (Process Syst. Eng., PSE '85), 651 (1985).

Henley, E.J., and Kumamoto, K., *Designing for Reliability and Safety Control*, pp. 195-240, Prentice-Hall, Englewood Cliffs, N.J. (1985).

Himmelblau, D.M., *Fault Detection and Diagnosis in Chemical and Petrochemical Processes*, Elsevier Scientific Publishing Company, New York, NY (1983).

Hoskins, J.C. and D.M. Himmelblau, "Fault Detection and Diagnosis Using Artificial Neural Networks," in *Artificial Intelligence in Process Engineering*, pp. 123-160, M. Mavrovouniotis, Editor, Academic Press, San Diego, CA (1990).

Huang, Y.W., S. Shenoi, A.P. Mathews, F.S. Lai, and L.T. Fan, "Fault Diagnosis of Hazardous Waste Incineration Facilities Using a Fuzzy Expert System," *Expert Systems in Civil Engineering Symposium*, Seattle, WA (1986).

Janusz, M.E., V. Venkatasubramanian, "Automatic Generation of Qualitative Descriptions of Process Trends for Fault Detection and Diagnosis," *AIChE Annual Meeting*, Chicago, IL, November (1990).

Kramer, M.A., and B.L. Palowitch, Jr., "Expert System and Knowledge-Based Approaches to Process Malfunction Diagnosis," *AIChE Annual Meeting*, Chicago, IL, November (1985).

Kramer, M.A., "Integration of Heuristic and Model-Based Inference in Chemical Plants," *IFAC Workshop on Fault Detection and Safety*, Kyoto, Japan (1986).

Kramer, M.A., "Malfunction Diagnosis Using Quantitative Models with Non-Boolean Reasoning in Expert Systems," *AIChE J.*, **33**, 130 (1987).

Kramer, M.A., and B.L. Palowitch, Jr., "A Rule-Based Approach to Fault Diagnosis Using the Signed Directed Graph," *AIChE J.*, **33**, 1067 (1987).

Kumamoto, H., K. Ikenchi, K. Inoue, and E.J. Henley, "Application of Expert System Techniques to Fault Diagnosis," *Chem. Eng. J.*, **29**, 1 (1984).

Lai, L. and R. Govind, "Development of a Process Diagnosis Scheme Using AI

Techniques," *AIChE Symp. Ser.*, **85** (267, Process Sens. Diagn.), 30 (1989)

Lapointe, J., B. Marcos, M. Veillette, and G. Laflamme, "BIOEXPERT - An Expert System for Wastewater Treatment Process Diagnosis," *Comput. Chem. Eng.*, **13**, 619 (1989).

Lee, C., "Fault Diagnosis Based on Qualitative and Quantitative Process Knowledge," M.S. Thesis, National Taiwan Institute of Technology, Taipei, Republic of China, July (1990).

Malin, J.T., B.D. Basham, and R.A. Harris, "Use of Qualitative Models in Discrete Event Simulation to Analyze Malfunctions in Processing Systems," in *Artificial Intelligence in Process Engineering*, pp. 37-78, M. Mavrovouniotis, Editor, Academic Press, San Diego, CA (1990).

McDowell, J.K. and J.F. Davis, "Integrating Deep Reasoning and Compiled Reasoning for the Resolution of Interacting Malfunctions in Chemical Process Diagnosis," *AIChE National Meeting*, Chicago, IL (1990).

Mooney, D.L., D. Chester, P. Dhurjati, and D. Lamb, "Design and Operation of the FALCON Interface," *Adv. in Instrum.*, **43**, 747 (1988).

Morales, E., and H. Garcia, "A Modular Approach to Multiple Fault

Diagnosis," in *Artificial Intelligence in Process Engineering*, pp. 161-185, M. Mavrovouniotis, Editor, Academic Press, San Diego, CA (1990).

Myers, D.R., J.F. Davis, and C.E. Hurley II, "An Expert System for Diagnosis of a Sequential, PLC-Controlled Operation," in *Artificial Intelligence in Process Engineering*, pp. 81-120, M. Mavrovouniotis, Editor, Academic Press, San Diego, CA (1990).

Naito, N., A. Sakuma, K. Shigeno, and N. Mori, "A Real-Time Expert System for Nuclear Power Plant Failure Diagnosis and Operational Guide," *Nucl. Technol.*, 79, 284 (1987).

Palowitch, B.L., "Fault Diagnosis of Process Plants Using Causal Models," PhD dissertation, Massachusetts Institute of Technology, Cambridge, MA (1987).

Petti, T.F., J. Klein, and P.S. Dhurjati, "Diagnostic Model Processor: Using Deep Knowledge for Process Fault Diagnosis," *AIChE J.*. 36 (4), 565 (1990). See also: Letters to the Editor concerning this paper, by M.A. Kramer, T. Petti, and J. Klein, appearing in *AIChE J.*, 36, 1121 (1990).

Rajaram, N.S., and H. Tsehaie, "Role of Expert Systems in Disaster

Prevention," *Adv. in Instrum.*, 40, 1471 (1985).

Ramesh, T.S., S.K. Shum, and J.F. Davis, "A Structured Framework For Efficient Problem Solving in Diagnostic Expert Systems," *Comput. Chem. Eng.*, 12, 891 (1988).

Ramesh, T.S., J.F. Davis, and G.M. Schwenzer, "CATCRACKER: An Expert System for Process and Malfunction Diagnosis in Fluid Catalytic Cracking Units," *AIChE National Meeting*, San Francisco, CA, November (1989).

Rich, S.H. and V. Venkatasubramanian, "Model-based Diagnostic Expert Systems for Chemical Process Plants," *Comput. Chem. Eng.*, 11, 357 (1987).

Rich, S.H. and V. Venkatasubramanian, "Failure-Driven Learning in Expert Systems for Process Fault Diagnosis," *AIChE Annual Meeting*, New York, NY, November (1987).

Rich, S.H. and V. Venkatasubramanian, "Causality-Based Failure-Driven Learning in Diagnostic Expert Systems," *AIChE J.*, 35, 943 (1989).

Rich, S.H., "Model-Based Reasoning in Diagnostic Expert Systems for Chemical Process Plants," PhD Dissertation, Columbia University, New

York, NY (1988).

Rowan, D.A., "On-Line Fault Diagnosis: Initial Success and Future Directions," *Adv. in Instrum.*, **42**, 1211 (1987).

Rowan, D.A., "On-Line Fault Diagnosis: Initial Success and Future Directions," *Proc. of the Amer. Control. Conf.*, pp. 1211-1218, Minneapolis, MN (1987).

Rowan, D.A., "AI Enhances On-Line Fault Diagnosis," *InTech*, **35**, 52 (1988).

Shirley, R.S. and D.A. Fortin, "Status Report: An Expert System to Aid Process Control," *Adv. in Instrum.*, **40**, 1463 (1985).

Shirley, R.S., "Status Report 2: An Expert System to Aid Process Control," *Proc. of the TAPPI Engineering Conference*, pp. 425-430 (1986).

Shirley, R.S., "Some Lessons Learned Using Expert Systems for Process Control," *Proc. of the Amer. Control. Conf.*, pp. 1342-1346, Minneapolis, MN (1987).

Shum, S.K., J.F. Davis, W.F. Punch, and B. Chandrasekaran, "An Expert System Approach to Malfunction Diagnosis in Chemical Plants," *Comput. Chem. Eng.*, 12, 27 (1988).

Ungar, L.H., "Model-Based Expert Systems for Diagnosis and Control," *MIT Expert Systems Short Course*, Lecture 12, Cambridge MA (1987).

Ungar, L.H., B.A. Powell, and S.N. Kamens, "Adaptive Networks for Fault Diagnosis and Process Control," *Comput. Chem. Eng.*, 14, 561 (1990).

Venkatasubramanian, V., "Rule-Based Systems for Diagnosis," *MIT Expert Systems Short Course*, Lecture 11, Cambridge MA (1987).

Venkatasubramanian, V., "CATDEX: An Expert System for Diagnosing a Fluidized Catalytic Cracking Unit," *Knowledge-Based System in Chemical Engineering, Vol. I*, CACHE Case Study Series, CACHE Corp., Austin, TX and Elsevier, New York, NY (1988).

Venkatasubramanian, V. and S.H. Rich, "An Object-Oriented Two-Tier Architecture for Integrating Compiled and Deep-Level Knowledge for Process Diagnosis," *Comput. Chem. Eng.*, 12, 903 (1988).

Waters, A., and J.W. Ponton, "Qualitative Simulation and Fault Propagation in Process Plants," *Chem. Eng. Res. Des.*, 67, 407 (1989).

16.4 APPLICATIONS TO PROCESS CONTROL

A. Introduction

Expert systems are useful for enhancing process control. They can be used for process optimization, process management, trend analysis, alarm processing, control-system design, and adaptive control. Process control is the second-most popular area for expert-system applications in chemical engineering (fault diagnosis is first). Most controller manufacturers have on-going research and development projects, attempting to mesh both numerical and symbolic processing. The potential payoff here is very high, but the area is very challenging.

In alarm analysis, the function of an expert system is to properly resolve difficulties during a plant upset. In a major upset of a complex plant, anywhere from 10-800 alarms can go off in one to two minutes. Frequently, these alarms do not provide a clear picture of the problem. Expert systems can be used to prevent "information overload" during an upset. They can relate alarms with process data and recommend actions. Alarm analysis is closely related to, and is frequently supplemented with, built-in process fault-diagnosis capability.

Expert control can be used to enhance classical controller performance (Tzouanas et. al., 1988). For instance, an expert system could be used to properly identify sensor failure, valve saturation, and process constraints. This is particularly useful for multivariable controllers that rely upon an accurate model of the process to function properly.

Trend analysis has been used from both control and operation standpoints. For instance, yield analyses, temperature profiles, and additional process data can be used to recommend shutting down a reactor for maintenance or a catalyst regeneration. In addition, based on process trends, an expert system can recommend changes in controller and process operating parameters to enhance process performance.

A growing application of expert systems is in control-system design (Shinskey, 1986; Birkby, 1988). Rather than place the expert system into a control loop itself, we use expert system techniques to design the best PID feedback control-loop system for a process. Thus, this expert system could be viewed as a productivity enhancement tool -- process engineers who are not experts in designing control systems can access the expert system and develop and optimal control strategy very quickly.

Another area (perhaps the most important) that expert systems have found use in chemical process engineering is in *adaptive control*. Adaptive control attempts to adjust controller tuning parameters in response to process dynamics. For example, an expert system may act as an "umbrella" for a traditional controller, and use qualitative as well as quantitative information to periodically adjust the controller to maintain optimal performance.

B. Applications

Perhaps the best known application of expert systems to process control is EXACT (EXpert for Adaptive Controller Tuning), a rule-based, adaptive controller from Foxboro (Kraus and Myron, 1984). EXACT is an adaptive PID controller with a "supervisory set" of logical conditions. Under a set of process conditions and constraints, EXACT adjusts PID controller parameters in an effort to improve controller performance. EXACT, therefore, is in many ways "self-tuning." A self-tuning controller can adjust its own tuning parameters, usually on-line, based on changes in process conditions or dynamics. We discuss self-tuning in more depth in section 17.3A. EXACT is the first commercial controller to mesh expert system techniques with process control, and will probably be considered a landmark application.

EXACT uses heuristics to tune a PID feedback control loop. No process modeling is required. EXACT was first developed in the form of an expert system (off-line). It was then "converted" to an on-line program. Programs in Prolog and LISP generally run too slowly to be implemented in real-time. Therefore, to facilitate a rapid-response and on-line implementation, EXACT was written in assembly language.

EXACT operates using principles of pattern recognition. The closed loop is perturbed, the pattern of the response is recorded, and then this response pattern is compared with the desired response. Ziegler-Nichols (1942) tuning criteria are used as the desired response.

Self-tuning is implemented only when the response exceeds a nominal noise threshold. PID settings remain fixed, but the controller begins a sequence of storing process data for the purpose of recalculating tuning parameters. For example, let us consider how EXACT responds to an overdamped process. In response to a disturbance, an overdamped curve typically lacks successive damped oscillations seen in lightly damped systems. Thus, if EXACT detects no oscillations in the process, EXACT assumes the process is overdamped, and increases the proportional band accordingly.

EXACT has a number of advantages; two in particular are its flexibility and robustness. These features especially stand out when EXACT is compared to other knowledge-based control approaches. EXACT accommodates noise, and allows the user to limit available ranges of PID settings. We may also "pretune" the controller to get PID parameters within a proximity of the optimum, and then allow the controller to find the optimal settings itself. In addition, the algorithm does not compromise PID settings that are already optimal. The algorithm will not destabilize the loop, and if it is unable to stabilize the loop due to some inherent feature of the process, the algorithm will accept some nominal cyclic performance.

Another commercial expert-system application was done by LISP Machines, Inc. (LMI). LMI initiated project PICON (Process Intelligent CONtrol) in 1983 (Moore et. al., 1986). The goal of the project was to develop an on-line process control machine that could perform both quantitative processing (from process data) and qualitative reasoning (from expert knowledge). Importantly, this project also had some other key objectives:

(1) *Provide real-time control*- conventional expert systems, when applied to process control, are inadequate for real-time control. The size of the knowledge base is enormous, and these systems dutifully and exhaustively search the entire knowledge base to draw conclusions. The result is that they run too slow to perform real-time control.

(2) *Provide reliability and consistency*- control situations may arise that have never been encountered before, resulting in unreliable and inconsistent control. Unfortunately, there is no way to generalize and guarantee consistency and reliability of the knowledge base. Therefore, the developed machine must provide tools for the control engineer to evaluate the system.

(3) *Provide rapid knowledge acquisition*- knowledge acquisition, i.e., obtaining the knowledge and programming it into the computer so it reasons properly, is the critical obstacle to getting an expert system up and running. Typically, the AI programmers are far away from the plant experts; communication is poor. It can take so long to make progress through this stage that people become demoralized and disinterested in continuing. The developed machine must provide both the hardware and the software to provide rapid

knowledge acquisition.

A prototype LISP Machine design was tested in 1985 at a Texaco plant in Port Arthur, Texas. It was used in distillation control. The main machine had two processors running in parallel. One processor performed LISP inference, while the other was used for real-time data acquisition and control. A distributed control system was used. In addition to the Texaco installation, testing of LMI equipment has been done at Exxon (Bayway, New Jersey), Johnson Controls (pilot plant), Du Pont, Eastman Kodak, as well as others. Applications such as these are expected to continue and grow in the future.

Since the FALCON project, it has been accepted that quantitative, model-based reasoning is required if on-line process control is to work properly. In the PICON development, Moore and Kramer (1986) indicated that the model-based approach using deep knowledge was more appropriate than the heuristic approach.

There have been additional applications of expert systems to process control. Shinskey (1986) introduced an expert system that designs a control configuration. The goal of the system is to give the optimal control configuration for a multivariable process (in this case, distillation).

Although still in the basic research stage, Tzouanas et al. (1990 a,b) from Lehigh University have done research on a key challenge facing the process control area: Expert Multivariable Control (EMC). Most controllers used in industry are single-input and single-output (SISO). Multiple-input and multiple-output (MIMO) controllers have only found limited use. The main reason the MIMO controller has not caught on is that it must rely on an accurate process model to relate the variables.

Tzouanas et. al. (1990) have attempted to overcome the difficulties of MIMO controllers by using an expert system to relate variables in the MIMO controller. For instance, an expert system could be used to detect valve saturation or sensor failure, thereby improving the controller performance. Tzouanas et. al. introduce an expert-system development methodology that is process-independent, i.e., the same methodology can be used on any process. Unfortunately, as is the case with almost all forms of expert systems applied to control, the actual knowledge base is process-specific. A whole new knowledge base needs to be developed from scratch for each process application of EMC.

Developing a whole new knowledge base for each application is a tedious and expensive endeavor. This difficulty is the primary factor holding back EMC. Controller manufacturers are working on minimizing this investment through:

(1) the development of a "core" of process-independent knowledge; and

(2) inclusion of hardware and software tools that facilitate rapid knowledge engineering.

C. Artificial Neural Networks

Artificial neural networks (ANNs) have found growing use in process control. We describe ANNs in detail in chapter 17, but a brief introduction is in order here. ANNs came from AI research, but use *numerical* (rather than symbolic) manipulation, and have some distinct advantages over "traditional" symbolic processing. In process control, a particularly important advantage of ANNs is the ability to properly process noisy, incomplete, or inconsistent data.

ANNs have been used in a number of areas in process control. Some applications include:

- *Multivariable and adaptive control*- one of the most promising future applications of neural networks. The idea here is to model a process with an ANN, and use the input-output response from the network for multivariable and adaptive control.

  This strategy hopes to improve the controller performance with a truly accurate multivariable process model. Controllers could become much more adaptive and flexible to changes in the process. The overall control system could be made much more robust.

- *Fault Diagnosis*- rule-based and model-based systems (such as FALCON) have a number of limitations. They waste valuable computer time trying to classify malfunction patterns. In addition, they cannot handle multiple faults unless the knowledge is explicitly placed into the system (a laborious task indeed).

ANNs overcome these difficulties. By design, they associate patterns of observable behavior with input parameters. Based on the connectivity between nodes, patterns typical with certain types of faults can be "built right in" to the network. There is no need to waste time classifying patterns explicitly (which is required in symbolic processing). Also, the network is well-suited to handling multiple faults as inputs and determining the proper output pattern.

- *Process Modeling-* ANNs can be used to do on-line or off-line process modeling. As mentioned, advanced control systems involving multiple variables rely on on-line process models. ANNs can provide the on-line modeling required. They are better suited than symbolic processors for handling rapidly changing systems. In addition, it makes no difference to the neural network if the process is nonlinear.

To do modeling, the neural network typically represents some simple mathematical relation when initially placed into the system. Once in use, the relation can be adjusted by changing the strength of signals hidden within the net. The goal is to create an optimal mapping between output and input. The end-result mapping is not bound by any mathematical relation, and is therefore very flexible.

- *Trend Analysis*- since ANNs are inherent pattern recognizers, they are excellent for trend analysis. Using trend analysis, we can optimize a controller based on the previous performance of the process. This gives a much more robust process-control system, especially if the process is nonlinear.

We introduce the principles and applications of ANNs in chapter 17.

D. Challenges in Process Control

Expert systems have advantages and disadvantages, and therefore, cannot and should not be applied to every type of problem. In the area of process control, expert systems have limited utility in applications involving (Shirley, 1987): standard (numerical) control algorithms, optimization techniques, advanced (numerical) control strategies, simulation, statistical process control and statistical quality control. In these application areas, expert systems may be helpful in augmenting these techniques, but it is doubtful that expert systems can replace these techniques.

Another difficulty with expert systems in process control is *brittleness*. Expert systems cannot be reliably "stretched" outside of their domain expertise. When an expert system encounters a novel situation, it may not perform adequately, or may outright fail. It is also proving to be unrealistic to believe that we can capture *all* of the knowledge of an expert. Perhaps only 70-90% of the knowledge is successfully articulated and encoded. Clearly, this "knowledge hole" contributes to brittleness and can cause problems when applied to process control.

Related to brittleness is *robustness*. A robust system can properly handle odd situations, equipment failure, errors, etc.. A robust system will not "lock up" when there is an input error; if a robust system cannot arrive at a solution, it will say why and how far it did get. It is difficult to make symbolic computing techniques as robust as numerical techniques. Consequently, control schemes using symbolic approaches have typically been less robust than more traditional approaches. While researchers have attempted to make expert systems tougher and more robust be incorporating deep knowledge, they are still too fragile for sensitive or important on-line applications in process control.

E. Future Directions

In the process-control area using traditional expert systems, the general trend appears to be more towards "process management". Using knowledge-based systems, we will be able to better control and manage process performance. In today's global economy, there is ever increasing pressure to produce top-quality products at the lowest possible cost. Using expert systems for process management will probably grow.

The consensus today is that symbolic-computing approaches are too slow for on-line, real-time applications in process control. EXACT, which *is* suitable for on-line control, is written in assembly language (Prolog or LISP both fail to be fast enough given today's hardware limitations).

Because of the difficulty in implementing expert-control techniques in real-time, full-scale implementations of AI systems for on-line process control have been limited to date. Hardware improvements, especially in parallel processing, would certainly help this cause. Applications involving both quantitative and qualitative processing (akin to the project PICON) would also help, and will probably continue to grow. Parallel processing can be used straightforwardly-- one processor performing numerical processing with the other performing symbolic processing.

We also cannot underestimate the importance of ANNs. Although from an industrial standpoint, there have been few commercial applications, ANNs have received much attention from control researchers (see chapter 17). Much of this research may come to fruition in the future.

A combination of symbolic computing and ANNs may be the most powerful implementation of artificial intelligence in process control. It is one of the few ways to combine both top-down (symbolic) theory with bottom-up (numeric) theory into a unified system. Symbolic processing could be used in a general, "top-down" mode as an explanation tool, a user-interface, and an overall program organizer. Numeric processing could be used in a "bottom-up" mode for specific modeling, control, trend analysis, and interaction analysis. A hybrid system combining the strengths of both areas would be very powerful indeed.

Finally, in more practical terms, engineers implementing expert systems realize two very important principles that have not been particularly appreciated as expert systems have developed: 1) *an excellent user-interface is vitally important* to a system's success, and 2) expert systems must be tied into current plant equipment -- *a stand-alone monolith will not suffice*. Improving an expert system's user-interfaces and/or plant-integration scheme is a very practical way to improve that expert system's utility.

F. References and Further Reading

A general reference covering many topics of expert-system applications to process control is *Chemical Process Control - CPC III: Proceedings of the Third International Conference on Chemical Process Control*, pp.803-913, Asilomar, CA, M. Morari and T.J. McAvoy, Editors, CACHE Corporation, Austin, Texas, and Elsevier, New York, NY January (1986). Another general reference in *Artificial Intelligence in Process Engineering*, M.L. Mavrovouniotis, Editor, Academic Press, San Diego, CA (1990). Some references of particular interest are Birky et. al. (1988), Kraus and Myron (1984), Higham (1985), Moore and Kramer (1986), Shirley (1987), Shinskey (1986), and Tzouanas (1988).

Astrom, K.J., and B. Wittenmark, "On Self-Tuning Regulators," *Automatica*, 9, 185 (1973).

Astrom, K.J., "Auto-Tuning Adaptation and Expert Control," *Proc. Amer. Control Conf.*, p.1514, Boston, MA, June (1985).

Basila, M.R. Jr., G. Stefanek, and A. Cinar, "A Model-Object-Based Supervisory Expert System for Fault-Tolerant Chemical Reactor Control," *Comput. Chem. Eng.*, **14**, 551 (1990).

Beaverstock, M., E.H. Bristol, and D. Fortrin, "Expert Systems as a Stimulus to Improved Process Control," *Proc. Amer. Control Conf.*, p.898, Boston, MA, June (1985).

Bhat, N. and T.J. McAvoy, "Use of Neural Nets for Dynamic Modeling and Control of Chemical Process Systems," *Comput. Chem. Eng.*, **14**, 573 (1990).

Birky, G.J., T.J. McAvoy, and M. Modarres, "An Expert System for Distillation Control Design," *Comput. Chem. Eng.*, 12, 1045 (1988).

Birky, G. "Knowledge Representation for Expert Systems in Chemical Process Control Design," PhD Dissertation, University of Maryland, College Park, MD (1988).

Birky, G.J. and T.J. McAvoy, "A General Framework for Creating Expert Systems for Control System Design," *Comput. Chem. Eng.*, 14, 713 (1990).

Bristol, E.H., "Objects, Rules, and Process Control," *Proc. Amer. Control Conf.*, p.1368, Pittsburgh, PA, June (1983).

Cao R. and T. McAvoy, "Effect of Interaction on EXACT Pattern Recognition Controller," *Proc. Amer. Control Conf.*, Seattle, WA, June (1986).

Chowdhury, J., H. Short, and R. Gibb, "Process Controllers Do Expert Guises," *Chemical Engineering*, pp.14-17, June 24 (1985).

Clarke, D.W. and P.J. Gawthrop, "Self-Tuning Controller," *Proc. IEEE Conf.*, 122, 929 (1975).

Cooper, D.J. and A.M. Lalonde, "Process Behavior Diagnostics and Adaptive Process Control," *Comput. Chem. Eng.*, 14, 541 (1990).

Cooper, D.J., R.F. Hinde, Jr., and L. Megan, "Pattern-Based Adaptive Process Control," *Comput. Chem. Eng.*, 14, 1339 (1990).

Dahlqvist, S.A., "Control of a Distillation Column Using Self-Tuning Regulators," *Can. J. Chem. Eng.*, 59, 118 (1973).

Dean, D.M., and G.F. Hall, "Improving Refinery Operations: A Goal of Today's Instrumentation," *Energy Prog.*, 7 (3), 142 (1987).

Freeman, D.D., "Artificial Intelligence Applications in Process Control," *Proc. Amer. Control Conf.*, Boston, MA, June (1985).

Garrison, D.B., D.M. Prett, and P.E. Steacy, "Expert Systems in Process Control and Optimization: A Perspective," *Shell Process Control Workshop*, Butterworths, Stoneham, MA (1986).

Gidwani, K., and C. Knickerbocker, "PICON (Process Intelligent CONtroller) - MAP (Manufacturing Automation Protocol) Network for Factory Automation," *ISA Adv. Instrum.*, 41 (1), 177 (1986).

Grainger, B.E., and M.L. Tomasello, "An Expert-System Interface to a Real-Time Data Monitoring and Control System," *ISA Adv. Instrum.*, 43 (1), 51 (1988).

Higham, E.H., "A Self-Tuning Controller Based on Expert Systems and Artificial Intelligence," *IEEE Control*, 85, 110 (1985).

Hoskins, J.C. and D.M. Himmelblau, "Artificial Neural Network Models of Knowledge Representation in Chemical Engineering", *Comput. Chem. Eng.*, 12, 881 (1988).

James, J.R., D.K. Frederick, and J.H. Taylor, "Use of Expert-Systems Programming Techniques for the Design of Lead-Lag Compensators," *IEE Proceedings*, **134**, 137 (1987).

Joseph, J., H.T. Wu, and B. Allan, "Automation of a Batch Process with Expert System Shells," *Chem. Eng. Prog.*, **85** (11), 87 (1989).

Karpoor, N., M. Modarres and T.J. McAvoy, "MAKE: Maryland Automatic Knowledge Extractor," *Proc. Amer. Control Conf.*, p. 1347, Minneapolis, MN, June (1987).

King, K.H. and N.F. Marsolan, "An Expert Adaptive Fuzzy Logic Control System on a Thermal Process Unit," *ISA Adv. Instrum.*, **43** (1), 67 (1988).

Konar, A.F., B.M. Thuraisingham, and P.E. Felix, "XIMKON - An Expert Simulation and Control Program," in *Artificial Intelligence in Process Engineering*, pp. 223-265, M. Mavrovouniotis, Editor, Academic Press, San Diego, CA (1990).

Kraus, T.W., and T.J. Myron, "Evaluation and Performance of a Self-Tuning PID Controller," from the EXACT product manual, The Foxboro Company, Foxboro, MA 02035.

Kraus, T.W., and T.J. Myron, "Self-Tuning PID Controller Uses Pattern Recognition Approach," *Control Engineering*, p.106, June (1984).

Kraus, T. "Self-Tuning Control: An Expert System Approach," *Adv. in Instrumentation*, 39, 695 (1984).

Leech, W.J., "A Rule-Based Process Control Method with Feedback," *ISA Adv. Instrum.*, **41**, 168 (1986).

Lewin, D.R. and M. Morari, "ROBEX: An Expert System for Robust Control Design," *Proc. Amer. Control Conf.*, p. 1374, Minneapolis, MN, June (1987).

Lewin, D.R. and R. Lavie, "Exothermic Batch Chemical Reactor Automation via Expert System," in *Artificial Intelligence in Process Engineering*, pp. 267-293, M. Mavrovouniotis, Editor, Academic Press, San Diego, CA (1990).

Lieslehto, J., and H.N. Koivo, "An Expert System for Interaction Analysis of Multivariable Systems," *Proc. American Control Conference*, Minneapolis, MN, June (1987).

Moore, R.L., "Chemical Process Control," *ACS Symp. Ser.*, **408** (Expert Syst. Appl. Chem.), 169 (1989).

Moore, R.L. and M.A. Kramer, "Expert Systems in On-Line Process Control," *Chemical Process Control- CPC III*, M. Morari and T.J. McEvoy, Editors, p.839, CACHE Corp., Austin TX, and Elsevier, New York, NY (1986).

Niida, K. and T. Umeda, "Process Control System Synthesis by an Expert System," *Chemical Process Control- CPC III*, M. Morari and T.J. McEvoy, Editors, p.869, CACHE Corp., Austin TX, and Elsevier, New York, NY (1986).

O'Conner, G.M., "An Expert System Approach to Fermentation Process Control," *MIT Short Course on Expert Systems in Process Engineering*, Lecture 15, Cambridge, MA (1987).

Orendo, R.S. J.A. Bernard, D.D. Lanning, and J.H. Hopps, "Design and Experimental Evaluation of an Automatically Reconfigurable Controller for Process Plants," *Proc. Amer. Control Conf.*, p.1662, Minneapolis, MN, June (1987).

Pang, G.K.H. "An Expert System for CAD of Multivariable Control Systems Using a Systematic Design Approach," *Proc. Amer. Control Conf.*, p.555, Boston, MA, June (1985).

Piovoso, M.J. and J.M. Williams, "Self-Tuning Control of pH," *Advances in*

Instrumentation, 39, 705 (1984).

Prett, D.M. and C.E. Garcia, *Fundamental Process Control*, pp. 190-241, Butterworth, Stoneham, MA (1988).

Shirley, R.S., "Some Lessons Learned Using Expert Systems for Process Control," *Proc. of the Amer. Control. Conf.*, pp. 1342-1346, Minneapolis, MN (1987).

Shinskey, F.G., "Two Expert Systems for Batch Reactor Control," *Proc. of the BIRA Conference on Expert Systems*, Antwerp, Belgium, October (1986).

Shinskey, F.G., "An Expert System for the Design of Distillation Controls," *Chemical Process Control-CPC III*, p. 895, M. Morari, T.J. McAvoy, Eds., CACHE Corp., Austin TX, and Elsevier, New York, NY (1986).

Shinskey, F.G., *Process Control Systems: Application, Design, and Adjustment*, Third Edition, pp.237-318, McGraw-Hill, New York, NY (1988).

Smith, R.E. and W.H. Ray, "CONSYDEX: An Expert System for Automated Computer-Aided Control System Design," *AIChE National Meeting*, Chicago, IL (199).

Stephanopoulos, George, "Process Control with a Computer," *CHEMTECH*, **17** (4), 251 (1987).

Stock, M., *AI in Process Control*, McGraw-Hill, New York, NY, (1989).

Taylor, J.H., "Expert-Aided Environments for CAE of Control Systems," *Proc. IEEE Symp. on CADCS*, Beijing, Peoples' Republic of China, August (1988).

Tzouanas, V.K., C. Georgakis, W.L. Luyben, and L.H. Ungar, "Expert Multivariable Control", *Comput. Chem. Eng.*, **12**, 1065 (1988).

Tzouanas, V.K., "An Expert Multivariable Controller for Distillation Process Control," PhD Dissertation, Lehigh University, Bethlehem, PA (1989).

Tzouanas, V.K., W.L. Luyben, C. Georgakis, and L.H. Ungar, "Expert Multivariable Control. 1. Structure and Design Methodology," *Ind. Eng. Chem. Res.*, **29**, 382 (1990a).

Tzouanas, V.K., W.L. Luyben, C. Georgakis, and L.H. Ungar, "Expert Multivariable Control. 2. Application of EMC to Two-Product

Distillation Columns," *Ind. Eng. Chem. Res.*, **29**, 389 (1990b).

Tzouanas, V.K., W.L. Luyben, C. Georgakis, and L.H. Ungar, "Expert Multivariable Control. 3. Extension of EMC to Three Product Side-Stream Distillation Columns," *Ind. Eng. Chem. Res.*, **29**, 404 (1990c).

Vagi, F., R.K. Wood, A.J. Morris, and M. Tham, "Self-Tuning Control of Distillation Columns: Theory and Practice," *Proc. American Control Conf.*, 1269, Boston, MA, June (1985).

Whitlow, J.E., "Expert System-Based Control of a Continuous Binary Distillation Column," PhD Dissertation, Vanderbilt University, Nashville, TN (1989).

Whitlow, J.E., and K.A. Debelak, "A Knowledge-Based Structure for Process Control," *Proc. Amer. Control Conf.*, p. 1354, Pittsburgh, PA, June (1989).

Yelstie, B.E., "Forecasting and Control Using Adaptive Connectionist Networks," *Comput. Chem. Eng.*, **14**, 583 (1990).

Ziegler, J.G., and N.B. Nichols, "Optimum Settings for Automatic Controllers," *Trans. ASME*, November (1942).

16.5 APPLICATIONS TO PROCESS DESIGN

A. Introduction

The goal of process design is to configure an industrial chemical process that achieves technical and economical objectives. To achieve technical objectives, there must be some type of prediction on how the process will behave. Modeling is required. Strategies are then used to attack and solve the problem. Achieving economical objectives requires designing the process in a cost-effective way. Optimization is implied.

Designing a process involves the following:

- *Process synthesis-* developing a flowsheet, i.e., determining appropriate unit operations and configuring them into a coordinated, process.

- *Method selection-* choosing the best processing method for the task at hand. For instance, if cooling needs to be done, should we use an air-cooled or water-cooled heat exchanger? If distillation is required, should we use ordinary distillation or extractive distillation? The optimal method must be chosen before a design can progress past a general process flowsheet.

- *Rigorous design and specification development-* developing final equipment specifications, including sizing, determining materials of construction, assessing energy and utility loads, etc.

- *Economic analysis-* determining both capital and operating costs; assessing the financial viability of the entire process.

- *Safety and sensitivity analysis-* the process must be safe, and not pose any unnecessary hazards or risks to personnel. In addition, the process should be robust and not overly sensitive. The process should be able to successfully adapt to changes both external and internal to the process itself. Examples include: 1) normal process variations in temperature and concentration, 2) changes in the climate or weather (including ambient temperature and humidity), and 3) changes in process equipment (e.g., filter cake build-up in a filtration process).

Computer-aided design (CAD) systems such as ASPEN PLUS, DESIGN II, and PRO II are written in FORTRAN. Historically, they have mostly been used for rigorous design, as well as economic analysis. Engineers typically choose the process methods, develop the flowsheet, and then input information into the CAD system to perform rigorous design and economic analysis.

Design utilizes both qualitative and quantitative information. For instance, if we wish to purify a corrosive component in a mixture, it would be best to separate it first; the corrosive nature of a material is qualitative knowledge. Design also requires efficient decision-making and problem-solving abilities. For these reasons, expert systems could be used to enhance the design process.

Expert systems can be applied to process design in a number of areas. In general, they can be used to:

- *Develop flowsheets*- utilize expert engineering knowledge to synthesize the process.

- *Select methods*- use process data with engineering heuristics to recommend the optimal processing method for the task at hand.

- *Automate the design process*- link process synthesis with method selection and rigorous CAD programs.

- *Select materials*- determine the appropriate materials of construction based on process and environmental conditions.

- *Select thermodynamic models and estimate physical properties*- select the best thermodynamic model(s) for the problem. For instance, if we were doing liquid-liquid extraction, we would *not* want to use the Wilson equation for activity coefficients (Walas, 1984; p.195). In addition, expert systems could operate a relational database for physical-property information retrieval. If no explicit data are available, the system could select a method and estimate the value of the physical property.

B. Applications

One challenge in applying expert systems to process design is that design is primarily a *synthesis* task. Expert systems are not particularly good at synthesis tasks. They are much better suited to problem solving via *problem decomposition*. Therefore, the most successful AI applications have taken a task-oriented, problem-decomposition, or "divide and conquer" approach to process design. We discussed this aspect previously in section 15.1B.

Applying expert systems to process design has a key advantage over applications to fault diagnosis and/or process control. As discussed above, fault-diagnosis and process control expert systems are application-specific, i.e., for each application, the knowledge base needs to be developed from scratch. In design, however, we can develop a *general* expert system. We only have to go through the expensive and time-consuming knowledge engineering stage once. The cost/benefit ratio for design expert systems is better than those in fault diagnosis and process control.

1. General Design Tools

There have been relatively few expert system tools to aid process engineering in a general sense. Most process engineering expert systems are focused, e.g., heat-exchanger network synthesis, separation process synthesis, etc. One of the better known research prototype programs that performs as a general design tool is DESIGN-KIT (Stephanopoulos et. al., 1987).

We discussed DESIGN-KIT section 10.2G , but let us review it again here. DESIGN-KIT is an object-oriented program developed to aid process-engineering activities such as: flowsheet development, control-loop configuration, and operational analysis. DESIGN-KIT supports graphic constructions -- the importance of the user-interface, with graphic support, cannot be underestimated if an expert system is to succeed.

As mentioned, DESIGN-KIT is object-oriented, and was developed on a SYMBOLICS LISP computer. It uses Common LISP as well as KEE (Knowledge Engineering Environment, by IntelliCorp). Four main objects in DESIGN-KIT that are relatively high in the hierarchy of objects are:

- *Graph*- includes the object name, graphic functions used to create it, and graphic procedures such as move, expand, and rotate.
- *Processing Unit*- includes the object name, input and output streams, modeling equations, design methodology, constraints, start-up procedures, and potential control loops.
- *Reaction*- includes the object name, reaction mechanism, kinetic-rate expression, catalyst, operating conditions, and yield data.

- *Equipment*- includes the object name, sizing and costing methodologies, costing assumptions, and design and operational constraints.

DESIGN-KIT is built to perform numerical analysis and simulation, rule-based reasoning using forward- and/or backward-chaining, and degree of freedom analysis.

DESIGN-KIT demonstrates how to structure and build an expert system for general process engineering. It clearly demonstrates the utility of object-oriented programming, particularly in the development of modular systems with a high degree of complexity.

DESIGN-KIT is only a research prototype; its knowledge base is limited. Implementing a "full-blown" system would be a major effort requiring many man-years of time, and would be an expensive endeavor. Thus, DESIGN-KIT amply demonstrates a challenge that continues to face expert systems: high development cost. Development costs are decreasing, but more efficient software systems that allow for rapid, lower cost development are still needed.

2. Separation Design

EXSEP, discussed in chapters 12 to 15, is an expert system for multicomponent separation design. It has been implemented into an operational prototype. EXSEP is a rule-based expert system for separation flowsheet development. It takes in multicomponent feed and product specifications for applications where distillation is appropriate. It develops an economically attractive flowsheet using both sharp and sloppy separations. It also does a shortcut thermodynamic feasibility analysis to ensure the feasibility of sloppy separations.

There have been a number of proposed process-synthesis expert systems in the literature. Unfortunately, most of these proposals are strictly conceptual. Few of these proposals have actually been implemented into research or prototype expert systems. Since expert-system development is evolutionary, it may be difficult at times to judge the utility of a proposal. The "acid test" for expert systems is the implementation stage, and that is where the true worth of proposed systems is demonstrated.

Kirkwood et al. (1988) have developed a prototype expert system for the synthesis of chemical process flowsheets. They use a hierarchical approach, combining quantitative and qualitative knowledge into the system. They also incorporate economic estimates, something not many systems have done.

Barnicki and Fair (1990) have proposed an overall structure for a *general* separation expert system. The proposed system would do:

(1) process synthesis,

(2) method selection, and

(3) rigorous design.

Their work is still in the structural development stages. Although implementing this system would require quite an investment in knowledge engineering, it holds much promise.

Structurally, their system first does an equilibrium flash to separate components into three groups:

(1) gases;

(2) transition gas-liquid components; and

(3) liquids.

To date, they have only addressed flowsheet development and design for liquid separations; they introduce three modules for this task:

(1) Manager- develops the flowsheet, and drives Selector and Designer.

(2) Selector- determines the separation method (e.g., ordinary distillation, liquid-liquid extraction, etc). Heuristically eliminates inappropriate methods and gives a set of methods to consider.

(3) Designer- designs the equipment to achieve the separation.

To develop a flowsheet for liquid separations, two parts are required:

(1) the actual ordered separation sequence; and

(2) the method of separation for each split.

When using distillation only, the choice of method and sequence can be *decoupled*. When considering other methods of separation, however, the sequence and method need to be developed together in a *coupled* fashion. They correctly recognize this point, and incorporate it into the system's knowledge base.

In developing the method and sequence conjunctively, they recommend: 1) favoring distillation over all other processes, and 2) deriving the sequence such that distillation-based separations are done first. Once all distillation processing is complete, separations using mass-separating agents, and crystallization, are considered.

With the technique developed by Barnicki and Fair, distillation is done first, with other methods following. The distillation separation sequencing uses heuristics. They use the heuristic method of Nadgir and Liu (1983; see section 12.1E and 13.6), modified to include azeotropes. They recommend performing:

(1) Ordinary distillation first, provided the separation is not difficult,;
(2) Ordinary distillation second, but this time with difficult separations (low relative volatilities between components); and
(3) Azeotropic separations last.

As mentioned, this work is in the conceptual stages, but offers promise.

SEPEX (Sunol et. al., 1990) is another proposed separation expert system in the conceptual stages. Given the feed streams and the desired products, the system first uses heuristic knowledge to recommend a separation method. A separation sequence is then developed, also with heuristic knowledge. The unit operations recommended are modeled using approximate methods, and then this recommended structure is optimized using mixed integer nonlinear programming (MINLP). The results from MINLP are fed into a genetic search algorithm, where conclusions deduced from this stage are fed back to the beginning of the program. What results is an iterative process of 1) method selection, 2) separation sequence development, 3) approximate modeling, 4) MINLP optimization, and 5) rigorous modeling. Feedback to the heuristics is done for the purpose of machine learning, or to correct an infeasible structure.

SEPEX is a complex mixture of modeling tools, heuristic knowledge, inexact reasoning, and algorithmic, mathematical optimization techniques. SEPEX uses heuristic and algorithmic methods hierarchically. One of the primary goals of SEPEX is to combine heuristic knowledge with deep knowledge, i.e., first-principles modeling.

Debelak et. al. (1987) have developed a simple prototype expert system for ion exchange. This system recommends a resin for effective ion exchange for the removal of the following heavy metals from process streams: Hg, Pb, Cd, Fe, Cu, U, Cs, and Pu. The system can only handle cations; anions and radioactive materials are outside of its abilities. They have a number of proposals to expand the system, such as: including equilibrium-design calculations, adding additional separation modules, and being able to handle radioactive materials.

3. Heat-Exchanger Network Synthesis

Heat-exchanger network synthesis (HENS) is another chemical process-design area where we are able to apply expert systems. The objective of HENS is develop a flowsheet that will:

(1) heat cold process streams up to desired temperatures;
(2) cool hot process streams down to desired temperatures; and
(3) achieve the energy exchange in a technically feasible and economically attractive way.

HEATEX (Grimes et.al 1982, Rychener, 1984) is a prototype expert system developed at Carnegie-Mellon targeted for heat-exchanger network synthesis. HEATEX uses an *evolutionary development strategy*. The system first proposes a technically feasible network that is not yet optimized. After developing this initial network, HEATEX enters an interactive phase, where the user can suggest evolutionary changes. HEATEX has no rules for optimizing the network itself. It relies exclusively on the user for evolutionary optimization.

HEATEX is written in OPS5, a language best-suited for forward-chaining, rule-based expert systems. HEATEX has grown to a system of 115 production rules, but runs too slowly to be practical. There are no plans to build evolutionary synthesis strategies into the system, nor are there any plans to expand the system.

Chen et. al. (1989) have developed an expert system called SPHEN (Synthesis of Practical Heat-Exchanger Networks). SPHEN develops a heat-exchanger network targeted for minimal energy consumption with the minimum number of units. The network development strategy in SPHEN involves:

(1) Simulation of the process.

(2) Modification of the network to improve cost-effectiveness.

(3) User-directed feedback to re-simulate and remodify the system.

This process continues until the user is satisfied with the network.

Like HEATEX, SPHEN uses an evolutionary process-synthesis strategy. An initial network is synthesized, and then repetitively altered, with the goal of improving cost-effectiveness with each alteration. Unfortunately, SPHEN does not contain the knowledge to do the process alterations itself. The user must perform all alterations to the process himself. Alterations at the user's disposal are:

- Adding an exchanger;
- Deleting an exchanger;

- Exchanging the hot and cold streams of two exchangers; and
- Changing the type of exchanger.

SPHEN has been demonstrated to work on a commercially significant problem, i.e., a nine-hot-stream, one-cold-stream network design for a crude oil unit.

SPHEN is written in Common LISP and is a forward-chaining rule-based system. However, it is a "hybrid" system in the sense that it is linked into a FORTRAN program too. The process simulator in SPHEN is, understandably, numerically intensive. Therefore, FORTRAN is used as the programming language in the simulator. Overall program control and flowsheet development for the expert system is done by LISP; heavy "number crunching" in the simulator is done by FORTRAN. The two languages are linked together and used on a personal computer.

4. Thermodynamic Model Selection and Physical Property Estimation

The foundation of accurate chemical process design and simulation is a correct estimate of physical and thermodynamic properties. Depending on:

(1) the thermodynamic state (i.e., the pressure-volume-temperature-conditions of the system; and

(2) the processing method (e.g., liquid-liquid extraction, distillation, etc.),

certain models may be more suitable than others. Most computer-aided-design and simulation packages require the user to specify models or equations of state to use. A design engineer may not have the knowledge and experience to choose the best model. Arbitrary choices based on hearsay or familiarity may result in incorrect results.

To aid design engineers in choosing the proper model for a simulation, an expert system can be used. Theoretically, an expert system could be linked right in with a commercial CAD package, thereby automating the thermodynamic method-selection task. Based on expert knowledge of the performance of various models under certain conditions, we can use an expert system to choose the method that is most appropriate.

CONPHYDE (COnsultant for PHYsical property DEcisions), developed at Carnegie-Mellon University (Banares-Alcantara, 1982), is a "stand-alone" expert system that selects the appropriate physical property model. By "stand-alone," we mean the system is not linked to any CAD package and can run independently on its own.

CONPHYDE selects thermodynamic equations of state for vapor-liquid equilibrium calculations. It uses an interactive "question and answer" session to acquire the information necessary to recommend a model.

CONPHYDE uses rules and statistical inference techniques to draw conclusions and make recommendations. An example rule from CONPHYDE is:

IF *operating conditions are near the critical region*

THEN *use the Peng-Robinson equation of state*

CONPHYDE is one of the first expert systems produced for the selection of thermodynamic models. What CONPHYDE lacks, however, is depth in its knowledge base. It is only practical for thermodynamic models for vapor-liquid equilibria (VLE). In addition, because CONPHYDE never progressed farther than the stage of research prototype, even the VLE knowledge is limited. Being the first application of expert systems to thermodynamic model selection, CONPHYDE is a well-organized system.

Gani and O'Connell (1989) have also developed a knowledge-based system for the selection of thermodynamic models. They attempt to add more depth to the knowledge base than CONPHYDE. In addition, they use a different problem-solving approach from that of CONPHYDE. First, they group the knowledge about method selection into three primary categories:

- *Problem type*- the process being used, e.g., distillation, liquid-liquid extraction, single-stage flash, or gas absorption.

- *Chemical system*- the nature of the chemicals being processed:

    - *Nonpolar fluids*: subdivided into light hydrocarbons (C_3 and lighter), intermediate aromatics and paraffins (C_4 and higher), aliphatics and napthenes (C_4 and higher), and petroleum fractions.

- *Polar non-associating compounds*: subdivided into organic (amines, ethers, chlorides, etc) and inorganic.
- *Physically associating/solvating compounds*: subdivided into organic (alcohols, ketones, aldehydes, acids, etc.) and inorganic (H_2SO_4, etc.).
- *Aqueous systems*: subdivided into distilled water and ionic water.
- *Polymers*.
- *Ionic Compounds:* subdivided into weak electrolytes and strong electrolytes.

Based on the chemical system, the program assesses whether the solutions are predicted to be ideal or not.

- *System conditions*- the pressure-temperature (P-T) conditions of the system, as well as the mixture composition. The P-T conditions are subdivided into very low pressure ($P < 0.1$ atm), low pressure ($0.1 < P < 1$ atm), atmospheric to moderate pressure ($1 < P < 20$ atm), high pressure ($20 < P < 100$ atm), very high pressure ($P > 100$ atm), and critical pressure ($P \rightarrow P_c$). The mixture conditions are subdivided into pure component, binary mixture, and multicomponent mixture.

Based on the three groupings of problem type, chemical system, and system conditions, a fourth is used to discriminate between models (e.g., model discrimination based on vapor-phase equilibria, liquid-phase equilibria, residual properties, excess free energy, PVT properties, electrolyte models, polymer models, surface-tension models, viscosity models, etc.).

After grouping the knowledge, the program issues a set of indices for each category. A large "index table" is used to simply look up the best model.

The program is algorithmic, and is written in FORTRAN. A LISP version is being developed. Because the program is algorithmic, we probably would not classify it as "artificial intelligence" if our definition of AI was grounded on symbolic computing. Nevertheless, we might call it "knowledge-based." In addition, the authors point out an advantage of writing it in FORTRAN-- it can be easily incorporated into commercial CAD programs, all of which are written in FORTRAN.

C. Challenges in Process Design

There are two major challenges to overcome if expert systems are to be widely used in chemical process design: 1) development costs are too high, and must be reduced, and 2) there must be a means of meshing both qualitative information (i.e., heuristic reasoning) with quantitative information (i.e., modeling, simulation, and optimization) into a single, integrated system.

Development costs are too high for many expert-system applications in process design. Consequently, the cost/benefit ratio is insufficient to support development. Expert-system applications to process design, however, do have a cost/benefit advantage over applications in other areas of chemical engineering. Why? Process-design expert systems can be used *repetitively*, and therefore, can be used to boost productivity. Process-design expert systems have a better economy of scale. For example, let us consider an expert system that sizes and selects a control valve based on the fluid and process conditions. Control valve sizing and selection is a process that engineers perform often. By accessing an expert system, they could perform this task almost instantaneously, resulting in a boost in productivity.

Design is a hybrid task that incorporates both qualitative and quantitative information. Qualitative information may be in the form of heuristics. For example, EXSEP uses the heuristic "remove corrosive and hazardous materials first." Thus, some form of qualitative reasoning is essential for a process-design expert system.

However, design goes beyond qualitative reasoning. We must model, adjust, and optimize the process. Numerical techniques are required. Thus, design is a unique, integrated combination of qualitative and quantitative approaches. Thus, if expert systems are to be widely used in chemical process design, we must develop ways of meshing qualitative and quantitative analysis into a single, integrated package.

D. Future Directions

Coupling expert systems and process design could be an area with a significant payoff in the future. The primary reason is that design expert systems do not have to be application-specific, giving them a favorable cost/benefit ratio.

There will probably be an emphasis on "automation" of the design process. Typically, process synthesis involves developing a tentative flowsheet, estimating physical properties, and using CAD systems to assess feasibility and cost. Expert systems will be used to make this process more productive. They will probably be used as interactive tools that control commercial CAD programs.

As mentioned in section 16.5C, two major hurdles to overcome are the high cost of developing expert systems, and the current inability to systematically mesh qualitative and quantitative information into a single, integrated system. There will clearly be an effort in both of these areas in the future.

In contrast to applications in fault diagnosis or process control, there are no "high profile" AI projects on design that stand out right now as being particularly ambitious. Nevertheless, there is sizable activity in design expert systems. Most major chemical companies have AI groups developing small design expert systems. While these projects are "quiet" outside of these companies, they are making contributions. As personal computer hardware continues to improve, and as these in-house groups develop more applications, design expert systems will become very common.

E. References

A few key references in this section are Barnicki and Fair (1990), Kirkwood et. al. (1988), Liu et. al. (1990), Myers et. al. (1988), Niida et. al. (1986), and Sargent (1990).

Banares-Alcantara, R., "Development of a Consultant for Physical Property Predictions", *Master's thesis*, Carnegie-Mellon University (1982).

Bar, M. and M. Zeitz, "A Knowledge-Based Flowsheet-Oriented User Interface for a Dynamic Process Simulator," *Comput. Chem. Eng.*, 14, 1275 (1990).

Barnicki, S.D., and J.F. Davis, "Designing Sieve Tray Columns. Part One: Tray Design," *Chemical Engineering*, pp. 140-146, October (1989).

Barnicki, S.D., and J.F. Davis, "Designing Sieve Tray Columns. Part Two: Column Design and Verification," *Chemical Engineering*, pp. 202-212, November (1989).

Barnicki, S.D. and J.R. Fair, "Separation Synthesis: A Knowledge-Based

Approach. 1. Liquid Mixture Separations," *Ind. Eng. Chem. Res.*, **29**, 421 (1990).

Barnwell, J., and E. Boris, "Expert Systems and the Chemical Engineer," *The Chemical Engineer*, **440**, 41 (1987).

Beltramini, L. and R.L. Motard, "KNOD-- a Knowledge-Based Approach for Process Design," *Comput. Chem. Eng.*, **12**, 939 (1988).

Beltrami, L. "A Knowledge-Based Approach for the Design of Chemical Processes," PhD Dissertation, Washington University, St. Louis, MO (1988).

Chen, B., J. Shen, Q. Sun, and S. Hu, "Development of an Expert System for Synthesis of Heat Exchanger Networks," *Comput. Chem. Eng.*, **13**, 1221 (1989).

Csukas, B., Z. Kozar, and P. Arva, "Multicriteria Valuated Prolog Synthesizing Algorithms," *Comput. Chem. Eng.*, **13**, 595 (1989).

Debelak, K. A., M.R. Leuze, J.R. Bourne, J.E. Whitlow, B.A. Antao, O. Patina-Siliceo, and D.J. Pruett, "A Prototype Expert System for Separation Science," *AIChE Summer National Meeting*, Minneapolis, MN August (1987).

Douglas, J.M., "Synthesis of Multi-Step Reaction Processes," *Foundations of*

Computer-Aided Process Design. p.79, J.J. Siirola, I.E. Grossman, and G. Stephanopoulos, editors, CACHE Corp., Austin, TX, and Elsevier, New York, NY (1990).

Engelmann, H.D., H.H. Erdmann, R. Funder, and K.H. Simmrock, "The Solving of Complex Synthesis Problems Using Distributed Expert Systems," *Comput. Chem. Eng.*, 13, 459 (1989).

Gandikota, M.S. and J.F. Davis, "Expert Systems for Selection in Design", *AIChE Annual Meeting, San Francisco* (1989).

Gandikota, M.S. and J.F. Davis, "A Task Approach to Knowledge-Based Systems for Process Selection and Synthesis," *AIChE Annual Meeting*, Chicago, IL (1990).

Gani, R. and J.P. O'Connell "A Knowledge-Based System for the Selection of Thermodynamic Models," *Comput. Chem. Eng.*, 13, 397 (1989).

Grimes, L.E., M.D. Rychener, and A.W. Westerberg, "The Synthesis and Evolution of Networks of Heat Exchange that Feature the Minimum Number of Units", *Chem. Eng. Comm.*, 14, 339 (1982).

Gunderson, T., "Retrofit Process Design: Research and Application of Systematic Methods," *Foundations of Computer-Aided Process Design*. p.213, J.J. Siirola, I.E. Grossman, and G. Stephanopoulos, editors, CACHE Corp., Austin, TX, and Elsevier, New York, NY (1990).

Huang, Y.W., "Designing Intelligent Knowledge and Data Bases for Separation Scheme Synthesis- A Case Study of Knowledge in Process Engineering," PhD Dissertation, Kansas State University, Manhattan, KS (1988).

Huang, Y.W., and L.T. Fan, "Fuzzy-Logic Rule-Based System for Separation Sequence Synthesis: An Object-Oriented Approach," *Comput. Chem. Eng.*, 12, 601 (1988).

Huang, Y.W., and L.T. Fan, "Designing an Object-Oriented Hybrid Database for Chemical Process Engineering," *Comput. Chem. Eng.*, 12, 973 (1988).

Kirkwood, R.L., "PIP- Process Invention Procedure: A Prototype Expert System for Synthesizing Chemical Process Flowsheets," PhD Dissertation,University of Massachusetts, Amherst, MA (1987).

Kirkwood, R.L., M.H. Locke, and J.M. Douglas, "A Prototype Expert System for Synthesizing Chemical Process Flowsheets," *Comput. Chem. Eng.*, 12, 329 (1988).

Lahdenpera, E., E. Korhonen, and L. Nystrom, "An Expert System for the Selection of Solid-Liquid Separation Equipment," *Comput. Chem. Eng.*, 13, 467 (1989).

Lien, K., G. Suzuki, and A.W. Westerberg, "The Role of Expert Systems Technology in Design," *Chem. Eng. Sci.*, **42**, 1049 (1987).

Liu, Y.A., T.E. Quantrille, and S.H. Cheng, "Studies in Chemical Process Design and Synthesis: 9. A Unifying Method for the Synthesis of Multicomponent Separation Sequences with Sloppy Product Streams," *Ind. Eng. Chem. Res.*, **29**, 2227 (1990).

Lu, M.D. and R.L. Motard, "Computer-Aided Total Flowsheet Synthesis," *Comput. Chem. Eng.*, **9**, 431 (1985).

Lu, M.D. and R.L. Motard, "An Expert System for Computer-Aided Flowsheet Design," *Inst. Chem. Eng. Symp. Ser.*, **92** (Process Syst. Eng., PSE '85) 517 (1985).

Mavrovouniotis, M., "Computer-Aided Design of Biochemical Pathways," PhD Dissertation, Massachusetts Institute of Technology, Cambridge, MA (1988).

Modi, A.K. and A.W. Westerberg, "Integrating Learning and Problem Solving within a Chemical Process Designer", *Engineering Design Research Center*,Report No. 06-82-89, Carnegie-Melon University, Pittsburgh, PA (1989).

Myers, D.R., J.F. Davis, and D.J. Herman, "An Expert System for Designing Distillation Column Plates," *AIChE Annual Meeting, New York* (1987).

Myers, D.R., J.F. Davis, and D.J. Herman, "A Task-Oriented Approach to Knowledge-Based Systems for Process Engineering Design," *Comput. Chem. Eng.*, 12, 959 (1988).

Nadgir, V. M. and Y. A. Liu, "Studies in Chemical Process Design and Synthesis: Part V. A. Simple Heuristic Method for Systematic Synthesis of Initial Sequences for Multicomponent Separations," *AIChE J.*, 29, 926 (1983).

Nelson, D.A. and J.M. Douglas, "A Systematic Procedure for Retrofitting Chemical Plants to Operate Utilizing Different Reaction Paths," *Ind. Eng. Chem. Res.*, 29, 819 (1990).

Netterfield, T., and A.K. Sunol, "An Expert System for Separation Technology Selection," AIChE Annual Meeting, New York, NY, November (1987).

Niida, K., J. Itoh, T. Umeda, S. Kobayashi, and A. Ichikawa, "Some Expert System Experiments in Process Engineering," *Chem. Eng. Res. Dev.*, **64**, 372, September (1986).

Rychener, M.D. "Expert Systems for Engineering Design: Problem Components, Techniques, & Prototypes", *Engineering Design Research Center*, Carnegie-Mellon University, Pittsburgh, PA (1984).

Sargent, R.W.H., "Process Design: What Next?" *Foundations of Computer-Aided Process Design*. p.213, J.J. Siirola, I.E. Grossman, and G. Stephanopoulos, editors, CACHE Corp., Austin, TX, and Elsevier, New York, NY (1990).

Sheppard, C.M., "Constraint-Directed Nonsharp Separation Sequence Design--A Knowledge-Based Approach," PhD dissertation, Washington University, St. Louis, MO (May, 1989).

Siletti, C.A., "Computer-Aided Design of Protein Recovery Processes," PhD Dissertation, Massachusetts Institute of Technology, Cambridge, MA (1988).

Siletti, C.A., "Design of Protein Purification Processes by Heuristic Search," pp. 295-310, in *Artificial Intelligence in Process Engineering*, M. Mavrovouniotis, Editor, Academic Press, San Diego, CA (1990).

Stephanopoulos, G., "Synthesis in Process Development: Issues and Solution Methodologies," *Inst. Chem. Eng. Symp. Ser.*, **92** (Process Syst. Eng., PSE '85) 427 (1985).

Stephanopoulos, G., and D.W. Townsend, "Synthesis in Process Development," *Chem. Eng. Res. Dev.*, **64**, 160 (1986).

Stephanopoulos, G., J. Johnston, T. Kriticos, R. Lakshmanan, M. Mavrovouniotis, and C. Siletti, "DESIGN-KIT: An Object-Oriented Environment for Process Engineering," *Comput. Chem. Eng.*, **11** (6), 655 (1987).

Stephanopoulos, G., "Artificial Intelligence and Symbolic Computing in Process Engineering Design," *Foundations of Computer-Aided Process Design*. p.21, J.J. Siirola, I.E. Grossman, and G. Stephanopoulos, editors, CACHE Corp., Austin, TX, and Elsevier, New York, NY (1990).

Sunol, A.K., and D.W.T. Rippin, "The Architecture of SEPEX," *Technische Chemisches Lab*, ETH, CH 8092 Zurich, Switzerland (1990).

Suzuki, K., K. Niida, and T. Umeda, "Computer-Aided Process Design and Production Scheduling With Knowledge Base," *Foundations of Computer-*

Aided Process Design. p.49, J.J. Siirola, I.E. Grossman, and G. Stephanopoulos, editors, CACHE Corp., Austin, TX, and Elsevier, New York, NY (1990).

Yadar, T. and R. Govind, "Nonsharp Separation Synthesis Using Genetic Algorithms," *AIChE Annual Meeting*, San Francisco, CA, November (1989).

Wehe, R.R., K. Lien, and A.W. Westerberg, "Control Architecture Consideration for a Separation Systems Design Expert," Report No. EDRC-06-29-87, Engineering Design Research Center, Carnegie-Melon University, Pittsburgh, PA (1987).

Wehe, R.R., and A.W. Westerberg, "An Algorithmic Procedure for the Synthesis of Distillation Sequences with Bypass," *Comput. Chem. Eng.*, 11, 619 (1987).

Westerberg, A.W., "Synthesis in Engineering Design," *Comput. Chem. Eng.*, 13, 365 (1989).

Wallas, S.M., *Phase Equilibria in Chemical Engineering*, Butterworth, Stoneham, MA (1984).

16.6 APPLICATIONS TO PROCESS PLANNING AND OPERATIONS

A. Introduction

Expert systems can enhance process operations. In most applications to process planning and operations, expert systems:

- develop operating procedures, or
- recommend previously developed procedures for a given situation.

For example, we could have an expert system controlling the inventory in a warehouse. Based on the amount of raw materials and finished products in the warehouse, as well as pending orders, an expert system could develop a production schedule. As another example, an expert system could procedurally help the start-up or shut-down of a production unit.

With the exception of batch process engineering, process planning and operations is not a typical topic discussed in chemical engineering. There is little formal theory involved. However, from an engineering-management standpoint, there are principles that can be used to guide our decision-making. Some of these are:

- *Science and engineering*- the conservation laws, principles of thermodynamics and rate processes, etc.
- *Finance*- financial management techniques and principles, such as

net-present-value analysis or just-in-time inventory control.

- *Risk management*- use of risk management techniques, issue prioritization, and risk minimization.
- *Safety*- assessment of physical hazards for the purpose of accident prevention.
- *Operating constraints*- constraints on time, physical or financial limitations, product specifications, etc.

Because there is no unifying theme or theory, applications of expert systems to planning and operations tends to be "fluid," i.e., developments and techniques are dependent on problem. As with fault diagnosis and control, most expert systems applied to planning an operations are application-specific.

There is not much in the literature on expert-system applications to process planning and operations. Most applications that have been developed are used by companies and are thus proprietary.

B. Applications

Foulkes et al. (1988) have developed an expert system that synthesizes complex pump and valve-sequencing operations. The goal is to *safely* achieve a desired flow through a process-piping network. When sequencing valve and pump operations, accidents occur because:

(1) operators are responsible for a very large area of the plant, and may not be able to remember or assimilate all valve and pump activities;and

(2) concurrent operations are frequent, further increasing the complexity of the operation.

The expert system developed by Foulkes et. al. helps prevent spills or safety hazards. They took techniques developed by Rivas et al. (1974) and O'Shima (1978), expanded upon them, and developed an expert system written in Prolog.

Halasz et. al. (1989) introduced BATCHKIT, a knowledge-based approach for batch process planning. BATCHKIT is a process tool that is targeted to:

- assign process tasks to available equipment items;
- generate a campaign, i.e., a set of feasible process tasks that achieve objectives; and
- allocate time schedules to campaigns to match production requirements.

Once a set of campaigns has been generated, one can be chosen based on linear programming. Objective functions such as minimum cost, minimum time, or maximum profit can be used.

On a more theoretical level, there have been attempts to formalize and generalize aspects of knowledge-based planning. Most of this work has been done by AI researchers rather than chemical engineers. Unfortunately, general-purpose (i.e., domain-independent) approaches to date are simply too broad to be practical. They do not possess the knowledge specific to process planning, and hence lack the power to solve problems unique to the chemical process industry.

Fusillo et. al. (1988) introduce the theory of using local numerical models in an expert system. They viewed chemical process planning as a task of getting from the starting state to an ultimate goal state. A means-end analysis is used to develop the plan. Fusillo et. al. also introduced planning via problem decomposition. Here, they use stationary (unchanging) states and intermediate target (desired) states. Functional modeling is introduced, where process units are modeled as sources and sinks of physical quantities.

Lakshmanan et. al. (1988) discuss the theory of *nonlinear* planning. Linear planning is starting at one end of the plan (i.e., the "initial state") and moving step-wise closer to the end of the plan (i.e., the "goal state"). Once the goal state is reached, the plan is complete. Nonlinear planning, on the other hand, uses local "partial plans". Some aspects of each partial plan are left unspecified. Pruning infeasible aspects of partial plans improves the efficiency of the expert system over linear planning. They use a hierarchical, object-oriented framework for planning. Nonlinear planning, while complex, has some promising characteristics.

More recently, Huang et. al. (1991) introduce MIN-CYANIDE, an expert system that minimizes cyanide wastes in electroplating plants. MIN-CYANIDE evaluates and recommends process changes to minimize cyanide production. It evaluates options such as drag-out minimization, bath-life extension, rinse-water reduction, and replacement with non-cyanide solution. MIN-CYANIDE uses *Personal Consultant Plus*, an expert system shell from Texas Instruments and implemented on an IBM PC AT. MIN-CYANIDE uses fuzzy logic. Unfortunately, MIN-CYANIDE was developed by consulting literature publications only. No plant personnel or domain experts were consulted.

An interesting work is that of Aelion and Powers (1990) on retrofit synthesis of flowsheet structures for improving operating procedures. The method developed requires the input of: the initial flowsheet description, local and global constraints, the initial and goal states, possible procedural and structural actions, a unit-model library, safety and reliability functions, as well as economic evaluation functions. The interesting aspect of the Aelion and Powers work is that it is a system that can detect inadequacies of a current design, recommend changes in current procedures, and if needed, *modify the existing process structure*.

The system operates on the principle of *stationary states*, i.e., safe and reliable processes have procedures that have intermediate, stable, *stationary states* where the system can wait until the next action is taken. If something goes wrong, the system can retreat back to an intermediate state, thereby avoiding a complete shutdown.

The system is written in LISP, operates on a Macintosh IIcx personal computer, and uses a backward-chaining inference strategy. Frames are included to better organize the knowledge. The system uses means-end analysis to solve problems. The initial and goal states are given, and the system applies operators that adjust the current state and drive it towards the goal state. Constraints are checked to make sure there are no violations, and the plan is evaluated for feasibility.

A safety assessment of the process is performed. This assessment identifies procedures with high failure rates. Operating procedures may be developed or changed in the event a failure does occur. Importantly, these changes may also incorporate modifications to the process flowsheet.

C. Future Directions

In the late 1980's, expectations for expert-system applications to operation management were reduced. As research progressed, it became clear that process planning and scheduling is challenging from an AI standpoint. One of the biggest problems is the volume of information and the size of the state space. Combinatorial complexity is so high that practical systems for plant-wide planning are not within reach.

Instead, we are seeing, and will continue to see success in systems with more moderate goals. For example, inventory and warehouse control is a practical application of expert-system techniques. Another area which may continue to find interest is in the development and modification of operating procedures.

One area of planning that has continued to grow is financial planning. Many banks, money managers, and financial planners have been utilizing knowledge-based systems. This growth is expected to continue, and to penetrate financial management in the process industries.

There have been a few application-specific planning aids developed by individual companies. For example, Digital Equipment has developed ISA, Intelligent Scheduling Assistant, to aid planning in production scheduling and inventory control. Most of the work done by companies is proprietary, and therefore, information is not readily available.

There will probably be continued growth in expert-system applications to process start-up and shut-down, as well as for scheduling in batch-processing plants. There is enough understanding of batch process engineering to support this work. In addition, many batch plants are complex enough to warrant an expert system, but are not so complex that the system would "balloon" out of control.

D. References and Further Reading

Some key articles are Daugherty and Felder (1990), Egli et. al. (1986), Musier and Evans (1990), and Stephanopoulos (1987).

Aelion, V., and G.J. Powers, "A Unified Strategy for Retrofit Synthesis of Flowsheet Structures for Safe Operating Procedures," *AIChE National*

Meeting, Chicago, IL (1990).

Bunn, A.R., and F.P. Lees, "Expert Design of Plant Handling Hazardous Materials," *Chem. Eng. Res. Des.*, **66**, 419 (1988).

Calandranis, J., G. Stephanopoulos, and S. Numokawa, "DIAD-KIT/Boiler: On-Line Performance Monitoring and Diagnosis," *Chem. Eng. Prog.*, **86** (1), 60 (1990).

Charpentier, L.R. and M.A. Tuck, "Batch Decision Support System Enhances Safety of Reactor Operation," *ISA Adv. Instrum.*, **41**, 71 (1986).

Chen, C.L., Y.C. Chao, H.P. Huang, J.J. Jen, and M.D. Fang, "An Expert System for Efficient Operation of a Blast Furnace," *J. Chin. Inst. Chem. Eng.*, **20**, 201 (1989).

D'Ambrosio, B., M.R. Fehling, S. Forrest, P. Raulefs, and B.W. Wilber, "Real-Time Process Management for Materials Composition in Chemical Manufacturing," *IEEE Expert*, **2**, 81 (1987).

Daugherty, D.P., and R.M. Felder, "An Expert System for Scheduling Production in a Multipurpose Specialty Chemicals Plant," *Plant/Operations Progress*, **9**, 44, January (1990).

Egli, U.M., and D.W.T. Rippin, "Short-Term Scheduling for Multi-Product Batch Chemical Plants," *Comput. Chem. Eng.*, 10, 303 (1986).

Finn, G.A., and K.F. Reinschmidt, "Expert Systems in Operation and Maintenance," *Chem. Eng.*, 97 (12), 131, February (1990).

Foulkes, N.R., M.J. Walton, P.K. Andow, and M. Galluzzo, "Computer-Aided Synthesis of Complex Pump and Valve Operations", *Comput. Chem. Eng.*, 12, 1035 (1988).

Fusillo, R.H. and G.J. Powers, "A Synthesis Method for Chemical Plant Operating Procedures," *Comput. Chem. Eng.*, 11, 369 (1987).

Fusillo, R.H. and G.J. Powers, "Operating Procedure Synthesis Using Local Models and Distributed Goals," *Comput. Chem. Eng.*, 12, 1023 (1988).

Fusillo, R.H. and G.J. Powers, "Computer-Aided Planning of Purge Operations," *AIChE J.*, 34, 558 (1988).

Halasz, H., M. Hofmeister, and D.W.T. Rippin, "A Flexible Knowledge-Based Tool Kit in Batch Process Engineering," *AIChE Annual Meeting*, San Francisco, CA (1989).

Hofmeister, M., L. Halasz, and D.W.T. Rippin, "Knowledge-Based Tools for

Batch Processing Systems," *Comput. Chem. Eng.*, 13, 1255 (1989).

Huang, Y.L., G. Sundar, and L.T. Fan, "MIN-CYANIDE: An Expert System for Cyanide Waste Minimization in Electroplating Plants," *AIChE Spring National Meeting*, Houston, TX (1991).

Lakshmanan, R. and G. Stephanopoulos, "Synthesis of Operating Procedures for Complete Chemical Plants-I. Hierarchal, Structured Modelling for Nonlinear Planning", *Comput. Chem. Eng.*, 12, 985 (1988).

Lakshmanan, R. and G. Stephanopoulos, "Synthesis of Operating Procedures for Complete Chemical Plants-II. A Nonlinear Planning Methodology", *Comput. Chem. Eng.*, 12, 1003 (1988).

Lakshmanan, R. and G. Stephanopoulos, "Synthesis of Operating Procedures for Complete Chemical Plants-III. Planning in the Presence of Qualitative, Mixing Constraints", *Comput. Chem. Eng.*, 14, 301 (1990).

Lamirande, S. and P.R. Roberge, "Managing the Water Chemistry of a CANDU Reactor with an Expert System," *Canadian J. Chem. Eng.*, 68, 685, August (1990).

Musier, R.F.H. and L.B. Evans, "Batch Process Management," *Chem. Eng.*

Prog., **86** (6), 66 (1990).

O'Neil, M.V., "Implementing a Chemical Process Plant Expert System," *ISA Trans.*, **26**, 19 (1987).

O'Shima, E., "Safety Supervision of Valve Sequences", *J. Chem. Engng. Japan*, **11**, 390 (1978).

Rivas, J.R. and D.F. Rudd, "Synthesis of Failure-Safe Operations", *AIChE J.*, **20**, 320 (1974).

Rivas, J.R., D.F. Rudd, and L.R. Kelly, "Computer-Aided Safety Interlock Systems", *AIChE J.*, **20**, 311 (1974)

Shacham, M., O. Shacham, and K. Amaral, "Applications of Expert Systems in Chemical Engineering and Corrosion Control," *Corros. Rev.*, **7**, 151 (1987).

Shah, M.J., and C.M. Morely, "A Knowledge-Based Dynamic Scheduler in Distributed Computer Control," *Adv. in Instrum.*, **43**, 41 (1988).

Stephanopoulos, G., "The Scope of Artificial Intelligence in Plant-Wide Operations," in *Foundations of Computer-Aided Design*, pp. 505-555, G.V. Reklaitis and H.D. Spriggs, Editors, Elsevier, New York, NY

(1987).

Tomita, S., K.S. Hwang, E. Oshima, and C. McGreavy, "Automatic Synthesizer of Operating Procedures for Chemical Plant by Use of Fragmentary Knowledge," *J. Chem. Engng. Japan*, **22**, 364 (1989).

Verhulst, J.D., "Batch Plant Scheduling and Coordination," *Adv. in Instrum.*, **43**, 1403 (1988).

Weatherhill, T. and I.T. Cameron, "A Prototype Expert System for Hazard and Operability Studies," *Comput. Chem. Eng.*, **13**, 1229 (1989).

16.7 APPLICATIONS TO PROCESS MODELING AND SIMULATION

A. Introduction

Almost every engineer has used some form of modeling to predict system behavior. By "modeling," most engineers think of solving a complex differential equation that require sophisticated analytical and numerical methods. In practice, modeling encompasses a much wider range of areas. We summarize four different classifications of models below, based on how quantitative each model is.

- *Numerical models*- these are strictly quantitative, and are the ones that engineers are most familiar with. Applications include algebraic and differential equations, finite-difference and finite-element methods, matrix algebra, etc.

- *Order-of-magnitude models*- these are one step down from numerical models. We are not worried about the actual, precise value of a parameter. Its value within an order of magnitude is sufficient.

- *Qualitative models*- these models focus on the relationships between variables. They *can* be used in a quantitative sense to give directionality, but are still qualitative.

  For instance, the terms +, -, and 0 can be assigned to parameters in relationships, depending on how these parameters affect each other. As an example, as the relative speed of a pump motor goes up, its flow rate goes up. This is a "positive" qualitative relation. A price increase in the raw-material cost negatively impacts the profit margin, and is the example of a "negative" relation. If two parameters are independent and mutually exclusive, then there may be "zero" relation between the two.

In qualitative modeling, the main focus is on variable relationships rather than numerical values and their magnitudes.

- *Boolean models*- these are simply "yes-no" models. They state whether a relation exists or not, but cannot give the nature of the relation. There is no information on the sign or magnitude of variables.

An example of a Boolean model is an incidence matrix. If we have a set of process streams and a set of chemicals, we may write an incidence matrix identifying whether a certain chemical is in a particular process stream:

	Chemical One	Chemical Two	Chemical Three	Chemical Four
Process Stream 1	1	0	1	1
Process Stream 2	1	1	1	0
Process Stream 3	0	0	0	1
Process Stream 4	0	1	0	1

The "1" denotes that the chemical is present in the process stream; a "0" indicates that the chemical is not present.

The vast majority of modeling and simulation done in chemical engineering is numerical. Examples of numerical modeling include a finite-element analysis of viscous or viscoelastic flow, or a rigorous multicomponent vapor-liquid equilibrium calculation. With the advent of artificial intelligence, however, engineers have begun to recognize the value of qualitative techniques, and have begun to use, for instance, order-of-magnitude and qualitative modeling.

B. Applications

Most AI applications in chemical engineering modeling have been in qualitative simulation. The purpose of qualitative simulation is to explain observations by reasoning from physical to behavioral descriptions. Qualitative modeling attempts to explain what is going on "behind the scenes" through reasoning from more fundamental principles.

Qualitative simulation has been used to supplement rule-based and frame-based expert systems. They have found particular use in process control. The typical expert system performing process control with "shallow knowledge" knows the rules (i.e, heuristics), but does not understand the physical principles behind them. If a situation arises where the knowledge has not been explicitly encoded into the knowledge base, the typical rule-based system fails. Qualitative simulation can correct these drawbacks, turning a "shallow knowledge" system into a "deep knowledge" system (although certainly not as deep as a rigorous numerical simulation).

When applied to rule-based expert systems, qualitative simulation can be used for the following:

- *Derivation of partial conclusions*- knowing the cause-and-effect behind heuristics, partial conclusions can be drawn based on incomplete information.
- *Creation of predictive models*- knowing the cause-and-effect behind heuristics, predictive models may be possible. They can answer "what if"-type questions, predict potential safety hazards, and eliminate sub-optimal choices.
- *Creation of robust models for novel situations*- if a novel situation arises, qualitative simulation can make the rule-based system more robust; it can sufficiently supplement the system such

that it does not fail due to the absence of explicit code.

These can be particularly helpful in process control and fault-diagnosis applications, where both predictive modeling and the ability to handle novel situations is required.

Rich and Venkatasubramanian (1987) incorporate model-based knowledge into a diagnostic expert system. This system diagnoses faults in simple process units such as valves, pipes, and tanks. One difficulty they identified, however, is that deep-knowledge systems typically do not run fast enough to be used in on-line process control (compiled, shallow-knowledge systems can run fast enough for on-line process control and have been implemented commercially). Venkatasubramanian and Rich (1988) introduce a "two-tier" approach using object-oriented programming to coordinate both shallow and deep knowledge into a single system. The program runs in the "top tier" using shallow knowledge. If the shallow knowledge approach fails, then the "bottom tier" consisting of deep knowledge is utilized. Using this approach, we get the speed and convenience of shallow knowledge, and have to spend the time in deep-knowledge qualitative modeling only when required.

Kramer and Palowitch (1987) discuss the use of a qualitative model called a "sign-directed graph". This graph represents pathways of causality in a process. Nodes, representing process state variables, are connected with each other in a cause-and-effect network. Each connection has an attached (+) or (-) sign to indicate whether a positive or negative deviation will result. For instance, if a bypass valve is opened around a heat exchanger, the flow rate through the exchanger will go down. This is a negative deviation. With this representation, we can qualitatively model how disturbances will propagate through a process.

One challenge that needs to be overcome in qualitative modeling is that most models generate multiple solutions, many of which are spurious. It is difficult to eradicate ambiguities. Kramer and Oyeleye (1988) attempt to overcome this difficulty in qualitative modeling of steady-state continuous processes. This work predicts steady-state measurement patterns as a result of *a prior* process malfunction. They use the steady-state process equations qualitatively; the only numerical information is the signs (+) or (-) and relative values of parameters. They use a generate-and-test strategy. Hypotheses are generated and subjected to a test of model predictions versus process observations. Spurious solutions are eliminated, because they do not match process observations, or they fail to fulfill constraints.

Dalle Molle et al. (1988) discuss qualitative modeling of dynamic systems. Their work includes single-input and single-output (SISO) systems, as well as multiple-input and multiple-output (MIMO) ones. The process models are dynamic, i.e., unsteady state. They applied their technique to process control, although it can be used in any process-modeling task (see section 16.4 for a description of SISO and MIMO systems in process control). Their technique generates qualitative descriptions for open-loop responses in linear, nonlinear, and multivariable processes.

They also are able to model a closed-loop feedback process, and demonstrate qualitative offset in a proportional controller. The Qualitative SIMulation algorithm they use is called QSIM. At the AAAI-88 Workshop on AI in Process Engineering (Minneapolis, MN), Dalle Molle et. al. (1988) demonstrated qualitative modeling of a plasma etcher with QSIM.

Vinson et. al. (1990) have used qualitative modeling in fault diagnosis. Applying qualitative modeling to fault diagnosis is not new. However, in their application, Vinson et. al. propose deriving and rebuilding the qualitative models automatically. Thus, at least in theory, a wide range of faults can be diagnosed with no *a priori* models.

To build qualitative models automatically, Vinson et. al. use the Qualitative Process Theory (QPT) and its implementation, the Qualitative Process Engine (QPE) of Forbus (1984, 1990). QPE generates qualitative models consisting of causal constraints which explicitly represent the assumptions upon which the models are built. When observations of plant performance differ from what is expected, some assumption about the process is in error (e.g., the steam pressure is sufficiently high).

The QPE builds constraints, and these constraints are used to propagate discrepancies through the chemical process to decide what phenomena to remove or add. Assumptions associated with each phenomena are also removed or added. By adjusting these assumptions, we adjust the qualitative model.

The system developed by Vinson et. al. uses a hypothesis-and-test strategy. They demonstrate its use in fault diagnosis of a chemical reactor. This system, however, needs work before it could be implemented on-line. Specifically, the authors suggest: 1) translation of numeric sensor data to qualitative values, and 2) use of information concerning the order in which discrepancies are observed (to improve system speed). This system looks promising, but the on-line implementation is the ultimate challenge.

Order-of-magnitude reasoning (denoted O[M]) is another form of qualitative (or perhaps more accurately, semi-quantitative) modeling that has been under-utilized in chemical engineering. The only exception may be rough error analyses of complex numerical measurements to determine if the measured value is realistic.

Order-of-magnitude modeling is "semi-quantitative," lying between qualitative and quantitative modeling. Qualitative models reason with directionality (i.e., +, 0, or -, as explained in pure qualitative modeling), but frequently this is not quite good enough. Order-of-magnitude reasoning quantifies further. It not only contains the directionality of qualitative modeling, but the relative magnitudes of variables as well.

Most engineering problems include partial numerical knowledge about the process that qualitative models fail to capture. Examples are: knowledge of *absolute* value ranges, *relative* orders of magnitude, *approximate* numerical values and constraints. Order-of- magnitude reasoning in theory can capture this important information while still reasoning within an AI framework.

O[M] reasoning is fairly new; theoretical formalism is lacking in the field. Nevertheless, O[M] reasoning *has* been done systematically in chemical engineering. A classic example is solving a fluid-mechanics problem with Stokes flow. All inertial terms are eliminated from the flow equation based on O[M] reasoning. Although this example is one instance of the systematic O[M] reasoning, for the most part, O[M] reasoning has been a more intuitive, "gut-level" numerical scheme that lacks formalism and is used by engineers for general guidance.

Mavrovouniotis and Stephanopoulos (1988) attempted to theoretically formalize order-of-magnitude reasoning. They introduced seven primitive relations, shown in Table 16.2.

Table 16.2 Primitive relations of the O[M] formalism.

Relation	Verbal Explanation
A << B	A is much smaller than B
A -< B	A is moderately smaller than B
A ˜< B	A is slightly smaller than B
A == B	A is exactly equal to B
A >˜ B	A is slightly greater than B
A >- B	A is moderately greater than B
A >> B	A is much greater than B

These primitives allow us to reason about the relative orders of magnitude between numbers.

In addition to the primitive relations of Table 16.2, Mavrovouniotis and Stephanopoulos also introduce "fuzzy interval boundaries" as shown in Figure 16.6. Fuzzy interval boundaries are used to delineate relative

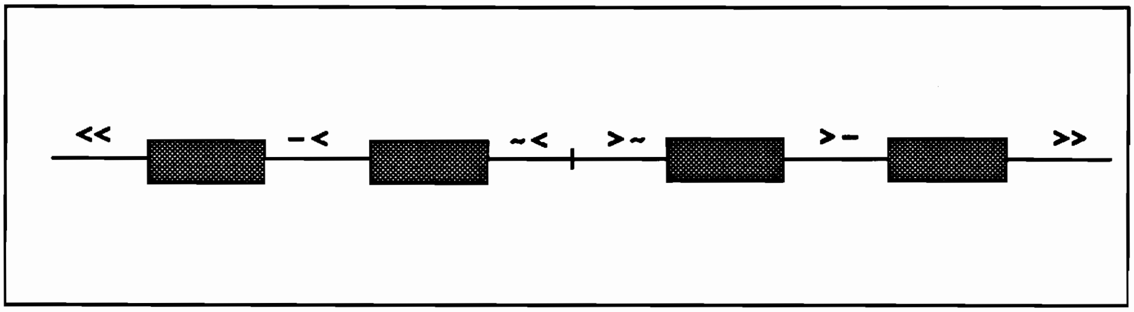


Figure 16.6. Fuzzy interval boundaries.

orders of magnitude between numbers. For example, let us consider the fuzzy interval between A -< B (A is moderately smaller than B) and A ˜< B (A is slightly smaller than B). The fuzzy interval is the interval of numerical values where A could be considered either moderately smaller, or slightly smaller than B. Because it is not clear which way to characterize the relationship between A and B, the interval is "fuzzy."

The tools introduced by Mavrovouniotis and Stephanopoulos are interesting, and enable us for the first time to do some systematic O[M] reasoning. Although just recently introduced, formal O[M] reasoning may be a technique that is able to capture engineering "common sense". It appears to be a good bridge between the rigorous quantitative approach and the more relation-oriented, qualitative approach. Time will tell if O[M] reasoning catches on.

C. Future Directions

Modeling and simulation has been, and will continue to be a supplement to rule- and frame-based expert systems. Modeling can turn shallow-knowledge systems into deep-knowledge ones. There is a lot of room for creativity in qualitative and order-of-magnitude modeling. This area has only recently seen in-depth investigation. Success in the area depends highly on the knowledge representation used to solve the problem. Importantly, the field is appreciating the fact that there are different levels of abstraction in performing qualitative or semi-quantitative reasoning.

Another application area for qualitative modeling is training. Trainees could "plug into" a qualitative simulator that emulates a complex process. Qualitative modeling within the simulator is excellent at capturing cause-and-effect relations. Learning the cause-and-effect relations in a process is a key to successful training.

As mentioned in section 16.4 on process control, one problem that continually crops up in qualitative modeling is the generation of multiple solutions. Efforts will probably continue in the direction of developing modeling techniques that eliminate spurious solutions early in the reasoning process. For example, let us consider the FALCON project. FALCON used modeling to guide on-line fault diagnosis. One difficulty realized was that competing causal influences could not be resolved in the absence of additional information. A hierarchical rule system was designed to resolve conflicts, but there was no guarantee that it can handle all situations. Development of better models could resolve this problem.

Finally, because qualitative modeling is just that -- qualitative, it in many ways lacks the depth and fails to capture the finer details of problems like quantitative modeling does. Consequently, the best models are those that combine both qualitative and quantitative information into a single system. A unified approach to meshing qualitative and quantitative modeling into a single system is certainly in need.

D. References and Further Reading

Some key references are Bobrow (1985), Kramer and Palowitch (1987), Mavrovouniotis and Stephanopoulos (1988), Rich and Venkatasubramanian (1987), and Waters and Ponton (1989).

Bhat, N. and T.J. McAvoy, "Dynamic Process Modeling via Neural Networks," *AIChE Annual Meeting*, San Francisco, CA, November (1989).

Bobrow, D.G., *Qualitative Reasoning about Physical Systems*, MIT Press, Cambridge, MA (1985).

Cheung, J.T.Y. and G. Stephanopoulos, "Representation of Process Trends: I. A Formal Representation Framework; and II. The Problem of Scale and Qualitative Scaling," *Comput. Chem. Eng.*, **14**, 511 and 541 (1990).

Dalle Molle, D.T., B.J. Kuipers, and T.F. Edgar, "Qualitative Modeling and Simulation of Dynamic Systems," *Comput. Chem. Eng.*, **12**, 853 (1988).

Dalle Molle, D.T., T.F. Edgar, and B.J. Kuipers, "Modeling Chemical Processes with Unknown Parameters," *ISA Adv. Instrum.*, **43**, 59 (1988).

Dalle Molle, D.T., T.F. Edgar, and I. Trachtenberg, "Qualitative Modeling of a Plasma Etcher," *AAAI-88 Workshop*, Minneapolis (1988)

Forbus, K., "Qualitative Process Theory," *Artificial Intelligence*, **24**, 85 (1984).

Forbus, K.D., "Qualitative Process Engine," in *Readings in Qualitative Reasoning About Physical Systems*, pp. 220-235, Morgan Kaufman, San Mateo, CA (1990).

Kramer, M.A., "Malfunction Diagnosis Using Quantitative Models with Non-Boolean Reasoning in Expert Systems," *AIChE J.*, **33**, 130 (1987).

Kramer, M.A., and B.L. Palowitch, Jr., "A Rule-Based Approach to Fault Diagnosis Using the Signed Directed Graph," *AIChE J.*, **33**, 1067 (1987).

Kuipers, B.J., "Qualitative Simulation," *Artificial Intelligence*, **29**, 289 (1986).

Lai, C.H., "Qualitative-Model Simplification: Multistage Separation Processes," M.S. Thesis, National Taiwan Institute of Technology, Taipei, Republic of China, July (1990).

Mavrovouniotis, M.L. and G. Stephanopoulos, "Reasoning with Orders of Magnitude and Approximate Relations," *Proc. of AAAI-87*, p.626, American Association for Artificial Intelligence, Menlo Park, CA (1987).

Mavrovouniotis, M.L. and G. Stephanopoulos "Formal Order-of-Magnitude Reasoning in Process Engineering," *Comput. Chem. Eng.*, **12**, 867 (1988).

Mavrovouniotis, M.L., George Stephanopoulos, and Gregory Stephanopoulos, "Formal Modeling of Approximate Relations in Biochemical Pathways," *Biotech. and Bioeng.*, **34**, 196 (1989).

McCroskey, P.S., "A Knowledge-Based Approach for the Development of Complex Kinetic Mechanisms," PhD Dissertation, Carnegie-Mellon University, Pittsburgh, PA (1988).

O.O. Oyeleye and M.A. Kramer, "Qualitative Simulation of Chemical Process Systems: Steady-State Analysis," *AIChE J.*, **34**, 1441 (1988).

Rich, S.H. and V. Venkatasubramanian, "Model-based Diagnostic Expert Systems for Chemical Process Plants," *Comput. Chem. Eng.*, 11, 357 (1987).

Venkatasubramanian, V. and S.H. Rich, "An Object-Oriented Two-Tier Architecture for Integrating Compiled and Deep-Level Knowledge for Process Diagnosis," *Comput. Chem. Eng.*, **12**, 903 (1988).

Vinson, J., S. Grantham, and L. Ungar, "Diagnosis Using Automatic Rebuilding of Qualitative Models in Chemical Plants," *AIChE National Meeting*, Chicago IL (1990).

Waters, A., and J.W. Ponton, "Qualitative Simulation and Fault Propagation in Process Plants," *Chem. Eng. Res. Dev.*, **67**, 407, July (1989).

16.8 APPLICATIONS TO PRODUCT DESIGN, DEVELOPMENT, AND SELECTION

A. Introduction

Expert systems have found use in industry in product design and development. In many ways, they have left the research stage and have gone into the commercial stage. Product design, development and selection can be a particularly valuable area for expert-system applications.

At first glance, one may think that expert systems would not perform well in a product-development environment. The reasoning is that in product development we are by definition in novel situations. Since expert systems are based, for the most part, on experiential knowledge, they do not perform well in novel situations and therefore would not perform well in product development.

This reasoning is only partially true. While there are novel situations in product development, there are also a great number of applied steps too. For example, let us consider the package development stage for consumer goods. Once a product formulation exists, we need to develop a package that will house the material, and achieve all technical and aesthetic objectives. For instance, we may need a package with a specific water-vapor transmission rate, aromatic resistance, or solvent resistance. This is an applied step done by experts that is a potential expert-system application.

Thus, to use a knowledge-based approach in product design or development, we may use it for *applied* or *routine* steps. The systems perform well in a procurement, sizing, or specification mode, e.g., an expert system for piping design. But if the system is utilized in novel product development, it is typically used as a "rough cut" product design or development tool. The system generates an initial prototype, and a human expert then modifies and optimizes the product design.

While expert-system applications have grown quickly in product development, there is unfortunately little that is published or known publicly. This is mainly due to the proprietary nature of the work.

B. Applications

CAPS (Venkatasubramanian, 1988) is an expert system for plastics selection. The user is questioned about usage of the final product, and CAPS recommends classifications of polymers to consider. CAPS includes an explanation facility if the user wants to know why it is asking the question. It also tests the consistency of its results before releasing them to the user. CAPS is a forward-chaining rule-based system written in OPS5, GCLISP version. It is a forward-chaining language which uses production rules that "fire" when a certain set of conditions is met. CAPS was implemented on an IBM PC AT with 3 MB of memory.

CAPS includes knowledge about physical properties in the following areas:

- *Environmental*- chemical resistance to solvents, weak acids and bases, strong acids and bases, salts, oxidizers, etc.
- *Mechanical*- tensile strength, impact strength, creep properties
- *Temperature*- use in low-, medium-, or high-temperature applications, as well as low-high "swing" usage.
- *Electrical*- dielectric strength and arc resistance,
- *Miscellaneous*- transparency, desired colors, and shrinkage.

Using these classifications, CAPS recommends the engineer to consider plastics from the following categories:

- ABS resin
- Acetal
- Acrylic
- Cellulosic
- Fluorocarbon
- Polyallomer
- Polyamide-imide
- Polyaryl-sulfone
- Polycarbonate
- Polyester
- Polyimide
- Polyphenylene oxide
- Polystyrene
- Polysulfone

In its current form, CAPS recommends a *set* of polymers to consider. Its knowledge base is not developed enough to recommend the *best* polymer classification. Of course, much more material-science knowledge could be added too, such as flexural modulus, copolymers (block, graft, random, etc.), uses of filler, etc. In addition to the material-science knowledge enhancements, some useful computer-science improvements could be:

(1) giving the user the ability to rank the importance of properties;

(2) giving the user the ability to ask "what if" certain requirements were relaxed or included; and

(3) including a knowledge-building facility to allow the user to build his own knowledge base.

DECADE (Design Expert for CAtalyst DEvelopment) is a prototype expert system developed at Carnegie-Mellon University for catalyst selection (Banares-Alcantara, et al. 1987, 1988). It is a *hybrid* expert system, i.e., it uses several modules written in different languages but linked through a *blackboard* system. The blackboard is an overall umbrella that controls access to these different languages. Three different languages are used:

- <u>OPS5-</u> for forward-chaining production rules.
- <u>SRL 1.5-</u> for frame representation.
- <u>FranzLISP-</u> for external functions.

The idea of a hybrid system using a blackboard is to utilize the strengths of each language for the sub-problems involved in catalyst selection. OPS5 is ideal for developing and using forward-chaining rules; SRL 1.5 and FranzLISP are not. By using three languages, we are not "saddled" with the inherent deficiencies of a single language.

On the downside, however, it is more difficult for the user to maintain continuity in a hybrid system. In addition, the programmer needs to test for data-type consistency when jumping from one module to the next. For example, a data object may be viewed as a string in one language and an atom in another. Proper classification of data type is essential for correct interfacing between modules.

On the chemical engineering side, DECADE's knowledge base is limited. It exclusively analyzes the Fisher-Tropsch process, also known as the "syngas" process. The Fisher-Tropsch process is the hydrogenation of carbon monoxide to form a wide variety of products. Its scope is shown in Figure 16.7.

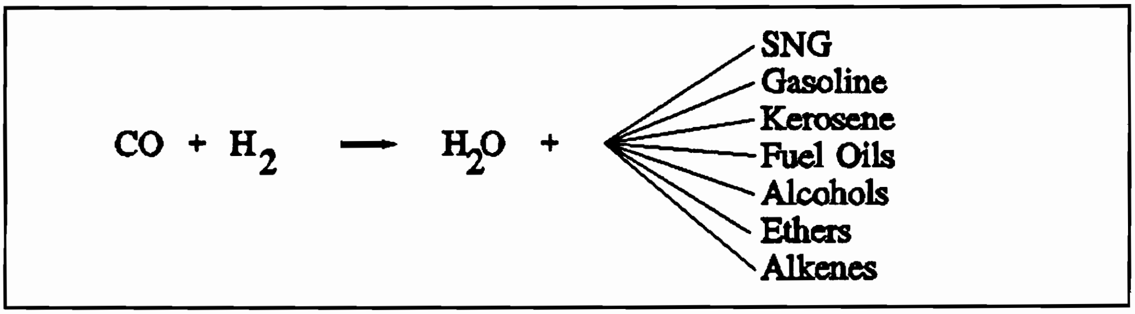


Figure 16.7. The Fisher-Tropsch process.

DECADE proposes a set of catalysts with high probability of being useful for a specified Fisher-Tropsch reaction. It will also give the conditions the catalyst should operate. DECADE uses a depth-first search to arrive at answers. Knowledge sources that DECADE uses to arrive at answers are:

- Specification of the reaction;
- Thermodynamic feasibility;
- Classification of the reaction (e.g., hydrogenolysis);
- Prediction of surface steps for the reaction;
- Catalyst selection; and
- Explanation.

In the catalyst-selection category, DECADE uses three levels of abstraction. It can choose a catalyst based on:

1. Known work from published results that is encoded explicitly in the knowledge base;
2. Classifying the reaction from general to specific classes, and recommending catalysts known to perform in all of these levels of classification; and
3. Determining both required and unwanted surface steps (adsorption, desorption, disassociation, addition) for the reaction, and selecting materials known to catalyze the required steps and to suppress the unwanted ones.

It probably would not be accurate to say that DECADE possesses deep knowledge. The second and third levels of abstraction definitely go beyond the shallow knowledge of the first level, but they do not reason from basic principles either. Therefore, it may be accurate to say the second and third levels of abstraction represent "medium level" knowledge.

PASS (Pump Application Selection System), by Venkatasubramanian (1988), is an expert system for pump selection. The goal of PASS is to choose the:

- pump type,
- pump drive, and
- pump housing material or liner

that meets appropriate process needs. PASS bases its decisions on:

- the volumetric flow rate of liquid to be pumped,
- the physical properties of the liquid (e.g., whether the liquid is corrosive)
- the head against which the liquid is to be pumped,
- the nature of the power supply,
- the conditions under which the pump will operate (e.g., intermittent or continuous), and
- cost and mechanical efficiency.

PASS operates by first evaluating the pump head (P) versus volumetric flow rate (Q) requirements for the pump. Based on the relation of Q vs. P, PASS determines whether to use single suction, double suction, or multiple stage pumps. It then refines its pump selection by asking the following questions:

1. Fluid Properties
   a) what is the kinematic viscosity?
   b) does the fluid contain coarse solids?
   c) what is the viscosity-shear rate relation? Is the fluid Newtonian?
   d) does the fluid have lubricating properties?

2. Process Requirements
   a) will the flow rate vary?
   b) is leakage tolerable?
   c) will fluid composition vary?
   d) must pulsation be avoided?
   e) will the pump operate under negative NPSH (net positive suction head)?

Based on this information, PASS then selects a pump from one of the following categories: rotary screw pumps (single or twin), rotary gear pumps, positive-displacement diaphragm pumps, displacement plunger pumps, displacement piston pumps, and dynamic centrifugal pumps. Within each of these types of pumps, there are a number of subsections. For example, within rotary pumps, we may have single-rotor or multiple-rotor pumps. Within multiple-rotor pumps, we may have gear, lobe, screw, or circumferential piston pumps.

PASS is a forward-chaining rule-based expert system. It uses production rules that "fire" when a given set of information exists. It is a *data-driven* system, and is written in OPS5. It is a very good example of the use of an expert system for selection and design.

Hulthage et. al. (1990) have developed ALADIN, a knowledge-based approach to alloy design. ALADIN is a system that aids in the design of aluminum alloys for aerospace applications. ALADIN uses composition and thermal- and mechanical-process steps along with product objectives to make decisions and design alloys. Product objectives include fracture toughness, stress resistance, corrosion resistance, ductility, damage-tolerance, modulus, chemical microstructure, and means of processing.

ALADIN provides a decision-support environment for expert designers to quickly and efficiently explore product alternatives. ALADIN uses both qualitative and quantitative knowledge to achieve its objectives. Its knowledge-based is organized hierarchically, with three primary categories: composition, properties, and processing. In addition, ALADIN includes a hierarchy on *chemical microstructure*, a first for a materials engineering expert system.

ALADIN uses a hypothesis-and-test search strategy. It has three levels of abstraction:

1. *The meta level*- the first level of abstraction that controls the entire program, plans the design process, and established priorities;

2. *The structure-planning level*- formulates targets at the phase and microstructure level to realize desired macroscopic properties;

3. *The implementation plane*- encompasses chemical composition, thermal processing, and mechanical processing.

Overall, ALADIN is an excellent practical example of a sophisticated expert system for product design.

C. Future Directions

Developing expert systems for product-development applications is still an expensive endeavor. At this point in time, mostly large businesses with sizable sales volumes have the economies of scale to afford and invest in these expert systems. This restriction is changing, however, as more powerful hardware becomes available at lower cost.

Expert systems for product design and development are inherently application specific. Therefore, they can be victims of the "knowledge engineering bottleneck," i.e., the painfully slow and expensive process of building the knowledge base for each application. Both the hardware and software companies in the expert-system industry are keenly aware of this problem. The "knowledge engineering bottleneck " drives the development costs up for expert systems so high, that the payoff is not there for many applications. There will certainly be a focus in the AI industry to automate and improve the efficiency of the knowledge-engineering stage. This will lower developmental cost, and expand usage into areas that were previously out of reach.

D. References and Further Reading

Some key references are Hulthage et. al. (1990) and Venkatasubramanian (1988a and b).

Banares-Alcantara R., A.W. Westerberg, E.I. Ko, and M.D. Rychener, "DECADE. A Hybrid Expert System for Catalyst Selection-I. Expert System Considerations," *Comput. Chem. Eng.*, 11, 265 (1987).

Banares-Alcantara R., A.W. Westerberg, E.I. Ko, and M.D. Rychener, "DECADE. A Hybrid Expert System for Catalyst Selection-II. Final Architecture and Results," *Comput. Chem. Eng.*, 12, 923 (1988).

Hulthage, I.A.E., M.S. Fox, M.D. Rychener, and M.L. Farinacci, "The Architecture of ALADIN: A Knowledge-Based Approach to Alloy Design," *IEEE Expert*, pp. 56-63, August (1990).

Joback, K.G., and G. Stephanopoulos, "Designing Molecules Possessing

Desired Physical Property Values," p.363, *Foundations of Computer-Aided Process Design*. p.79, J.J. Siirola, I.E. Grossman, and G. Stephanopoulos, editors, CACHE Corp., Austin, TX, and Elsevier, New York, NY (1990).

Mavrovouniotis, M.L., George Stephanopoulos, and Gregory Stephanopoulos, "Computer-Aided Synthesis of Biochemical Pathways," *Biotech. and Bioeng.*, in press (1990).

Seressiotis, A. and J.E. Baily, "MPS: An Artificial Intelligence Software System for the Analysis and Synthesis of Metabolic Pathways," *Biotech. and Bioeng.*, 31, 587 (1988).

Venkatasubramanian, V., "PASS: An Expert System for Pump Selection," *Knowledge-Based System in Chemical Engineering, Vol. II*, CACHE Case Study Series, CACHE Corporation, Austin, TX (1988a).

Venkatasubramanian, V., "CAPS: An Expert System for Plastics Selection," *Knowledge-Based System in Chemical Engineering, Vol. III*, CACHE Case Study Series, CACHE Corporation, Austin, TX (1988b).

16.9 CONCLUSIONS

In this overview chapter, we have seen expert systems applied in many ways to chemical engineering. They have been applied to:

- fault diagnosis,
- process control,
- process design,
- planning and operations,
- modeling and simulation, and
- product design and development.

Applications in these and other areas of chemical engineering are growing daily.

Unfortunately, at the time of this writing, developing an expert system is still an expensive endeavor that takes a lot of time. Thus, most of the commercially successful expert systems that exist today are for those types of problems that fit very well with the expert-system approach (defined in section 16.2 A, B, and C). For economic reasons, many expert systems are either still in the research departments as prototype systems, or are being applied commercially to areas that have a big payoff.

The current "frontier" in expert-system development in chemical engineering is a two-pronged effort:

(1) *Improve performance*- develop software techniques and knowledge representations that improve both the depth and the breadth of expert-system performance. In addition, there is a great need to develop excellent user-interface tools. This will improve the return on investment by increasing the payoff.

(2) *Reduce developmental cost*- improve hardware, develop software tools for rapid knowledge engineering, develop systems with a broader application area (most right now are application-specific), such that the system-development cost is reduced. A lower system-development cost will open up expert applications to more areas.

If progress can be made in these areas, expert systems will become more useful and pragmatic chemical engineering tools with increasing scope and usage.

17

INTRODUCTION TO ARTIFICIAL NEURAL NETWORKS

This chapter introduces artificial neural networks (ANNs). We first discuss what makes up an ANN, and contrast ANNs with empirical modeling and expert systems. We then move on to the fundamentals of neural computing, and discuss aspects of ANN learning. Finally, we discuss applications, and summarize the strengths and limitations of ANNs.

17.1 INTRODUCTION

A. The Essence of Artificial Neural Networks (ANNs)

An artificial neural network (ANN), also called a "neural net," is a computational tool having AI origins. It differs from conventional AI applications, though, and consequently it deserves separate treatment. Expert systems programmed in LISP and Prolog use "classical" symbolic processing. The programs manipulate symbols, such as atoms and lists, to solve problems. ANNs, on the other hand, use *subsymbolic* processing.

But what is subsymbolic processing? To understand subsymbolic processing, we first need to examine the history behind ANNs.

1. Subsymbolic Processing

The term "artificial neural network" resulted from AI research that attempted to understand and model brain behavior. Most neurologists believe that true intelligence goes beyond symbolic processing. Some *subsymbolic* manipulation exists that manifests itself eventually as macroscopic, intelligent, symbolic behavior.

In the human brain, neurons within the nervous system interact in a complex fashion. The human senses detect stimuli, and send this "input" information (via neurons) to the brain. Within the brain, neurons are excited and interact with each other. Based on the input, a conclusion is drawn, and an "output" is sent from the brain in the form of an answer or response. The interaction between neurons is not seen by anyone, but manifests itself as identifiable intelligent behavior.

Is it possible to develop the same type of structure for a computer modeling of intelligent behavior? Certainly. Neurologists and AI researchers have proposed a highly interconnected network of "neurons," or nodes, for this purpose. We input information into a network of nodes. The nodes mathematically interact with each other in a fashion unknown by the user. Eventually, based on the input, an output arises that maps the expected, macroscopic, input-output pattern.

Expert systems operate symbolically, on a *macroscopic* scale. They use symbolic processing, require knowledge of relationships, and do not care how these relationships develop. ANNs however, operate sub-symbolically on a *microscopic* scale. The interactions between nodes is well-defined and adjusted until the desired input-output relationships are properly matched. Thus, ANNs are intimately concerned with how relationships develop. The *microscopic*, subsymbolic processing that occurs in ANNs manifests itself as *macroscopic*, symbolic, and intelligent behavior.

The interconnection of nodes form the artificial neural network (ANN), as shown in Figure 17.1. All ANNs have an *input layer*, at least one

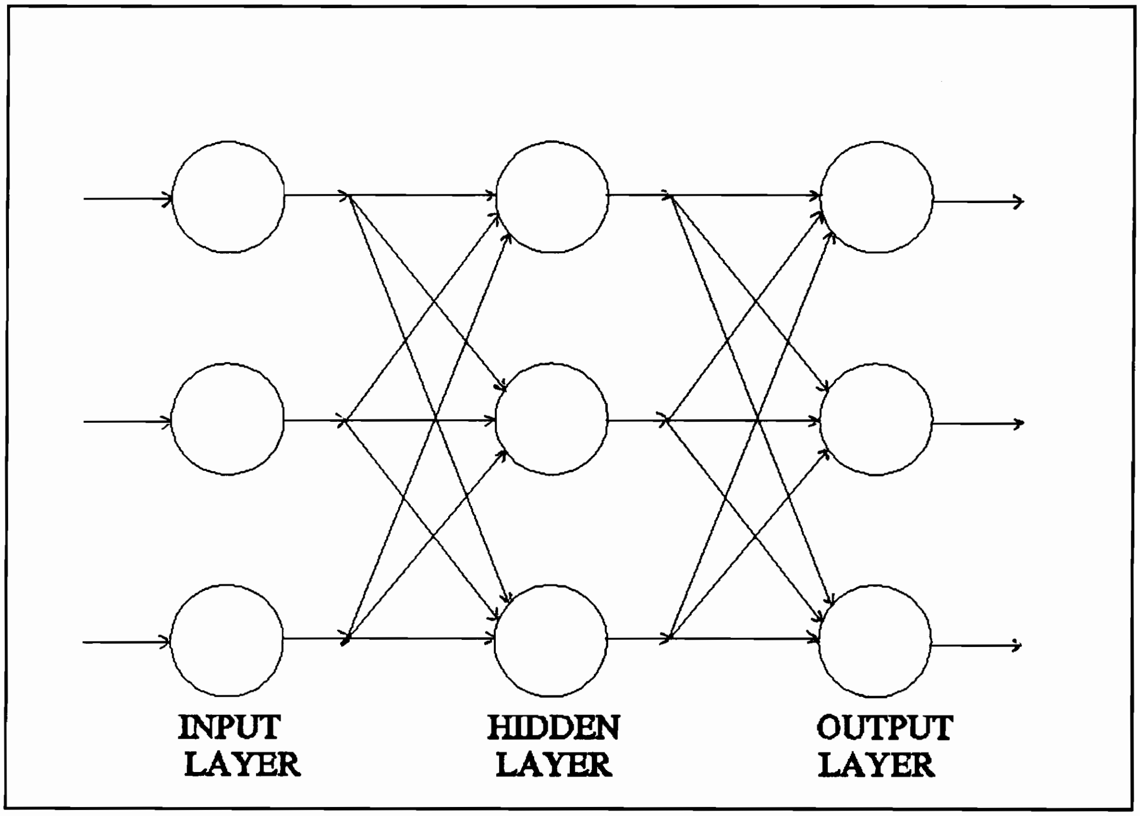


Figure 17.1. An ANN with one hidden layer.

(and sometimes more) *hidden layer*, and an *output layer*. Each layer is essential to the success of the ANN. An ANN can be viewed as a "black box" into which we send a specific input to all the nodes in the input layer. The ANN processes this information through its interconnections between nodes (the entire processing step is hidden from us). Finally, the ANN gives us a final output, which results from the nodes on the output layer. We can then summarize the purpose of each layer as follows:

- *Input Layer*- receives information from an external source, and passes this information into the ANN for processing.

- *Hidden Layer*- receives information from the input layer, and "quietly" does all of the information processing. The entire processing step is hidden from view.

- *Output Layer*- receives processed information from the ANN, and sends the results out to an external receptor.

When the input layer receives information from an external source, it becomes "activated" and emits signals to its neighbors. The neighbors receive excitation from the input layer, and in turn emit an output to their neighbors. What results is a patterned activation that eventually manifests itself in the output layer. Depending on the strength of the connections, signals can excite *or* inhibit the nodes.

One important characteristic of ANNs is that within the network, the processing is numerical, *not* symbolic (although the results *can* manifest themselves symbolically - hence the name subsymbolic). The network retains information through:

(1) the magnitudes of the signals passing through the network, and
(2) the connections of nodes with neighbors.

Because information passed is numerical, the ANN can be used as a multivariable empirical modeling tool.

2. Operating an ANN

To operate an ANN, we must go through three phases:

- the *training phase*,
- the *recall phase*, and
- the *generalization phase*.

In the training phase, we repeatedly present a set of input-output patterns to the ANN. We adjust the weights of all the interconnections between nodes until the specified input yields the desired output. Through these activities, the ANN "learns" the correct input-output response behavior.

In ANN development, the training phase is typically the largest and most time-consuming step. After the training phase, we move to the recall and generalization phases. In the recall phase, we subject the ANN to a wide array of input patterns seen in training, and introduce adjustment to make the system more reliable and robust. During the generalization phase, we subject the ANN to novel input patterns, where the system hopefully performs properly.

The training or learning phase is critical to the success of the ANN. We use backpropagation in this stage to correct errors. Backpropagation is a numerically intensive technique, and there are many different ways to perform backpropagation to teach the ANN how to respond.

Researchers in mathematics, statistics, computer science, and engineering have all suggested algorithms, generally quite complex mathematically, for backpropagation.

Basically, however, almost all forms of backpropagation take the following steps:

(1) Enter a specific input and measure the actual output.

(2) Compare the actual output to the desired output; calculate a quantitative error based on the input-output analysis.

(3) Iteratively minimize the error (or the average squared error) by adjusting the connectivity strength between nodes.

(a) Begin at the output nodes and adjust their weights.

(b) Propagate "backwards" to the layer adjacent to the output layer. Calculate errors in that layer and adjust weights.

(c) Continue this backward propagation (from the output side of the network towards the input side) until all errors are calculated and weights are adjusted.

The intense mathematics comes in step three, when we calculate errors and adjust connectivity strengths.

3. Properties of ANNs

ANNs have a number of properties that make them advantageous over other computational techniques, as described below.

(1) *Information is distributed over a field of nodes.* This provides greater flexibility than symbolic processing, where information is held in one fixed location.

(2) *ANNs have the ability to learn.* If an error or a novel situation occurs that creates inaccurate system results, we can use "backpropagation" to correct it. During backpropagation, we adjust the strengths of the signals emitted from the nodes until the error disappears. At that point, the system has effectively "learned." When the system encounters that situation in the future, the ANN will model it properly.

In symbolic processing, such as a fault-diagnosis expert system, learning has proven a very difficult roadblock. Typically, the system is unreliable if it faces a novel situation. The programmer must develop a new set of rules to correct the situation. Consequently, using a system that has the ability to learn significantly increases efficiency.

(3) *ANNs allow extensive knowledge indexing.* Knowledge indexing is the ability to store a large amount of information and access it in a simple manner. In a symbolic program using rule-based knowledge, knowledge indexing can be awkward. The information may not be easily

accessible, and a sizable amount of time may be wasted to retrieve that information. For this reason, in symbolic systems, we utilize frame-based systems when extensive knowledge indexing is required.

An ANN provides inherent knowledge indexing. It can recall, for example, diverse amounts of information associated with a chemical name, a process, or a set of process conditions. The knowledge is retained in the network via two means: 1) the connections between nodes, and 2) the weights of these connections. Because of so many interconnections, the ANN can index and house large amounts of information corresponding to the interrelations between variables.

(4) *ANNs are better suited for processing noisy, incomplete, or inconsistent data.* No single node within an ANN is directly responsible for associating a certain input with a certain output. Instead, each node encodes a *microfeature* of the input-output pattern. The concept of microfeature implies that each node affects the input-output pattern only *slightly*. Only when we assemble *all* the nodes together into a single coordinated network, can these microfeatures map the macroscopic input-output pattern.

Other computational techniques do not include this microfeature concept. In empirical modeling, for instance, most models rely heavily on each variable used. Consequently, if the value of one variable is off, the model most likely will yield inaccurate results. With the microfeature concept, however, if the value of one variable is off, the model *will not* be affected substantially.

In addition to the microfeature concept for ANNs, the signals sent to and from nodes are continuous functions. Consequently, the ANN can deduce proper conclusions, even from noisy, incomplete, or inconsistent input signals.

(5) *ANNs mimic human learning processes.* Most human learning and problem-solving occurs by trial and error. For example, if a piece of equipment is not operating correctly, we observe its symptoms and recommend corrective actions. Based on the results of this action, we recommend additional corrections. This process continues until we properly correlate symptoms with corrective actions, and the machine operates correctly.

ANNs operate in the same fashion. We can *train* them by iterating and adjusting the strength of the connectivity between the nodes. After numerous iterations and adjustments, the ANN can properly predict cause-and-effect relationships.

4. Applications of ANNs

ANNs have found use in a number of areas. They serve as numerical modeling tools in intelligent process control. When combined with symbolic information processing, they function as support in "traditional" expert systems. In this role, the ANNs effectively operate as symbolic-to-numeric or numeric-to-symbolic converters. Finally, ANNs act as a stand-alone tool for process-fault diagnosis.

B. Artificial Neural Networks and Empirical Modeling

Consider the ANN shown in Figure 17.1. Each layer -- inner, hidden, and outer -- has three nodes. Thus, we have a total of eighteen connections, and eighteen weights to adjust to train the network. An engineer may say, "Hold on here! If you give me eighteen variables, I can curve-fit almost anything. This artificial neural network is nothing but empirical modeling, which has been around for more than fifty years. You are just doing some fancy curve-fitting."

There is truth in that claim (Mah, 1991). An ANN *is* an empirical modeling tool, and it does operate by "curve-fitting." However, some notable differences exist between ANNs and more typical empirical models. As a result, ANNs offer distinct advantages in some areas, as explained below, but have limitations in other areas (see section 17.4).

First, ANNs have a better *filtering capacity* than empirical models because of the *microfeature* concept (see section 17.1A3 for a detailed description of microfeature). In other words, each node encodes a microfeature of the overall input-output pattern; that is, each node affects the input-output pattern only slightly. ANNs are also *massively* *parallel*. Each node operates independently of the other nodes. We can view each node as a processor in its own right, and these processors all operate in parallel. As a result, the network does not depend on a single node as heavily as, for instance, an empirical model depends on an independent variable. Because of this microfeature concept, ANNs have a filtering capacity and can generally perform better than empirical models with *noisy* or *incomplete* data.

A second advantage that ANNs have over empirical models is the ability to *adapt*. ANNs have specified *training algorithms*, where we adjust connection weights between nodes until the desired input-output pattern is attained. If conditions change such that the ANN model performance is inadequate, we can subject the ANN to further training under these new conditions to correct performance. In addition, we can design the ANN to periodically update its input-output performance, resulting in a continuous, "on-line", self-correcting model. Typical empirical models do not have this ability.

Third, ANNs are truly multiple-input and multiple-output (MIMO) systems. Most empirical modeling tools map one, or at most two or three dependent variables. ANNs can map many independent variables with many dependent variables. Consequently, ANNs perform better at *pattern recognition* than traditional empirical modeling systems.

To summarize, ANNs *are* an empirical modeling tool. Based on their structure, however, they have some unique properties. In section 17.2, we explain he fundamentals of ANNs; in section 17.3, we focus on applications. Finally, in section 17.4, we discuss advantages and limitations of ANNs from an application standpoint.

C. Artificial Neural Networks and Expert Systems

Sections 17.1A and B have briefly described ANNs from the perspective of subsymbolic processing, and compared subsymbolic to symbolic processing. Expert systems, which are grounded in symbolic processing, have seen tremendous growth in the past ten years. For problems well suited to expert-system development, these systems have been successful and have proven their worth. However, expert systems cannot solve every problem, and ANNs provide an effective alternative. Questions then arise concerning expert systems and ANNs. Will ANNs replace expert systems? Where do the two fit in?

ANNs probably will not replace expert systems. Instead, the two systems will most likely *complement* each other. To answer why and how, we must first understand the nature of problem-solving.

When a problem arises, experienced engineers can instantaneously diagnose root-causes and recommend corrections. They do not require long, drawn-out analysis and reasoning -- they simply know the answers, and are usually correct! In this type of problem-solving mode, engineers are operating by principles of *pattern recognition.* Based on the response pattern, they generate quick, spontaneous, remedies with little or no conscious effort.

However, this rapid response approach to problem-solving sometimes fails. In that case, engineers turn to a more in-depth analysis. This analysis requires more data and more time, but, in the end, usually yields favorable results.

Similarly, we can envision a complementary ANN-expert system that incorporates both types of problem-solving (Hendler, 1989 and 1991). Within a given domain, the trained ANN can perform pattern recognition and solve problems very quickly. Importantly, we design the system to use the ANN only when the data are within a specific domain. Outside this domain, the system defers to a more in-depth expert-system analysis. Perhaps a second ANN would convert the quantitative data into qualitative information. The expert system would then take over, and might ask for additional information. It then reasons through the problem and draws conclusions. In this way, the ANN and expert system work together and take advantage of their individual strengths to solve problems accurately and efficiently.

17.2 FUNDAMENTALS OF NEURAL COMPUTING

This section introduces the fundamentals of neural computing. We discuss the structure of the nodes that make up the ANN, as well as the nodes' interconnections. We then discuss the *topology* of ANNs, i.e., how the nodes are interconnected, then move on to training ANNs, and the different ways that ANNs can "learn." Finally, we elaborate on these training techniques with examples.

A. Components of a Node

The foundation of an ANN is the artificial neuron, or node (sometimes called *neurode*). In most scientific and engineering applications, this node is called a *processing element* (PE). Figure 17.2 shows the anatomy of a processing element. The PEs are the elements in the ANN where most calculations are performed.

1. Inputs and Outputs

The first element in the jth PE is an *input vector*, $\underline{a}$, with components a_1, a_2, ..., a_i, ..., a_n. The node manipulates these inputs, or activities, to give the output, b_j. This output can then form the part of the input for other PEs.

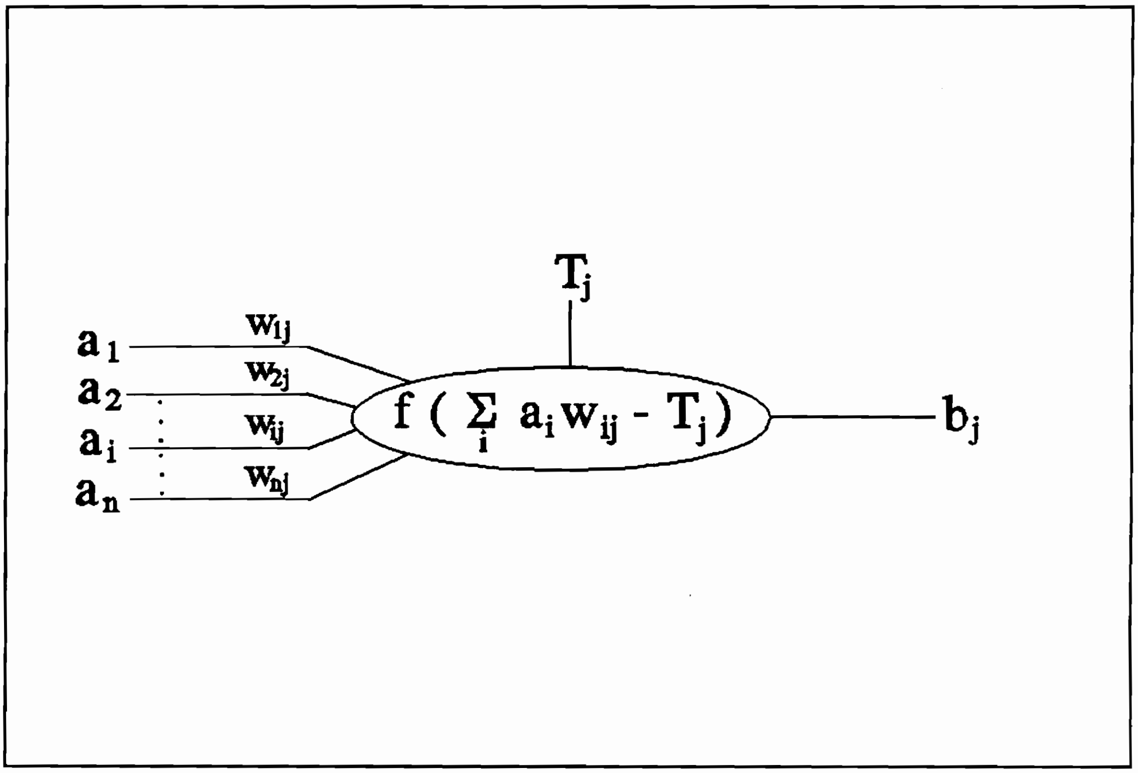


Figure 17.2. The anatomy of the jth processing element (PE).

2. Weight Factors

What determines the output from the PE? Certainly, component values of input vector a have an effect. However, some additional components of the PE also affect b_j. One such component is the *weight factor*, w_{ij}, for the i-th input a_i corresponding to the jth node. Every input is multiplied by its corresponding weight factor.

For example, let us consider the processing element six. The first input into the element is a_1. Multiplying this input by the corresponding weight factor gives a_1w_6. The PE uses this *weighted input* to perform further calculations.

Importantly, the weight factors can have an inhibitory or excitatory effect. If we adjust w_{ij} such that a_iw_{ij} is positive (and preferably large), we tend to excite the PE. If a_iw_{ij} is negative, it inhibits the node. Finally, if a_iw_{ij} is very small in magnitude relative to other signals, it (i.e., a_iw_{ij}) will have little or no effect on the node.

3. Internal Thresholds

The internal threshold for the jth PE, denoted T_j, controls activation of the node. The node calculates all its a_iw_{ij}'s, sums the terms together, and then calculates the total activation by subtracting the internal threshold value:

$$\text{Total Activation} = \sum_{i=1}^{n} (w_{ij} \; a_i) - T_j$$

If T_j is large and positive, the node has a high internal threshold, which inhibits node-firing. Conversely, if T_j is zero (or negative, in some cases), the node has a low internal threshold, and that low value of T_j excites node-firing.

Some, but not necessarily all, PEs may have an internal threshold level. If no internal threshold is specified, we assume T_j to be zero.

4. Functional Forms

The PE performs calculations based on its input. It takes the dot product of vector $\underline{a}$ with vector $\underline{\mathbf{W}}_j = [W_{1j}, W_{2j}, \ldots, W_{nj}]^T$, subtracts the threshold T_j, and passes this result to a functional form, f(). Thus, the node calculation is:

$$f(\underline{W_j} \cdot \underline{a} - T_j) = f\left(\sum_{i=1}^{n} (W_{ij}\, a_i) - T_j \right)$$

This calculation, then, is a function of the *difference*, or *error* between the input function and the internal threshold.

What functional form do we choose for f()? We could choose whatever we want -- square root, $\prod$ (product), log, e^x and so on. Mathematicians and computer scientists, however, have found that the *sigmoid* (S-shaped) function is particularly advantageous. A typical sigmoid function is:

$$f(x) = \frac{1}{(1 + e^{-x})}$$

which is shown in Figure 17.3.

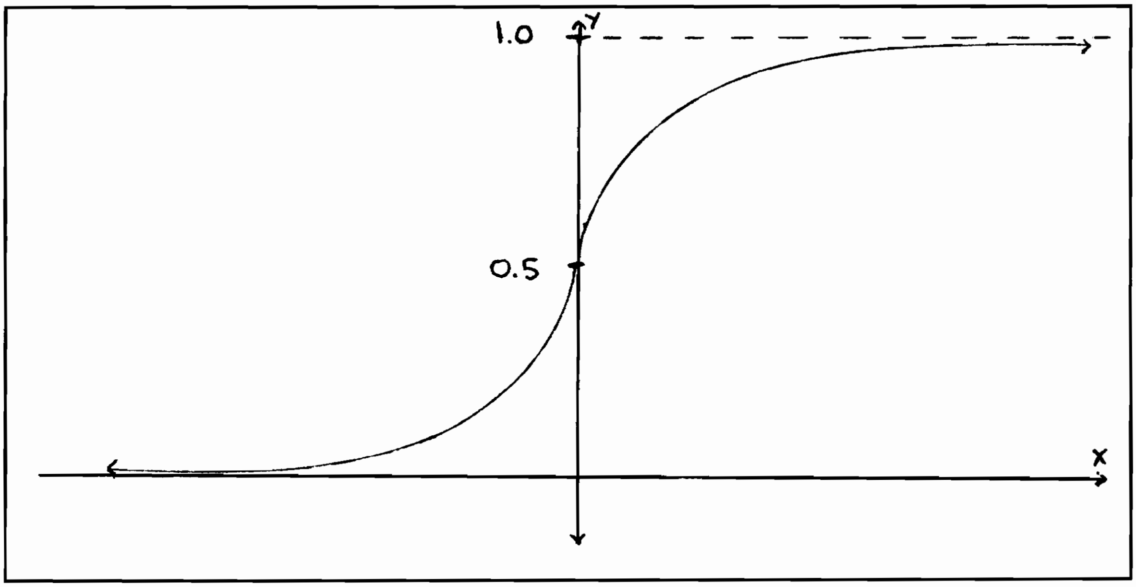


Figure 17.3. A sigmoid (S-shaped) function.

This function is monotonically increasing, with limiting values of 0 (at $x = -\infty$) and 1 (at $x = +\infty$). Because of these limiting values, sigmoid functions are called ***threshold functions***. At very low input values, the threshold-function output is zero. At very high input values, the output value is one. All sigmoid functions have upper and lower limiting values.

Another sigmoid function used is the hyperbolic tangent:

$$f(x) = \tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

with limiting values of -1 and +1. A third common sigmoid function is the ratio of squares:

$$f(x) = \{ \begin{array}{l} \frac{x^2}{1 + x^2} \text{ if } x > 0 \\ 0 \text{ if } x \leq 0 \end{array}$$

with limiting values of 0 and 1.

The sigmoid (S-shaped) functions, in general, give fairly well-behaved ANNs. The inhibitory and excitatory effects of the weight factors is straightforward when we use sigmoid functions (i.e., $w_{ij} < 0$ is inhibitory, and $w_{ij} > 0$ is excitatory). Sigmoid functions are continuous and monotonic, and are well-behaved even as x approaches $\pm \infty$. Because they are monotonic, they also provide for more efficient *training*. We frequently move down the slope of the curve in training, and the sigmoid functions have slopes that are well-behaved as a function of x (this topic will be discussed in gradient-descent learning, section 17.2F).

5. Summary of Processing-Element Anatomy

The fundamental building block of an ANN is the processing element (PE), seen in Figure 17.2. The PE has an n-dimensional input vector, a, an internal threshold value, T_j, and n weight factors (w_{1j}, ..., w_{ij}, ..., w_{nj}) which multiply all inputs. If the weighted input is large enough, the node becomes active and performs a calculation based on the difference between the internal threshold value and the weighted input value. Typically, a sigmoid (S-shaped) function is used for f(x), since it is a threshold function, is well-behaved, and provides for more rapid training.

B. Topology of an Artificial Neural Network

The *topology* of an ANN refers to how its PEs are interconnected. Figure 17.1 shows a very common topology. The network has three layers, one hidden, and each node's output feeds into all nodes in the subsequent layer. We form these topologies, or architectures, by organizing the PEs into layers, connecting them, and weighing the interconnections.

1. Inhibitory or Excitatory Connections

As mentioned, connections can either inhibit or excite the node. If the weight is positive, it will excite the node, increasing the activation of the PE. If the signal is negative, it will inhibit the node, decreasing the activation of the PE. If the signal is highly inhibitory, it may lower the input below the threshold level and shut the node down.

2. Connection Options

We have three connection options open to us, as shown in Figure 17.4. In *intra-layer* connections, the outputs from a node feed into other nodes in the same layer. In *inter-layer* connections, the outputs from a node in one layer feed into nodes in another layer. Finally, in *recurrent* connections, the output from a node feeds into itself.

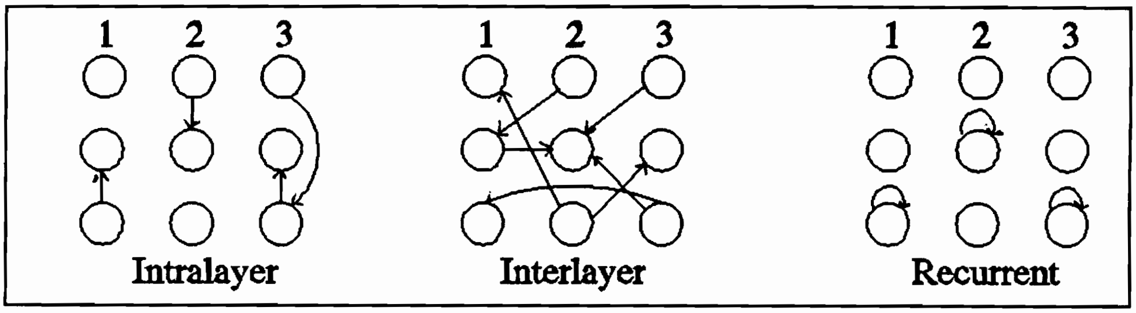


Figure 17.4. Connection options in an ANN.

Generally, when we first build an ANN, *we prespecify the topology*. That is, we specify the interconnections, but leave the numerical values of the weights up to the training phase. The inter-layer connection is particularly important to engineering applications. Within the inter-layer connecting scheme, we have two options: 1) *feedback connections*, and 2) *feedforward connections*, shown in Figure 17.5.

The type of problem we are trying to solve determines which topology we favor. For example, if we wish to develop an ANN that trains itself, we may use feedback connections. In contrast, in dynamic modeling of a chemical reactor, we are trying to map an output response based on an input signal; therefore, we favor the feedforward connection.

There is limited availability of rigorous mathematical theory for

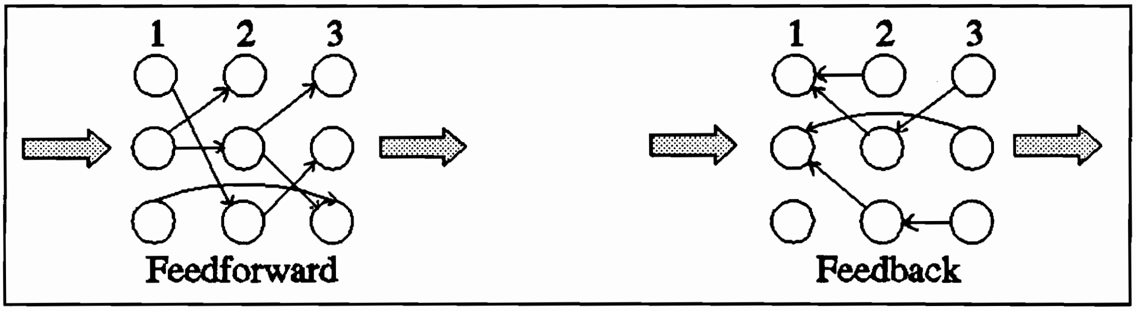


Figure 17.5. Feedback and feedforward connections.

training ANNs. Most approaches are based on feedforward networks, since they are the most applicable to science and engineering, the least complicated, and straightforward to implement.

C. Introduction to Learning and Training with Artificial Neural Networks

To *train* ANNs, we adjust the weight factors until the calculated output pattern (response) based on the given input matches the desired cause-and-effect relations. *Learning* is the actual process of adjusting weight factors based on trial-and-error. ANNs have many "buttons and knobs" to turn. There are many free variables to adjust, and as one person put it, "given enough parameters, you can curve-fit an elephant's back" (Mah, 1991). The challenge, however, is to systematically adjust the weight factors to give *proper* results in an *efficient* manner. With ANNs, this task is challenging.

1. Stability and Convergence

The training phase needs to produce an ANN that is both *stable* and *convergent*. A globally stable ANN maps any set of inputs to a fixed output. Stability guarantees a result, but it does not necessarily guarantee an accurate result.

A *convergent* ANN produces *accurate* input-output relations. Convergence, then, is related to the accuracy of the ANN. The degree of error between real-world results and those predicted by the ANN is a direct measurement of ANN's convergence.

2. Types of Learning

There are many different approaches to training ANNs. Simpson (1990), in his book *Artificial Neural Systems*, classifies a number of approaches. Most approaches fall into one of two groups:

- *Supervised learning* - an external teacher controls the learning and incorporates global information.
- *Unsupervised learning* - no external teacher is used and the ANN relies upon both internal control and local information. Frequently, the ANN develops its own models without additional input information.

We discuss some specific learning procedures below.

Error-Correction Learning- the most common type of learning used in ANNs today. It is a form of supervised learning, where we adjust weights in proportion to the output error vector, $\underline{\epsilon}$. This output error vector has n components, where n is the number of nodes on the output layer. We denote the n-th component of the vector as ϵ_n, and thus obtain an error component for each output from the ANN.

We begin error-correction learning by defining the output error from a single node on the output layer as:

$$\epsilon_n = d_n - b_n$$

where ϵ_n is the output error, d_n is the desired output, and b_n is the calculated output, for the *nth* node on the *output layer only.* We then calculate the total squared error on the output layer, *E*, as:

$$E = \sum_n \epsilon_n^2 = \sum_n (d_n - b_n)^2$$

Knowing *E*, we can calculate the change in the weight factor for the ith connection to the jth node, Δw_{ij}:

$$\Delta w_{ij} = \beta_j a_i E$$

β_j is a linear proportionality constant for node j (typically, $0 < \beta_j << 1$), and a_i is the i-th input to node j.

Reinforcement Learning- a type of supervised learning that is closely related to error-correction learning. In error-correction learning, we calculate a *vector* of error values, $\underline{\epsilon}$. Thus, n different error values represent the total performance of the ANN. In contrast, reinforcement learning has only one *scaler* error value to represent to total performance of the ANN. Since we have only one error value to reckon with, reinforcement learning is generally simpler and easier than classical error-correction learning. Reinforcement learning is "selectively supervised," and requires less information, possibly at infrequent intervals.

Stochastic Learning- utilizes statistics, probability, and/or random processes to adjust connection weights. We accept a random weight change if it reduces the error vector, $\underline{\epsilon}$. If the change increases $\underline{\epsilon}$, we generally reject the change. However, we may accept this change if, according to a specifically encoded probability analysis, it has a better-than-average probability of moving us to the global minimum in error. Accepting seemingly poorer weights allows stochastic processes to escape local minima and move to the global minimum.

Hardwired ANNs- have all connections and weights predetermined (hardwired). They have a speed advantage, and have been used with additional *a priori* information in speech recognition, language processing, vision, and robotics.

Hebbian Learning (named after Donald Hebb, 1949)- adjusts weights based on a correlation between the two nodes that the weight is associated with.

The simplest form of Hebbian learning uses direct proportionality. With node a_i in one layer connected to node b_j in another layer, we adjust the weight w_{ij} according to the equation:

$$w_{ij\,,\,new} = w_{ij\,,\,old} + \beta_j\, a_i\, b_j$$

where β_j is a learning rate constant for node j, and $0 < \beta_j < 1$.

D. Backpropagation Learning: Vanilla Backpropagation Algorithm

As mentioned, the most common form of learning utilized in ANNs today is error-correction learning. Previously, mathematicians and computer scientists looked down upon error-correction learning, primarily because the technique did not work on ANNs with hidden layers. It applied to two-layer ANNs since we could not define how much error came from nodes in the hidden layer. Through a technique known as *backpropagation*, however, we can now apply error-correction learning to ANNs with hidden layers.

Another challenge in error-correction learning is determining the value of the proportionality constant, β_j. Historically, values for β_j have been restricted such that $0 < \beta_j << 1$. Some researchers have identified values for β_j that give a stable solution. This section explains backpropagation and discusses how to adjust β_j and determine values for connection weights w_{ij}.

1. Requirements for Backpropagation Learning

Backpropagation requires an ANN known as a *perceptron*. What is a perceptron? No single definition exists, but for our purposes, we define a perceptron as an ANN with only *feedforward inter-layer connections*, and no intra-layer or recurrent connections. Each layer must feed sequentially into the next layer, with no feedback connections. Theory does exist for backpropagation on ANNs with multiple hidden layers, connections that skip over layers, recurrent connections, and even feedback connections. However, for simplicity, we shall investigate only the three-level, sequential perceptron shown in Figure 17.6.

The perceptron ANN in Figure 17.6 has three layers, A, B, and C. Feeding into layer A is the input vector $\underline{\mathbf{I}}_L$. Thus layer A has L nodes, that is, a_1, a_2, ..., a_i, ... a_L. Layer B, the hidden layer, has m nodes: b_1, b_2, ..., b_j, ... b_m. Note that in the drawing, $L = m = 3$; in practice, $L \neq m$ is acceptable. Layer C, the output layer, is next. There are n C-layer nodes, c_n, and again, $L \neq m \neq n$ is acceptable. The interconnection weight between the i-th node of layer A and the j-th node of layer B is denoted as v_{ij}, and that between the i-th node of layer B and the j-th node of layer C are w_{ij}. Each node has an internal threshold value. For layer A, the threshold is T_{Ai}, for layer B, T_{Bj}, and for layer C, T_{Ck}.

2. The Vanilla Backpropagation Technique

Backpropagation learning attempts to properly map given inputs with desired outputs by minimizing an error function. Typically, we use the

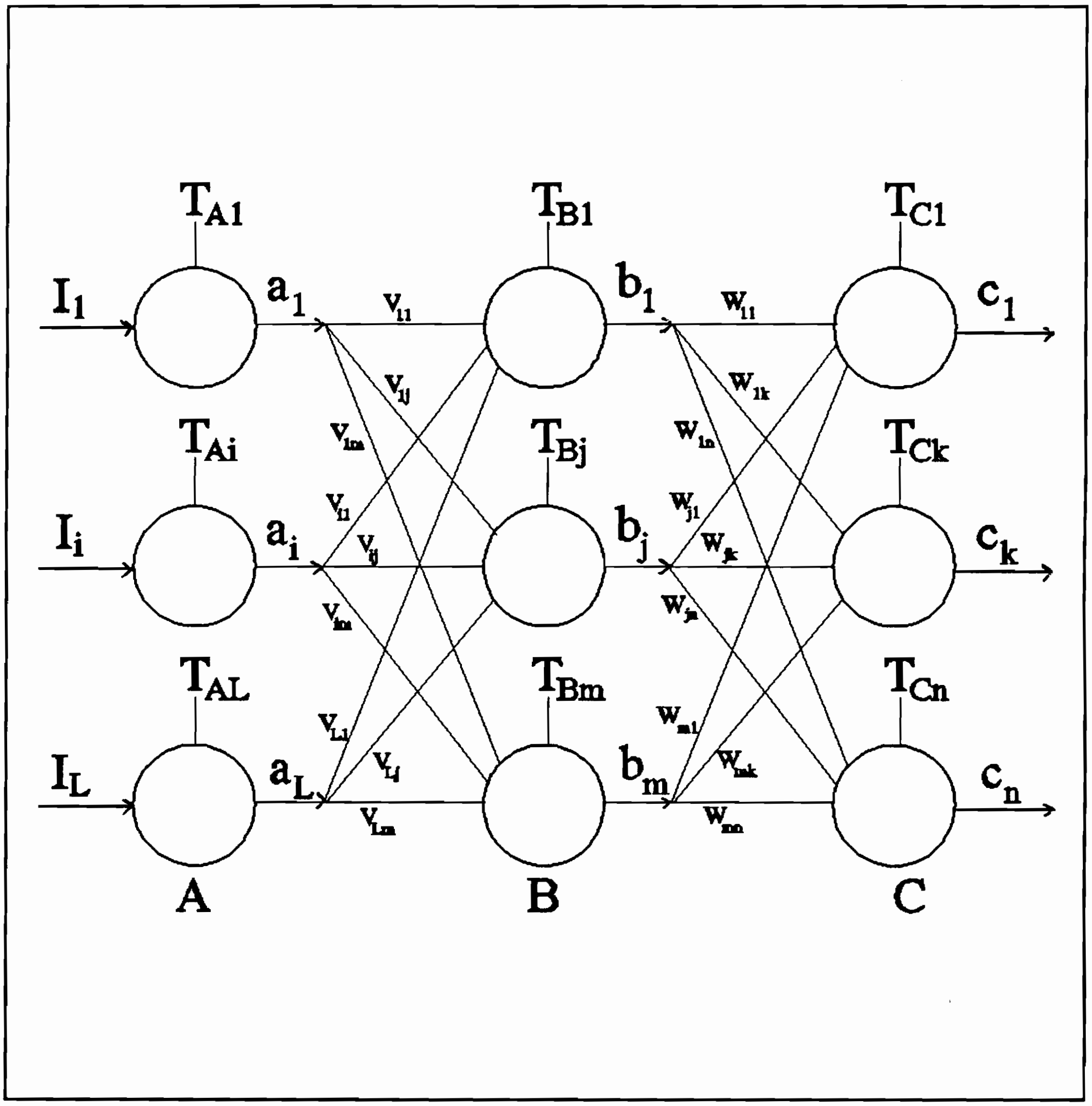


Figure 17.6. A three-level perceptron ANN.

sum-of-squares error. The component of the output error vector from the nth node on the output layer, ϵ_n, is defined as: $\epsilon_n = d_n - c_n$, where d_n is the desired output value and c_n is the calculated value. The total (squared) error function, E is:

$$E = \sum \epsilon_n^2 = \sum (d_n - c_n)^2$$

We adjust both interconnection weights, v_{ij}'s and w_{jk}'s, shown in Figure 17.6 to minimize *E*. Below is the step-by-step adjustment procedure known as the *vanilla backpropagation algorithm* (Simpson, 1990). To use this algorithm, we must have sets of data that map the input vector $\underline{I}_L$ with the output c_n.

Step 1: Randomly specify numerical values for all weight factors (v_{ij}'s and w_{jk}'s) within the interval [-1, +1]. Likewise, assign internal threshold values (T_{Ai}, T_{Bj}, T_{Ck}) for every node, also between +1 and -1. Note that i = 1, 2, ..., *L*, where *L* is the number of nodes in layer A; j = 1, 2, ..., m, where m is the number of nodes in layer B; and k = 1, 2, ..., n, where n is the number of nodes in layer C.

Step 2: Introduce the input vector I_i into the ANN. Calculate all outputs from the first layer, using the standard sigmoid function introduced previously:

$$x_i = I_i - T_{Ai}$$

$$a_i = f(x_i) = \frac{1}{(1 + e^{-x_i})}$$

Here, I_i is the input into the i-th node on the input layer, T_{Ai} is internal threshold for the node, and a_i is the output from the node.

Step 3: Given the output from layer A, calculate the output from layer B, using the equation:

$$b_j = f\left(\sum_{i=1}^{L} (v_{ij} a_i) - T_{Bj} \right)$$

where f() is the same sigmoid function.

Step 4: Given the output from layer B, calculate the output from layer C, using the equation:

$$c_k = f\left(\sum_{j=1}^{m} (w_{jk} b_j) - T_{Ck} \right)$$

where f() is the same sigmoid function.

Step 5: Now *backpropagate* through the network, starting at the output and moving backward toward the input. Calculate the k-th component of the output error, ϵ_k, for each node in layer C, according to the equation:

$$\epsilon_k = c_k \ (1 - c_k) \ (d_k - c_k)$$

where d_k is the desired result and c_k is the actual result.

Step 6: Continue backpropagation, moving to layer B. Calculate the j-th component of the error vector, e_j, of layer B relative to each ϵ_k, using the equation:

$$e_j = b_j \ (1 - b_j) \ \left(\sum_{k=1}^{n} (w_{jk} \ \epsilon_k) \right)$$

Step 7: Adjust weights, calculating the new w_{jk} ($w_{jk,\ new}$) as:

$$w_{jk\ ,\ new} = w_{jk\ ,\ old} + \beta_C \ b_j \ \epsilon_k$$

for j = 1 to m and k = 1 to n. The term β_C is a positive constant controlling the learning rate in layer C.

Step 8: Adjust the thresholds T_{Ck} (k = 1 to n) in layer C, according to the equation:

$$T_{Ck\,,\,new} = T_{Ck\,,\,old} + \beta_C \epsilon_k$$

Step 9: Adjust weights v_{ij}, according to the equation:

$$v_{ij\,,\,new} = v_{ij\,,\,old} + \beta_B\, a_i\, e_j$$

for i = 1 to L and j = 1 to m. The term β_B is a positive constant controlling the learning rate in layer B.

Step 10: Adjust the thresholds T_{Bj} (j = 1 to m) in layer B, according to the equation:

$$T_{Bj\,,\,new} = T_{Bj\,,\,old} + \beta_B e_j$$

Step 11: Repeat steps 2-10 until the squared error, E, or the output error vector, $\underline{\epsilon}$, is zero or sufficiently low.

This algorithm requires a few comments. Note that we do *not* adjust the internal threshold values for layer A. Therefore, depending on the problem being solved, we may wish to set all T_{Ai}'s equal to zero. Note also that in this network, the sigmoid function restricts the output to values between zero and one. This restriction can cause some problems when the ANN is used for empirical modeling. If we desire values outside the [0,1] interval, some type of "normalized" output value is required (see practice problems in section 17.3E).

The vanilla backpropagation algorithm is a *gradient-descent technique*. We use Newton's method (moving down a gradient on a surface) to minimize the error. The advantage of this method is that weight changes are estimated systematically rather than arbitrarily. The sigmoid function on the output layer is:

$$f(x_k) = \frac{1}{(1 + e^{-x_k})}$$

Therefore, the partial derivative of the sigmoid function is:

$$\frac{\partial f}{\partial x_k} = \frac{e^{-x_k}}{(1 + e^{-x_k})^2}$$

Thus, it is evident that

$$\frac{\partial f}{\partial x_k} = \frac{1}{1 + e^{-x_k}} \left(1 - \frac{1}{(1 + e^{-x_k})^2}\right)$$

and therefore,

$$\frac{\partial f}{\partial x_k} = c_k \ (1 - c_k)$$

Thus, we see that the term $c_k(1-c_k)$ in Step 5 of the algorithm is actually the gradient for Newton's Method (i.e., the partial derivative with respect to x_k).

E. An Example of Backpropagation Learning

1. A Qualitative Interpretation Problem

Consider again the perceptron network shown in Figure 17.6. We will use this network for *qualitative interpretation* of process data from a chemical reactor. The input and output vectors are shown in Table 17.1.

Table 17.1. Input and output for fault diagnosis ANN.

INPUT VECTOR		**OUTPUT VECTOR**	
I_1	Reactor Inlet Temperature, °F	C_1	Low Conversion
I_2	Reactor Inlet Pressure, psia	C_2	Low Catalyst Selectivity
I_3	Feed Flow Rate, lb/min	C_3	Catalyst Sintering

The *desired* output from the ANN is Boolean, i.e., zero indicates that no problem exists, and one indicates that a problem does exist. The *actual* output from the ANN is a numeric value between 0 and 1, and can almost be viewed as a "probability" that the problem will result from this type of input (0 = the problem definitely does not exist; 1 = the problem definitely does exist).

2. Training

Given the above inputs and outputs, we can train the network to recognize the following specific conditions:

Input:	I_1 = 300 °F	*Output:*	C_1 = 1 (low conversion)
	I_2 = 100 psia		C_2 = 0 (no problem)
	I_3 = 200 lb/min		C_3 = 0 (no problem)

We use the perceptron ANN shown in Figure 17.6, and implement the backpropagation algorithm:

Step 1: To use a computer, let layer A = layer 1, layer B = layer 2, and layer C = layer 3. We randomly assign values for v_{ij} and w_{jk}, within the interval [-1, +1]. These values are:

$$v_{ij} = \begin{vmatrix} -1.0 & -0.5 & 0.5 \\ 1.0 & 0.0 & -0.5 \\ 0.5 & -0.5 & 0.5 \end{vmatrix} \qquad w_{jk} = \begin{vmatrix} -1.0 & -0.5 & 0.5 \\ 1.0 & 0.0 & 0.5 \\ 0.5 & -0.5 & 0.5 \end{vmatrix}$$

For the internal threshold values (T_{1i}, T_{2j}, T_{3k}), we set $T_{1i} = 0$, and randomly assign values between -1 and 1 to T_{2j} and T_{3k}. All internal thresholds are represented by the matrix:

$$T = \begin{vmatrix} 0.0 & 0.0 & 0.0 \\ 0.5 & 0.0 & -0.5 \\ 0.0 & 0.5 & -0.5 \end{vmatrix}$$

Step 2: We introduce the input vector into the ANN. We calculate outputs from layer 1:

$$x_1 = I_1 - T_{11} = 300 - 0 = 300$$
$$x_2 = I_2 - T_{12} = 100 - 0 = 100$$
$$x_3 = I_3 - T_{13} = 200 - 0 = 200$$

Substituting these values into the sigmoid function, we get: $a_1 = a_2 = a_3 = 1.0$.

Step 3: We calculate the output from each node in layer 2:

$$b_1 = f(v_{11}a_1 + v_{21}a_2 + v_{31}a_3 - T_{21}) = f(0) = 0.50000$$

$$b_2 = f(v_{12}a_1 + v_{22}a_2 + v_{32}a_3 - T_{22}) = f(-1) = 0.26894$$

$$b_3 = f(v_{13}a_1 + v_{23}a_2 + v_{33}a_3 - T_{23}) = f(1) = 0.73106$$

Step 4: We calculate the output from each node in layer 3:

$$c_1 = f(w_{11}b_1 + w_{21}b_2 + w_{31}b_3 - T_{31}) = f(0.13447) = 0.53356$$

$$c_2 = f(w_{12}b_1 + w_{22}b_2 + w_{32}b_3 - T_{32}) = f(-1.11522) = 0.24684$$

$$c_3 = f(w_{13}b_1 + w_{23}b_2 + w_{33}b_3 - T_{33}) = f(1.25) = 0.77730$$

Step 5: We backpropagate, first calculating the error for each node in layer 3:

$$\epsilon_1 = c_1 (1 - c_1) (d_1 - c_1) = 0.11608$$

$$\epsilon_2 = c_2 (1 - c_2) (d_2 - c_2) = -4.5890 \times 10^{-2}$$

$$\epsilon_3 = c_3 (1 - c_3) (d_3 - c_3) = -0.134554$$

Step 6: We calculate the errors associated with layer 2, the hidden layer:

$$e_1 = b_1(1 - b_1)(w_{11}\epsilon_1 + w_{12}\epsilon_2 + w_{13}\epsilon_3) = -4.01036 \times 10^{-2}$$

$$e_2 = b_2(1 - b_2)(w_{21}\epsilon_1 + w_{22}\epsilon_2 + w_{23}\epsilon_3) = 9.59574 \times 10^{-3}$$

$$e_3 = b_3(1 - b_3)(w_{31}\epsilon_1 + w_{32}\epsilon_2 + w_{33}\epsilon_3) = 2.69542 \times 10^{-3}$$

Step 7: We adjust weight factors for interconnections between layers 2 and 3. For simplicity, we assume that $\beta = 0.7$ for *all* values of β. Thus, the learning rate, β_j, equals 0.7. Similarly the rates to calculate internal threshold values, i.e., β_2 and β_3, also equal 0.7. We adjust the w_{jk} weight factors as follows:

$$w_{11,new} = w_{11,old} + \beta b_1 \epsilon_1 = -0.95937$$

$$w_{12,new} = w_{12,old} + \beta b_1 \epsilon_2 = -0.51606$$

$$w_{13,new} = w_{13,old} + \beta b_1 \epsilon_3 = 0.45291$$

We continue these adjustments for the rest of the w_{jk}'s. A comparison of the old and new weight factors shows:

$$w_{jk,old} = \begin{vmatrix} -1.0 & -0.5 & 0.5 \\ 1.0 & 0.0 & 0.5 \\ 0.5 & -0.5 & 0.5 \end{vmatrix} \qquad w_{jk,new} = \begin{vmatrix} -0.96 & -0.52 & 0.45 \\ 1.02 & 0.0086 & 0.47 \\ 0.56 & -0.52 & 0.43 \end{vmatrix}$$

Step 8: We adjust the internal thresholds for layer 3:

$$T_{31,new} = T_{31,old} + \beta \epsilon_1 = 8.1258 \times 10^{-2}$$

$$T_{32,new} = T_{32,old} + \beta \epsilon_2 = 0.46788$$

$$T_{33,new} = T_{33,old} + \beta \epsilon_3 = -0.59419$$

Thus, new and old threshold values are:

Old: $T_{31} = 0.0$	$T_{32} = 0.5$	$T_{33} = -0.5$
New: $T_{31} = 0.081$	$T_{32} = 0.468$	$T_{33} = -0.594$

<u>Step 9:</u> We adjust the weight factors, v_{ij}, for interconnections between layers 1 and 2. Again, $\beta = 0.7$.

$$v_{11,new} = v_{11,old} + \beta a_1 e_1 = -1.02807$$

$$v_{12,new} = v_{12,old} + \beta a_1 e_2 = -0.49328$$

$$v_{13,new} = v_{13,old} + \beta a_1 e_3 = 0.50188$$

We continue these adjustments for the rest of the v_{ij}'s, and the old and new weight factors are:

$$v_{ij,old} = \begin{vmatrix} -1.0 & -0.5 & 0.5 \\ 1.0 & 0.0 & -0.5 \\ 0.5 & -0.5 & 0.5 \end{vmatrix} \qquad v_{ij,new} = \begin{vmatrix} -1.03 & -0.493 & 0.502 \\ 0.972 & 0.0067 & -0.498 \\ 0.472 & -0.493 & 0.502 \end{vmatrix}$$

<u>Step 10:</u> We adjust the internal thresholds for layer 3:

$$T_{21,new} = T_{21,old} + \beta e_1 = 0.47193$$

$$T_{22,new} = T_{22,old} + \beta e_2 = 6.7170\text{x}10^{-3}$$

$$T_{23,new} = T_{23,old} + \beta e_3 = -0.49811$$

Thus, new and old internal threshold values are:

Old: $T_{21} = 0.5$	$T_{22} = 0.0$	$T_{23} = -0.5$
New: $T_{21} = 0.472$	$T_{22} = 0.0067$	$T_{23} = -0.498$

We have completed one time step. Now, we go back to step 2 and repeat the procedure until we converge on the correct values. For this problem, we need 3860 time steps to get results at less than 1% error on variable d_1 (i.e., error $e_1 < 0.01$):

Number of time steps: 3860

Desired values:	$d_1 = 1$	$d_2 = 0$	$d_3 = 0$
Actual values:	$c_1 = 0.9900$	$c_2 = 0.0156$	$c_3 = 0.0098$
Percent error:	$e_1 = -1.00\%$	$e_2 = 1.56\%$	$e_3 = 0.98\%$

One of the biggest challenges for ANNs is the training: it is often long and tedious. For one data point, we need 3860 iterations to train the network. When we need hundreds of data points to map out the entire reactor operating range, the ANN borders on impracticality.

Programming this backpropagation algorithm is not too difficult. Table 17.2 gives a simple, 112-line program written in BASIC.

Table 17.2. Code for the Backpropagation Algorithm. Written in BASIC.

```
1 REM *** VANILLA BACKPROPAGATION ALGORITHM ***
2 REM ***      WRITTEN IN BASIC             ***
3 REM *****************************************
4 REM  * VARIABLES AND THEIR MEANINGS         *
5 REM  * A(I) = OUTPUT FROM NODES IN LAYER 1  *
6 REM  * B(I) = OUTPUT FROM NODES IN LAYER 2  *
7 REM  * BETA = LEARNING RATE                 *
8 REM  * C(I) = OUTPUT FROM NODES IN LAYER 3  *
9 REM  * D(I) = DESIRED OUTPUT FROM LAYER 3   *
10 REM * E(I) = ERROR CALCULATED FROM LAYER 3 *
11 REM * E(I) = ERROR CALCULATED FROM LAYER 3 *
12 REM * EB(I) = ERROR CALCULATED FROM LAYER 3*
13 REM * PNPUT(I) = INPUT VECTOR INTO LAYER 1 *
14 REM * T(I,J) = INTERNAL THRESHOLD VALUES,  *
15 REM *     I = LAYER NUMBER                 *
16 REM *     J = NODE NUMBER WITHIN LAYER     *
17 REM * V(I,J) = LAYER 1-2 CONNECTION WEIGHTS*
18 REM * W(J,K) = LAYER 2-3 CONNECTION WEIGHTS*
19 REM * X(I) = POST-THRESHOLD INPUT, LAYER 1 *
20 REM ****************************************
21 DIM A(3),B(3),C(3),D(3),E(3),EB(3),PNPUT(3)
22 DIM T(3,3),V(3,3),W(3,3),X(3)
23 REM *** STEP 1: INITIALIZE THE VARIABLES ***
24 V(1,1) = -1 : V(1,2) = -.5 : V(1,3) = .5
25 V(2,1) =  1 : V(2,2) = 0.0 : V(2,3) = -.5
26 V(3,1) = .5 : V(3,2) = -.5 : V(3,3) = .5
27 W(1,1) = -1 : W(1,2) = -.5 : W(1,3) = .5
28 W(2,1) =  1 : W(2,2) =  0  : W(2,3) = .5
29 W(3,1) = .5 : W(3,2) = -.5 : W(3,3) = .5
30 T(1,1) =  0 : T(1,2) =  0  : T(1,3) =  0
31 T(2,1) = .5 : T(2,2) =  0  : T(2,3) = -.5
32 T(3,1) =  0 : T(3,2) = .5  : T(3,3) = -.5
33 PNPUT(1) = 300
34 PNPUT(2) = 100
35 PNPUT(3) = 200
36 D(1) = 1 : D(2) = 0 : D(3) = 0
37 BETA = .7
38 REM *** STEP 2: INTRODUCE INPUT VECTOR ***
39 REM *** ALSO CALCULATE LAYER ONE OUTPUTS ***
40       FOR I = 1 TO 3
41       X(I) = PNPUT(I) - T(1,I)
42       A(I) = 1/(1+EXP(-X(I)))
43       NEXT I
44 REM *** STEP 3: FIND LAYER TWO OUTPUTS ***
45       FOR J = 1 TO 3
46       SUM = 0
47              FOR I = 1 TO 3
48              SUM = SUM + V(I,J)*A(I)
49              NEXT I
50       B(J) = 1/(1+EXP(-(SUM-T(2,J))))
51       NEXT J
52 REM *** STEP 4: FIND LAYER THREE OUTPUTS ***
53       FOR K = 1 TO 3
54       SUM = 0
55              FOR J = 1 TO 3
56              SUM = SUM + W(J,K)*B(J)
57              NEXT J
58       C(K) = 1/(1+EXP(-(SUM-T(3,K))))
59       NEXT K
60 REM *** STEP 5: FIND OUTPUT DIFFERENCE ***
61 DIF1 = D(1)-C(1)
62 DIF2 = D(2)-C(2)
63 DIF3 = D(3)-C(3)
64 PRINT "DESIRED VALUES: D1  D2  D3"
65 PRINT D(1);D(2); D(3)
66 PRINT "ACTUAL VALUES: C1  C2  C3"
67 REM *** STOP EXECUTION IF ERROR IS LOW ***
68 IF ABS(DIF3) < .0001 THEN END
69 REM *** FIND OUTPUT ERROR ***
70       FOR K = 1 TO 3
71       E(K) = C(K)*(1-C(K))*(D(K)-C(K))
72       NEXT K
73 PRINT "BEGINNING BACKPROPAGATION ..."
74 REM *** FIND LAYER TWO ERROR ***
75       FOR J = 1 TO 3
76       SUM = 0
77              FOR K = 1 TO 3
78              SUM = SUM + W(J,K)*E(K)
79              NEXT K
80       EB(J) = B(J)*(1-B(J))*SUM
81       NEXT J
82 REM *** STEP 7: ADJUST W(J,K) WEIGHTS ***
83 REM *** STEP 8: ADJUST T(3,K) THRESHOLDS ***
84       FOR K = 1 TO 3
85              FOR J = 1 TO 3
86              W(J,K) = W(J,K) + BETA*B(J)*E(K)
87              NEXT J
88       T(3,K) = T(3,K) + BETA*E(K)
89       NEXT K
90 REM *** STEP 9: ADJUST V(I,J) WEIGHTS ***
91 REM *** STEP 10: ADJUST T(2,J) THRESHOLDS ***
92       FOR J = 1 TO 3
93              FOR I = 1 TO 3
94              V(I,J) = V(I,J) + BETA*A(I)*EB(J)
95              NEXT I
96              T(2,J) = T(2,J) + BETA*EB(J)
97              NEXT J
98 PRINT "IF YOU WISH TO EXECUTE ANOTHER
99 PRINT "TIME-STEP, ENTER: CONT"
100 STOP
101 GOTO 40
102 REM ****************************************
103 REM *                                      *
104 REM * THE USER IS ENCOURAGED TO PLACE BOTH *
105 REM * PRINT AND STOP STATEMENTS ANYWHERE   *
106 REM * IN THE PROGRAM TO VIEW THE PROGRESS  *
107 REM * OF THE PROGRAM. WHEN THE PROGRAM     *
108 REM * CEASES EXECUTION AT A STOP STATEMENT,*
109 REM * SIMPLY ENTER CONT (CONTINUE) AT THE  *
110 REM * INTERPRETER TO RESUME EXECUTION      *
111 REM *                                      *
112 REM ****************************************
```

3. Recall

Note that the program shown in Table 17.2 is incomplete if we wish to implement a "commercial" ANN. The program is for *backpropagation training only*. Implementing a commercial ANN requires a *recall* algorithm for use after the ANN is completely trained. In the reactor-analysis problem shown in Table 17.1, we would use the ANN recall algorithm to predict undesirable consequences based on operating conditions.

F. Sigmoid Threshold Functions in BackPropagation

The ANN described in Figure 17.6 has a potential problem: the presence of the internal threshold values, T_{Ai}, T_{Bj}, T_{Ck}. As these threshold values are adjusted during backpropagation, node-activation levels (as determined by weights from the previous time step) can suddenly be excessively low or high, depending on the new threshold value. These sudden changes in node activation induce some undesirable consequences. First, these changes introduce discontinuities in the network. The discontinuities may, in turn, lead to poor network stability. During training, the network may be prone to more oscillations. In addition, mathematicians have proven that the network is not able to simulate all types of mathematical functions, such as the exclusive-or gate used in logic and Boolean algebra.

Because of these difficulties, many ANNs do not use internal threshold values (i.e the T_{Ai}'s, T_{Bj}'s, and T_{Ck}'s are all set to zero). Instead, the sigmoid function itself acts as a "gradual threshold," since it has a limiting value of zero at low activation and unity at high activation. Figure 17.7 shows a network with these properties. The sigmoid functions deactivate the node, and hence, they are called *sigmoid threshold functions*.

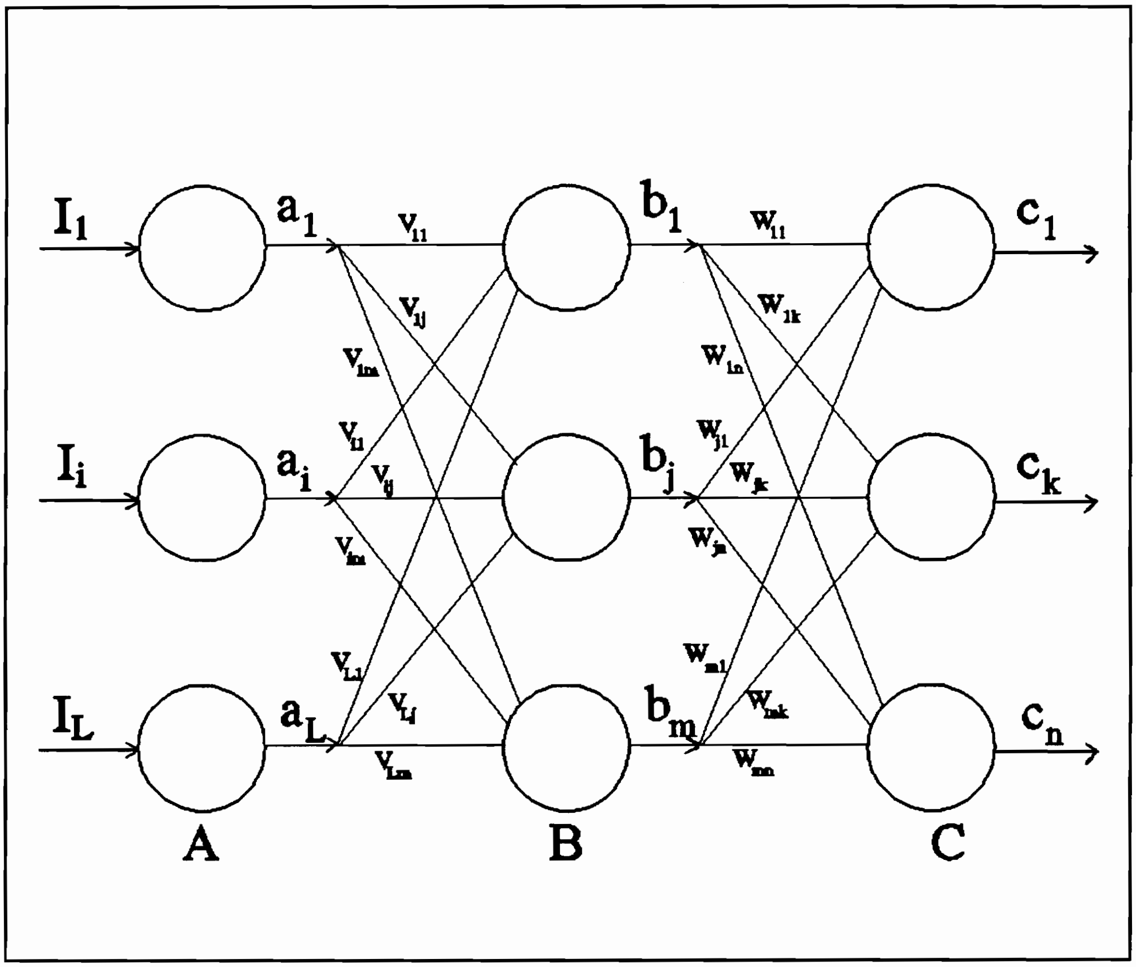


Figure 17.7. An ANN with sigmoid threshold functions. All T_{Ai}'s, T_{Bj}'s, and T_{Ck}'s are set to zero.

Researchers developed the sigmoid function in part to work in the absence of T_{Ai}'s, T_{Bj}'s, and T_{ck}'s. Normal internal thresholds created discontinuity problems. Researchers therefore preferred to have a gradual, S-shaped threshold. To solve this problem, they set the T_{Ai}'s, T_{Bj}'s, and T_{ck}'s to zero and rely on the sigmoid function itself to deactivate the node. Hence, we have a sigmoid threshold function, as shown in Figure 17.7.

G. Generalized Delta-Rule (GDR) Algorithm

Today, however, with the growing number of more sophisticated training algorithms, many of those algorithms use both the internal threshold and sigmoid threshold functions. An example is GDR algorithm, a gradient-descent technique described in this section.

1. Generalized Delta-Rule (GDR) Algorithm

One difficulty with backpropagation algorithms is the extensive time required to train the network. Depending on the size of the ANN, training can take hours or even days on a mainframe computer. Researchers have investigated many different training procedures in an attempt to speed up the learning process. One technique that has been frequently applied, as we investigated with the vanilla backpropagation algorithm, is gradient-descent learning.

The *Generalized Delta-Rule* (GDR), is an iterative gradient-descent method that minimizes the squared error. It is related to the vanilla backpropagation algorithm, but has some differences. First, the GDR uses a technique known as *momentum* to speed up the training. Momentum is an extra weight added onto the weight factors when they are adjusted. By accelerating the change in the weight factors, we improve the training rate.

Another difference between the GDR and the vanilla backpropagation algorithm is the presence of a *bias function* instead of internal threshold values. The internal thresholds (T_{1i}, T_{2j}, and T_{3k}) become a bias function when we *add* (rather than subtract as in the vanilla backpropagation) a fixed number to the nodal summation. In addition, when serving as a bias function, the values of T_{1i}, T_{2j}, and T_{3k} are *not* changed or updated as training progresses. In the GDR, we set $T_{1i} = 0$, and $T_{2j} = T_{3k} = 1$, and these values remain unchanged throughout the entire life of the ANN.

By using momentum coupled with a bias function, the GDR algorithm is more efficient than the vanilla backpropagation algorithm. Let us investigate the GDR algorithm. We use the GDR for the three-layer feedforward perceptron shown in Figure 17.6.

Step 1: Randomly assign values between 0 and 1 to weights v_{ij} and w_{jk}. For GDR, the internal threshold values *must* be assigned as follows: all input-layer thresholds must equal zero, i.e. T(1,i) = 0; all hidden- and output-layer thresholds must equal one, i.e., T(2,j) = T(3,k) = 1.

Step 2: Introduce the input vector I_i into the ANN, and calculate the output from the first layer according to the equations:

$$x_i = I_i - T_{1i} = I_i - 0 = I_i$$

$$a_i = \frac{1}{(1 + e^{-x_i})}$$

<u>Step 3:</u> Knowing the output from the first layer, calculate outputs from the second layer, using the equation:

$$b_j = f\left(\sum_{i=1}^{L} (v_{ij}\, a_i) + T_{2j} \right)$$

where f() is the sigmoid function. Note that $T_{2j} = 1$ in this algorithm, and we are *adding* it to the summation (rather than *subtracting* it as we did in the vanilla backpropagation algorithm). When used in this mode, T_{2j} acts as a bias function.

<u>Step 4:</u> Knowing the output from the second layer, calculate the result from the output layer, according to the equation:

$$c_k = f\left(\sum_{j=1}^{m} (w_{jk}\, b_j) + T_{3k} \right)$$

where f() is the sigmoid function. Note again, $T_{3k} = 1.0$ and we are *adding* it to the sum. Again, T_{3k} acts as a bias function.

<u>Step 5:</u> Continue steps 1-4 for M number of training patterns presented to the input layer. Calculate the total-squared error, *E*, according to the following equation:

$$E = \sum_{m=1}^{M} \sum_{k=1}^{n} (d_k^m - c_k^m)^2$$

where M is the number of training patterns presented to the input layer, n is the number of PEs on the output layer, d_k^m is the desired output value from the k-th PE in the m-th training pattern, and c_k^m is the actual output value from the k-th PE in the m-th training pattern.

Step 6: Knowing the m-th pattern, calculate δ_{3k}^m, the gradient-descent term for the k-th PE in the *output layer* (layer 3) for training pattern m. Use the following equation:

$$\delta_{3k}^m = (d_k^m - c_k^m) \frac{\partial f}{\partial x_k}$$

where *f* is the sigmoid function:

$$f(x_k) = \frac{1}{(1 + e^{-x_k})}$$

The partial derivative of the sigmoid function is:

$$\frac{\partial f}{\partial x_k} = \frac{e^{-x_k}}{(1 + e^{-x_k})^2}$$

Note that x_k is the sum of the weighted inputs to the k-th node on the output layer; i.e., for the m-th training session:

$$x_k^m = \sum_j w_{jk}^m b_j^m + T_{3k}^m$$

where, for training pattern m, b_j^m is the output value of the j-th element in the hidden layer and T_{3k}^m is a threshold value on the output layer.

Step 7: Again knowing the m-th pattern, calculate $\delta_{2j}{}^{m}$, the gradient-descent term for the j-th PE on the *hidden layer* (layer 2). Use the equation:

$$\delta_{2j}^{m} = \left(\sum_{k} \delta_{3k}^{m} w_{jk}^{m}\right) \frac{\partial f}{\partial x_j}$$

where the subscript k denotes a node in the output layer. Recall that x_j is defined by:

$$x_j^{m} = \sum_{i} v_{ij}^{m} a_i^{m} + T_{2j}^{m}$$

and the partial derivative of the sigmoid function, again, is:

$$\frac{\partial f}{\partial x_j} = \frac{e^{-x_j}}{(1 + e^{-x_j})^2}$$

Step 8: Knowing $\delta_{2j}{}^{m}$ for the hidden layer and $\delta_{3k}{}^{m}$ for the output layer, calculate the weight changes using the equations:

$$\Delta v_{ij}^{m} = \eta\ \delta_{2j}^{m}\ a_i^{m} + \alpha\ \Delta v_{ij}^{m-1}$$

$$\Delta w_{jk}^{m} = \eta\ \delta_{3k}^{m}\ b_j^{m} + \alpha\ \Delta w_{jk}^{m-1}$$

where η is the learning rate, and α is a coefficient of *momentum*. As mentioned, momentum is simply an added weight used to speed up the training rate. The coefficient for momentum, α, is usually restricted such that $0 < \alpha < 1$. Thus, the momentum terms, $\alpha\ \Delta w_{jk}{}^{m-1}$ and $\alpha\ \Delta v_{ij}{}^{m-1}$, are fractional values of the weight change from the previous iteration.

Step 9: Knowing the weight changes, update the weights according to the equations:

$$w_{jk}^{m} = w_{jk}^{m-1} + \Delta w_{jk}^{m}$$

$$v_{ij}^{m} = v_{ij}^{m-1} + \Delta v_{ij}^{m}$$

where v_{ij}^{m} is the connection weight between the i-th element in the input layer and j-th element in the hidden layer, w_{jk}^{m} is the connection weight between the j-th element in the hidden layer and k-th element in the output layer, both for the m-th training iteration. Repeat steps 2-9 for all training patterns until the squared error is zero or sufficiently low.

From these steps, we see that the *Generalized Delta-Rule* computes the output error, then generates new weight values based on the gradient of the sigmoid function and that error. It updates the output node first, and propagates back to the next layers, until it reaches the input layer. It also uses momentum and a bias function, which are two distinguishing features between the GDR and the vanilla backpropagation algorithms.

H. An Example of the Generalized Delta-Rule

Let us compare the GDR and the vanilla backpropagation algorithms. We will use the ANN for qualitative interpretation of a chemical reactor, shown in Table 17.1. The GDR can handle multiple input patterns (vectors), but for this comparison, we will use it for one input pattern only. Our desired input-output pattern is:

Input: $I_1 = 300\ ^\circ F$ *Output:* $c_1 = 1$ (low conversion)

$I_2 = 100$ psia $c_2 = 0$ (no problem)

$I_3 = 200$ lb/min $c_3 = 0$ (no problem)

We will now march through the step-by-step procedure and see how well the GDR algorithm works.

Step 1: We assign the same values for v_{ij} and w_{jk} as we did in the vanilla backpropagation algorithm.

$$v_{ij} = \begin{vmatrix} -1.0 & -0.5 & 0.5 \\ 1.0 & 0.0 & -0.5 \\ 0.5 & -0.5 & 0.5 \end{vmatrix} \qquad w_{jk} = \begin{vmatrix} -1.0 & -0.5 & 0.5 \\ 1.0 & 0.0 & 0.5 \\ 0.5 & -0.5 & 0.5 \end{vmatrix}$$

For the internal threshold values (T_{1i}, T_{2j}, T_{3k}), we set the threshold of layer 1 equal to zero, and set those of layers 2 and 3 equal to one:

$$T = \begin{vmatrix} 0.0 & 0.0 & 0.0 \\ 1.0 & 1.0 & 1.0 \\ 1.0 & 1.0 & 1.0 \end{vmatrix}$$

This T-matrix serves as a bias function.

Step 2: We introduce the input vector into the ANN and calculate the outputs from layer 1:

$$x_1 = I_1 - T_{11} = 300 - 0 = 300$$
$$x_2 = I_2 - T_{12} = 100 - 0 = 100$$
$$x_3 = I_3 - T_{13} = 200 - 0 = 200$$

When we substitute these values into the sigmoid function, we get: $a_1 = a_2 = a_3 = 1.0$.

Step 3: We calculate the output from each node in layer two:

$$b_1 = f(v_{11}a_1 + v_{21}a_2 + v_{31}a_3 + T_{21}) = f(1.5) = 0.81757$$

$$b_2 = f(v_{12}a_1 + v_{22}a_2 + v_{32}a_3 + T_{22}) = f(0) = 0.5$$

$$b_3 = f(v_{13}a_1 + v_{23}a_2 + v_{33}a_3 + T_{23}) = f(1.5) = 0.81757$$

Step 4: We calculate the output from each node in layer three:

$$c_1 = f(w_{11}b_1 + w_{21}b_2 + w_{31}b_3 + T_{31}) = f(1.09121) = 0.74861$$

$$c_2 = f(w_{12}b_1 + w_{22}b_2 + w_{32}b_3 + T_{32}) = f(0.18243) = 0.54548$$

$$c_3 = f(w_{13}b_1 + w_{23}b_2 + w_{33}b_3 + T_{33}) = f(2.06757) = 0.88771$$

Step 5: We are training the network with just one input pattern, and our squared error is:

$$E = \sum_{k=1}^{3} (d_k - c_k)^2 = (d_1 - c_1)^2 + (d_2 - c_2)^2 + (d_3 - c_3)^2 = 1.14878$$

Step 6: We calculate δ_{3k}, the gradient-descent term for layer 3. To obtain δ_{3k}, we must first calculate the gradients:

$$\frac{\partial f}{\partial x_1} = \frac{e^{-x_1}}{(1 + e^{-x_1})^2} = \frac{e^{-1.091213}}{(1 + e^{-1.091213})^2} = 0.1881931$$

$$\frac{\partial f}{\partial x_2} = \frac{e^{-x_2}}{(1 + e^{-x_2})^2} = \frac{e^{-0.182426}}{(1 + e^{-0.182426})^2} = 0.2479316$$

$$\frac{\partial f}{\partial x_3} = \frac{e^{-x_3}}{(1 + e^{-x_3})^2} = \frac{e^{-2.067575}}{(1 + e^{-2.067575})^2} = 0.0996799$$

Knowing the gradients, we calculate the δ_{3k}'s:

$$\delta_{31} = (d_1 - c_1) \frac{\partial f}{\partial x_1} = 4.730985 \text{ x } 10^{-2}$$

$$\delta_{32} = (d_2 - c_2) \frac{\partial f}{\partial x_2} = -0.1352417$$

$$\delta_{33} = (d_3 - c_3) \frac{\partial f}{\partial x_3} = -8.848695 \text{ x } 10^{-2}$$

<u>Step 7:</u> We calculate δ_{2j}, the gradient-descent term for layer 2. Again, we must first calculate the gradients:

$$\frac{\partial f}{\partial x_1} = \frac{e^{-x_1}}{(1 + e^{-x_1})^2} = \frac{e^{-1.5}}{(1 + e^{-1.5})^2} = 0.1491465$$

$$\frac{\partial f}{\partial x_2} = \frac{e^{-x_2}}{(1 + e^{-x_2})^2} = \frac{e^{0}}{(1 + e^{0})^2} = 0.25$$

$$\frac{\partial f}{\partial x_3} = \frac{e^{-x_3}}{(1 + e^{-x_3})^2} = \frac{e^{-1.5}}{(1 + e^{-1.5})^2} = 0.1491465$$

Then the δ_{2j}'s are:

$$\delta_{21} = (\delta_{31}w_{11} + \delta_{32}w_{12} + \delta_{33}w_{13}) \frac{\partial f}{\partial x_1} = -3.56944 \times 10^{-3}$$

$$\delta_{22} = (\delta_{31}w_{21} + \delta_{32}w_{22} + \delta_{33}w_{23}) \frac{\partial f}{\partial x_2} = 7.66593 \times 10^{-4}$$

$$\delta_{23} = (\delta_{31}w_{31} + \delta_{32}w_{32} + \delta_{33}w_{33}) \frac{\partial f}{\partial x_3} = 7.01470 \times 10^{-3}$$

Step 8: We calculate the changes in weights, Δv_{ij}, and Δw_{jk}. We arbitrarily set the learning rate, η, to 0.9, and the momentum coefficient, α, to 0.7. Thus, for Δv_{ij}:

$$\Delta v_{11} = \eta \delta_{21} a_1 + \alpha \Delta v_{11,\text{old}} = (0.9)(-0.003569)(1) + 0 = -0.00321249$$

$$\Delta v_{12} = \eta \delta_{22} a_1 + \alpha \Delta v_{12,\text{old}} = (0.9)(0.000766594) + 0 = 6.899346 \times 10^{-4}$$

$$\Delta v_{13} = \eta \delta_{23} a_1 + \alpha \Delta v_{13,\text{old}} = (0.9)(-0.0070147)(1) + 0 = 6.31323 \times 10^{-3}$$

On this first time step, $\Delta v_{ij,\text{old}}$ (from the "previous step") is zero since no previous step exists. We continue this procedure for all Δv_{ij}'s, and end up with:

$$\Delta v_{ij} = \begin{vmatrix} -3.21 x 10^{-3} & 6.90 x 10^{-4} & 6.31 x 10^{-3} \\ -3.21 x 10^{-3} & 6.90 x 10^{-4} & 6.31 x 10^{-3} \\ -3.21 x 10^{-3} & 6.90 x 10^{-4} & 6.31 x 10^{-3} \end{vmatrix}$$

We also calculate the weight change Δw_{jk} using the same learning rate (η = 0.9) and momentum coefficient (α = 0.7):

$$\Delta w_{11} = \eta \delta_{31} b_1 + \alpha \Delta w_{11,\text{old}} = (0.9)(4.731 \times 10^{-2})(0.8175745) + 0 = 3.4811 \times 10^{-2}$$

$$\Delta w_{12} = \eta\,\delta_{32} b_1 + \alpha \Delta w_{12,old} = (0.9)(-0.13524)(0.8175745) + 0 = 9.951310 \times 10^{-2}$$

$$\Delta w_{13} = \eta\,\delta_{33} b_1 + \alpha \Delta w_{13,old} = (0.9)(-0.08849)(0.8175745) + 0 = -6.511 \times 10^{-2}$$

Again, on this first time step $\Delta w_{ij,old} = 0$ since no previous time step exists. We continue this procedure for all Δw_{ij}'s to yield:

$$\Delta w_{jk} = \begin{vmatrix} 0.0348 & -0.0995 & -0.0651 \\ 0.0213 & -0.0609 & -0.0398 \\ 0.0348 & -0.0995 & 0.0651 \end{vmatrix}$$

Step 9: Knowing the weight changes, we update the weights. The old and new weights are:

$$v_{ij,old} = \begin{vmatrix} -1.0 & -0.5 & 0.5 \\ 1.0 & 0.0 & -0.5 \\ 0.5 & -0.5 & 0.5 \end{vmatrix} \qquad v_{ij,new} = \begin{vmatrix} -1.003 & -0.4993 & 0.5063 \\ 0.997 & 0.0007 & -0.4937 \\ 0.497 & -0.4993 & 0.5063 \end{vmatrix}$$

$$w_{jk,old} = \begin{vmatrix} -1.0 & -0.5 & 0.5 \\ 1.0 & 0.0 & 0.5 \\ 0.5 & -0.5 & 0.5 \end{vmatrix} \qquad w_{jk,new} = \begin{vmatrix} -0.9652 & -0.5995 & 0.4349 \\ 1.0213 & -0.0609 & 0.4602 \\ 0.5348 & -0.5995 & 0.4349 \end{vmatrix}$$

We have now completed one time step. We repeat steps 2-9 until the error is sufficiently low. With this gradient-descent algorithm, we need 784 time steps to achieve less than a 1% error on variable d_1 (i.e. $e_1 < 0.01$). The results are:

Number of time steps: 784
Desired values: $d_1 = 1$ $d_2 = 0$ $d_3 = 0$
Actual values: $c_1 = 0.9900$ $c_2 = 0.0099$ $c_3 = 0.0099$

Percent error: e_1 = -1.00% e_2 = 0.99% e_3 = 0.99%

The superiority of the GDR algorithm versus the vanilla backpropagation algorithm is evident: the vanilla backpropagation algorithm required *3860 time steps* to achieve a 1% error on d_1, while the GDR algorithm only required 784. That is almost an 80% decrease in time steps. In addition, the GDR algorithm yielded lower errors for both d_2 and d_3.

Note that above, we used only one data point (input pattern). The GDR algorithm can accommodate, and indeed is made for, multiple input patterns. Refer to Venkatasubramanian et. al. (1990) for use of the GDR with multiple input patterns. Table 17.3 shows a BASIC program for training using the GDR algorithm.

Table 17.3. Code for the Generalized Delta Rule (GDR) Algorithm. Written in BASIC.

```
1 REM *** GENERALIZED DELTA RULE ALGORITHM  ***
2 REM ***      WRITTEN IN BASIC             ***
3 REM ******************************************
4 REM  * VARIABLES AND THEIR MEANINGS          *
5 REM  * ALPHA = COEFFICIENT OF MOMENTUM       *
6 REM  * A(I) = OUTPUT FROM NODES IN LAYER 1   *
7 REM  * B(I) = OUTPUT FROM NODES IN LAYER 2   *
8 REM  * BETA = LEARNING RATE                  *
9 REM  * C(I) = OUTPUT FROM NODES IN LAYER 3   *
10 REM  * D(I) = DESIRED OUTPUT FROM LAYER 3   *
11 REM * DELTA(I,J) = GRADIENT DESCENT TERM    *
12 REM *      FOR THE I-TH NODE IN J-TH LAYER  *
13 REM * DELTA(I,J) = GRADIENT DESCENT TERM    *
14 REM *      FOR THE I-TH NODE IN J-TH LAYER  *
15 REM * DELTAV(I,J) = CHANGE IN WEIGHT V(I,J) *
16 REM * DELTAW(J,K) = CHANGE IN WEIGHT W(J,K) *
17 REM * DERV(I,J) = DERIVATIVE OF SIGMOID     *
18 REM *      FUNCTION FOR THE TOTAL INPUT TO  *
19 REM *      THE I-TH NODE IN J-TH LAYER      *
20 REM * PNPUT(I) = INPUT VECTOR INTO LAYER 1  *
21 REM * SUMB(J) = POST-BIAS INPUT FOR NODE J  *
22 REM *           IN THE HIDDEN LAYER         *
23 REM * SUMC(K) = POST-BIAS INPUT FOR NODE K  *
24 REM *           IN THE OUTPUT LAYER         *
25 REM * SUMDELTA(K) = WEIGHTED SUM OF GRADIENT*
26 REM *        DESCENT TERM FROM OUTPUT LAYER *
27 REM * T(I,J) = BIAS FUNCTION FOR J-TH NODE  *
28 REM *          IN THE I-TH LAYER            *
29 REM * V(I,J) = LAYER 1-2 CONNECTION WEIGHTS *
30 REM * W(J,K) = LAYER 2-3 CONNECTION WEIGHTS *
31 REM * X(I) = POST-BIAS INPUT, I-TH NODE     *
32 REM *****************************************
33 DIM A(3),B(3),C(3),D(3),DELTA(3,3)
34 DIM DERV(3,3),PNPUT(3),SUMB(3)
35 DIM SUMC(3),T(3,3),V(3,3)
36 DIM W(3,3),DELTAV(3,3),DELTAW(3,3)
37 DIM SUMDELTA3(3)
38 REM *** STEP 1: ASSIGN WEIGHTS              ***
39 V(1,1) = -1 : V(1,2) = -.5 : V(1,3) = .5
40 V(2,1) = 1  : V(2,2) = 0   : V(2,3) = -.5
41 V(3,1) = .5 : V(3,2) = -.5 : V(3,3) = .5
42 W(1,1) = -1 : W(1,2) = -.5 : W(1,3) = .5
43 W(2,1) = 1  : W(2,2) = 0   : W(2,3) = .5
44 W(3,1) = .5 : W(3,2) = -.5 : W(3,3) = .5
45 T(1,1) = 0  : T(1,2) = 0   : T(1,3) = 0
46 T(2,1) = 1  : T(2,2) = 1   : T(2,3) = 1
47 T(3,1) = 1  : T(3,2) = 1   : T(3,3) = 1
48 PNPUT(1) = 300 : PNPUT(2)=100: PNPUT(3)=200
49 D(1) = 1 : D(2) = 0 : D(3) = 0
50 ETA = .9 : ALPHA = .6
51 REM ***  STEP 2: INTRODUCE INPUT VECTOR.  ***
52 REM *** CALCULATE OUTPUT FROM FIRST LAYER ***
53   FOR I = 1 TO 3
54   X(I) = PNPUT(I) - T(1,I)
55   A(I) = 1/(1+EXP(-X(I)))
56   NEXT I
57 REM *** STEP 3: FIND HIDDEN LAYER OUTPUTS  ***
58   FOR J = 1 TO 3
59   SUMB(J) = 0
60      FOR I = 1 TO 3
61      SUMB(J)=SUMB(J)+V(I,J)*A(I)
62      NEXT I
63    SUMB(J) = SUMB(J)+T(2,J)
64   B(J) = 1/(1+EXP(-(SUMB(J))))
65   NEXT J
66 REM *** STEP 4: FIND LAYER THREE OUTPUTS  ***
66   FOR K = 1 TO 3
67   SUMC(K) = 0
68      FOR J = 1 TO 3
70      SUMC(K)=SUMC(K)+W(J,K)*B(J)
71       NEXT J
72   SUMC(K) = SUMC(K)+T(3,K)
73   C(K) = 1/(1+EXP(-(SUMC(K))))
74   NEXT K
75 DIF1=D(1)-C(1): DIF2=D(2)-C(2): DIF3=D(3)-C(3)
76 IF ABS(DIF1) < .01 THEN END
77 REM ***     STEP 5: FIND SQUARED ERROR     ***
78 SQERROR = 0
79   FOR K = 1 TO 3
80   SQERROR= SQERROR+((D(K)-C(K))^2
81   NEXT K
82 REM *** STEP 6: FIND THE GRADIENT DESCENT  ***
83 REM *** TERM FOR OUTPUT LAYER NODES        ***
84   FOR K = 1 TO 3
85          DERV(3,K)    =    EXP(-SUMC(K))/((1
+EXP(-SUMC(K)))^2
86   DELTA(3,K) = (D(K)-C(K))*DERV(3,K)
87   NEXT K
88 REM *** STEP 7: FIND THE GRADIENT DESCENT  ***
89 REM *** TERM FOR HIDDEN LAYER NODES        ***
90  FOR J = 1 TO 3
91       DERV(2,J)    =    EXP(-SUMB(J))/((1    +
EXP(-SUMB(J)))*(1 + EXP(-SUMB(J))))
92  SUMDELTA3(J) = 0
93    FOR K = 1 TO 3
94   SUMDELTA3(J)=SUMDELTA3(J)+DELTA(3,K)*W(J,K)
95    NEXT K
96  DELTA(2,J) = DERV(2,J)*SUMDELTA3(J)
97  NEXT J
98 REM *** STEP 8: FIND DELTA V AND DELTA W   ***
99   FOR I = 1 TO 3
100    FOR J = 1 TO 3
101      DELTAV(I,J)=ETA*DELTA(2,J)*A(I)+ALPHA*
DELTAV(I,J)
102      DELTAW(I,J)=ETA*DELTA(3,J)*B(I)+ALPHA*
DELTAW(I,J)
103 REM *** STEP 9: UPDATE WEIGHTS             ***
104    V(I,J) = V(I,J) + DELTAV(I,J)
105    W(I,J) = W(I,J) + DELTAW(I,J)
106    NEXT J
107  NEXT I
108 GOTO 53
109 REM *** THE USER IS ENCOURAGED TO PUT PRINT
110 REM *** AND STOP STATEMENTS IN PROGRAM    ***
```

I. Comments on Backpropagation

The backpropagation algorithm, discussed in sections 17.2D and G, will minimize or eliminate the error between the input and desired output. In addition, backpropagation can handle highly complex, nonlinear relations between the input and the output. It is flexible, and can store many mapping patterns for its relatively small size.

However, backpropagation does have some limitations. First and foremost is the time needed to develop the computer code and train the network. The initial programming alone (including a good user-interface) is a major time investment. In addition, once the program is in place, the actual training may require tens of thousands of time steps, making the development stage slow and tedious.

Reducing the training time is one major goal of current ANN research. This research effort has both commercial and academic thrusts. Commercially, some computer companies are selling "preprogrammed" ANNs that require only the training stage. The user-interface, and the training and recall algorithms are bought in one software package. These software packages eliminate the need for initial programming, albeit at a cost.

Academically, approaches to reducing the training time can be broken into the following categories:

- Adjusting the topology of the network (e.g., pruning nodes or connections).

- Adjusting the learning rate (e.g., adjusting proportionality constants or weights to improve convergence).
- Using alternate threshold functions (e.g., use of the gaussian function or hyperbolic tangent).

Some researchers have recommended heuristics to prune the number of nodes in a layer (Yu and Teh, 1988). Others suggest heuristics for adjusting the learning rate of specific nodes (Vogl et. el., 1988). Stornetta and Huberman (1987) speeded up learning process by using a sigmoid threshold function with limiting values of [-1, +1]. Weiland and Leighton (1988) propose the use of a *shaping schedule*, where the initial training is done with data that are quickly and easily mapped. Once the ANN begins to "take shape" with respect to its weights, the process switches to the more difficult mapping problems. Barto et. al. (1983) recommend "freezing" some weights while allowing others to vary. Others have investigated the effect of the number of nodes in the hidden layer on learning rate (Hoskins and Himmelblau, 1988), but few have come up with an *a priori* way to determine the optimal number of hidden-layer nodes.

Another technique to reduce the training time, discussed in section 17.2G, is ***momentum***. Momentum is simply adding weight to the calculated adjustment to speed up the learning rate. Usually, the momentum term is a fractional value of the weight calculated from the previous training phase. Rumelhart et. al. (1986) discuss the effects of momentum.

A very common technique to decrease training time is the use of *gradient-descent methods*. These methods, discussed in section 17.2D-E and G-H, have received a great deal of attention by researchers. Gradient-descent methods calculate the new weight systematically rather than arbitrarily. Specifically, they use Newton's method to drive the error down to a minimum through descending down the gradient of the error surface. While gradient-descent methods have improved ANN training rates, the consensus is still that ANN training is too slow and more is required to speed up the process.

Besides the slow training time and the associated limitation, ANNs have another potential difficulty: they cannot guarantee that the *global* minimum in error can be found. Backpropagation may only arrive at a *local* minimum. In addition, the algorithm can lead to oscillations in the weights (particularly when many local minima exist), further extending the training stage.

The inability to guarantee the global minimum is inherent in perceptron ANNs. There is really very little we can do about it now, but it is a topic of research. Sandon and Uhr (1988) have suggested a heuristic approach to selecting weight factors to prevent getting "trapped" at a local minimum. In addition, Wasserman (1988) has described a combined stochastic-backpropagation technique that uses statistics with backpropagation squared-error reduction to drive the network to the global minimum.

J. Practice Problems

1. The examples shown for the vanilla backpropagation algorithm (section 17.2E) and the generalized delta-rule algorithm (section 17.2G) both have a potential problem, and we must fix it. If I_1 = 300 °F, then $a_1 = 1$. If I_1 = 150 °F, then $a_1 = 1$ again. Thus, we see that the sigmoid function is *insensitive* to changes in input, especially when $I_1 > 2$.

To fix this problem, some researchers recommend *normalizing* the input value to the interval [-1,1]. We will do just that. The new input-output relations are:

Input:	I_1 = 300 °F/1000 = 0.3	*Output:*	c_1 = 1 (low conversion)
	I_2 = 100 psia/1000 = 0.1		c_2 = 0 (no problem)
	I_3 = 200 lb/min/1000 = 0.2		c_3 = 0 (no problem)

With this new input-output pattern:

(a) Use the vanilla backpropagation algorithm to train the ANN to less than 1% error on d_1. How many time steps does it take? (Use the program in Table 17.2.)

(b) Use the generalized delta-rule algorithm to train the ANN to less than 1% error on d_1. How many time steps does it take? (Use the program in Table 17.3.)

2. Use the GDR algorithm of Section 17.2G to train an ANN with multiple

input vectors. Refer to Venkatasubramanian et. al. (1990) for details. We use three sets of input-output data, listed below:

Input:	I_1 = 300 °F/1000 = 0.3	*Output:*	C_1 = 1 (low conversion)
	I_2 = 100 psia/100 = 1.0		C_2 = 0 (no problem)
	I_3 = 200 lb/min/1000 = 0.2		C_3 = 0 (no problem)
Input:	I_1 = 325 °F/1000 = 0.325	*Output:*	C_1 = 1 (low conversion)
	I_2 = 90 psia/100 = 0.9		C_2 = 0 (no problem)
	I_3 = 200 lb/min/1000 = 0.2		C_3 = 0 (no problem)
Input:	I_1 = 350 °F/1000 = 0.35	*Output:*	C_1 = 1 (low conversion)
	I_2 = 80 psia/100 = 0.8		C_2 = 0 (no problem)
	I_3 = 200 lb/min/1000 = 0.2		C_3 = 0 (no problem)

3. Write a program that will perform *recall* for the ANNs trained in problem one. The recall program must:

(1) Prompt the user for the input vector.

(2) Use the weights derived from training to calculate outputs from the input, hidden, and output layers.

(3) Report the result from the output layer to the user.

4. Create a fully operational ANN, performing qualitative interpretation of process data from a chemical reactor using the same three-layer, three-input, three-output ANN used throughout the chapter. This process integrates problems 2 and 3 above into a single system. Write a program, in the computer language of your choice, that achieves the objectives

described below.

a. *Develop the training section* - use the GDR algorithm, capable of training with multiple input vectors (problem 2 above).

b. *Develop the recall section* - achieve the three objectives of problem 3 above, i.e., prompt the user for the input; calculate outputs from the input, hidden, and output layers; and report the result to the user.

c. *Train the network*- use the following data to train the network:

Correct reactor operating zone:

Input:	I_1 = 400 °F/1000 = 0.40	*Output:*	c_1 = 0 (no problem)
	I_2 = 100 psia/100 = 1.0		c_2 = 0 (no problem)
	I_3 = 200 lb/min/1000 = 0.2		c_3 = 0 (no problem)
Input:	I_1 = 420 °F/1000 = 0.42	*Output:*	c_1 = 0 (no problem)
	I_2 = 100 psia/100 = 1.0		c_2 = 0 (no problem)
	I_3 = 200 lb/min/1000 = 0.2		c_3 = 0 (no problem)
Input:	I_1 = 380 °F/1000 = 0.38	*Output:*	c_1 = 0 (no problem)
	I_2 = 100 psia/100 = 1.0		c_2 = 0 (no problem)
	I_3 = 200 lb/min/1000 = 0.2		c_3 = 0 (no problem)
Input:	I_1 = 400 °F/1000 = 0.40	*Output:*	c_1 = 0 (no problem)
	I_2 = 110 psia/100 = 1.1		c_2 = 0 (no problem)
	I_3 = 200 lb/min/1000 = 0.2		c_3 = 0 (no problem)
Input:	I_1 = 400 °F/1000 = 0.40	*Output:*	c_1 = 0 (no problem)
	I_2 = 90 psia/100 = 0.9		c_2 = 0 (no problem)

	I_3 = 200 lb/min/1000 = 0.2		c_3 = 0 (no problem)

Input:	I_1 = 400 °F/1000 = 0.40	*Output:*	c_1 = 0 (no problem)
	I_2 = 100 psia/100 = 1.0		c_2 = 0 (no problem)
	I_3 = 220 lb/min/1000 = 0.22		c_3 = 0 (no problem)
Input:	I_1 = 400 °F/1000 = 0.40	*Output:*	c_1 = 0 (no problem)
	I_2 = 100 psia/100 = 1.0		c_2 = 0 (no problem)
	I_3 = 180 lb/min/1000 = 0.18		c_3 = 0 (no problem)

Sintering of Catalyst

Input:	I_1 = 550 °F/1000 = 0.55	*Output:*	c_1 = 0 (no problem)
	I_2 = 100 psia/100 = 1.0		c_2 = 0 (no problem)
	I_3 = 200 lb/min/1000 = 0.20		c_3 = 1 (sintering)
Input:	I_1 = 525 °F/1000 = 0.525	*Output:*	c_1 = 0 (no problem)
	I_2 = 100 psia/100 = 1.0		c_2 = 0 (no problem)
	I_3 = 180 lb/min/1000 = 0.18		c_3 = 1 (sintering)

Low catalyst selectivity

Input:	I_1 = 400 °F/1000 = 0.40	*Output:*	c_1 = 0 (no problem)
	I_2 = 100 psia/100 = 1.0		c_2 = 1 (low selectivity)
	I_3 = 250 lb/min/1000 = 0.25		c_3 = 0 (no problem)
Input:	I_1 = 430 °F/1000 = 0.40	*Output:*	c_1 = 0 (no problem)
	I_2 = 100 psia/100 = 1.0		c_2 = 1 (low selectivity)
	I_3 = 230 lb/min/1000 = 0.23		c_3 = 0 (no problem)

Low conversion

Input:	I_1 = 370 °F/1000 = 0.37	*Output:*	c_1 = 1 (low conversion)
	I_2 = 100 psia/100 = 1.0		c_2 = 0 (no problem)
	I_3 = 200 lb/min/1000 = 0.20		c_3 = 0 (no problem)
Input:	I_1 = 380 °F/1000 = 0.38	*Output:*	c_1 = 1 (low conversion)
	I_2 = 100 psia/100 = 1.0		c_2 = 0 (no problem)
	I_3 = 180 lb/min/1000 = 0.18		c_3 = 0 (no problem)

d. *Perform recall*- based on the input data below, what is the output from the ANN? What potential problems could arise from these operating conditions?

Input:	I_1 = 418 °F/1000 = 0.418	*Output:*	c_1 = ?
	I_2 = 102 psia/100 = 1.02		c_2 = ?
	I_3 = 190 lb/min/1000 = 0.19		c_3 = ?
Input:	I_1 = 381 °F/1000 = 0.381	*Output:*	c_1 = ?
	I_2 = 92 psia/100 = 0.92		c_2 = ?
	I_3 = 185 lb/min/1000 = 0.185		c_3 = ?

17.3 APPLICATIONS OF ARTIFICIAL NEURAL NETWORKS

Applications of ANNs to chemical engineering have increased significantly since 1988. One of the first application papers was by Hoskins and Himmelblau (1988), who applied an ANN to fault diagnosis. Since then, the number of research publications on ANN applications in chemical engineering has risen astronomically. Bhagat (1990) has written an introductory article on ANN applications in chemical engineering, which appears in *Chemical Engineering Progress*. Samdani (1990) has written a similar which appears in *Chemical Engineering*.

To date, ANN applications in chemical engineering cover four major areas: 1) process control, 2) fault diagnosis, 3) dynamic modeling, and 4) forecasting. The following sections discuss each area in turn.

A. Process Control

Process control has been by far the most popular area for ANN applications. We divide this section on ANN applications to process control into the following categories:

- *adaptive control;*,
- qualitative interpretation of process data for the purpose of control; and
- sensor-failure detection.

These application categories are discussed below.

1. Adaptive Control

Adaptive control continuously monitors on-line process data, and based on that information, retunes the controller for optimal performance. Essentially, this technique adjusts controller tuning parameters based on process performance. One of the biggest challenges in adaptive control is *process identification*, where we must determine process dynamics at different operating conditions, and subsequently decide how to adjust the controller based on those dynamics. ANNs can certainly help here.

a. ANN-based Controller

Psaltis et. al. (1988) describe an ANN-based controller. This controller can operate in a feedforward or feedback environment, and it performs adaptive control. They suggest the use of a three-layer sigmoid threshold ANN, as shown in Figure 17.7, and recommend back-propagation training. They use a least-squared-error training algorithm, and suggest a gradient-descent method for adjusting the weight, w_{ij}. In addition, they introduce two different types of learning:

- *Generalized Learning*- to tune the controller over a wide domain of operational parameters. Hopefully, this wide-domain tuning will give the controller the robustness it needs to respond properly under any possible operating conditions.

- *Specialized Learning-* to improve controller performance for a specific operating range. Hopefully, this narrow-domain tuning will give the controller superior performance under operating conditions that the controller normally sees.

The work of Psaltis et. al. is conceptual, but it does demonstrate the versatility of ANNs in process control. In addition, the concepts of generalized and specialized learning are novel, and they reveal one potential pitfall we can run into: *an ANN may get "biased" by training with a lot of data in a narrow variable domain, and consequently may be unable to perform well outside that domain.*

Figure 17.8 shows a closed-loop control diagram using an ANN-based controller. To train the ANN, we must propagate the error backwards through all calculations, determine nodal errors relative to desired outputs, and adjust the weights accordingly. A fundamental problem arises from Figure 17.8: *it is difficult, if not impossible, to propagate the error back through an actual process*. We do not have a good process model. This problem must be overcome for an ANN to serve as a controller.

b. ANNs in Self-Tuning Nonlinear Systems

Nguyen and Widrow (1990) attempt to overcome the problem of backpropagation through the process. They have developed another form of adaptive control using ANNs: *self-tuning* in nonlinear systems. What is

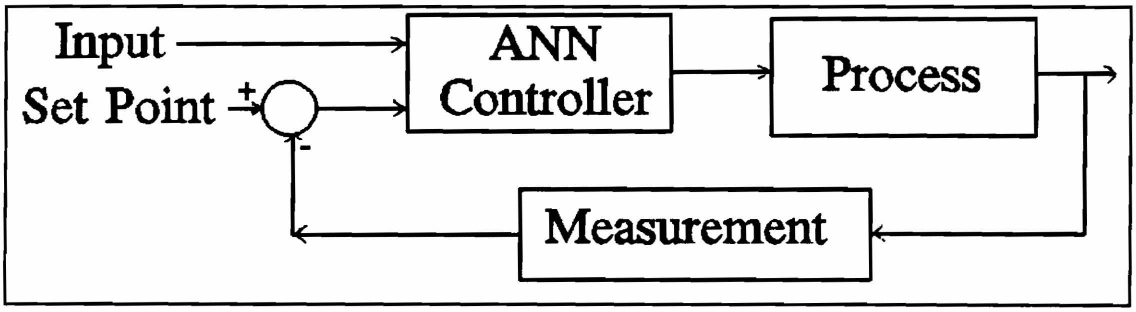


Figure 17.8. An ANN in a control loop.

self-tuning? A self-tuning controller can adjust its own tuning parameters, usually on-line, based on changes in process conditions or dynamics.

Some mathematical theory exists for self-tuning controllers, but these analytical techniques require linear control problems. The real world, of course, is nonlinear, and nonlinear self-tuning is a very complex area. One advantage of neural networks is that nonlinear problems pose no special obstacles to training the network.

As mentioned, Nguyen and Widrow (1990) attempt to solve the problem of trying to backpropagate through a process. They first recommend using a *separate* ANN to emulate the process. This step is equivalent to process identification in control theory, except that here an ANN models the nonlinear process. They use backpropagation to achieve a least-squared-error, and use a gradient-descent method to train the ANN to perform process emulation.

Once the emulator adequately models the process dynamics, it can be used to train the controller. Training the controller this way overcomes the problem of backpropagation through a nonlinear process, at least in principle. Nguyen and Widrow's work addresses ANNs in control theory in general, and is not restricted to chemical process control (CPC). In CPC, their approach might be feasible to implement in a real-time environment.

c. ANNs in Model-Based Control

ANNs have also been applied to ***model-based control***. To understand model-based control, let us contrast it with traditional control. In traditional controller design, we develop the control strategy based on experience and knowledge about the chemical process. Form this knowledge, and possibly some open- or closed-loop response information, we can *tune* the controller.

In model-based control, we use a process model explicitly to aid in controller design. There are a host of different ways to implement model-based control. With a model emulating the process, we can:

- properly design a traditional control system;
- investigate advanced control strategies off-line;
- place the model directly into the control loop, and use it on-line for advanced control strategies and predictive control.

Psichogios and Ungar (1990) use ANNs for what they call *direct* and *indirect* model-based control. In *direct* control, the ANN is trained with process input-output data. Given the current state and the desired state of the process, the ANN will adjust its output to drive the process into the desired state. Thus, in direct control, the ANN controller takes corrective action based on process deviation.

In *indirect* control, the controller functions in a *predictive* mode. Given the current state of the process and the current controller output, the ANN predicts the process response. Thus, in indirect control, the ANN acts as a process model and predictor.

Psichogios and Ungar compare ANN-based approaches for direct and indirect control using two model-based control strategies: internal-model control (IMC) and model-predictive control (MPC). We will briefly discuss IMC and MPC before moving into Psichogios and Ungar's application.

Internal-model control (Morari and Zafiriou, 1989; Rivera et. al., 1986) is an advanced feedback-control strategy that uses a model on-line, built into the control loop. Controller settings are adjusted based on the model in a straightforward manner (Seborg et. al., 1989). IMC is advantageous because it: readily supports PID controller implementation, takes into account model uncertainty, and can be adjusted to balance controller performance with control system robustness (when either modeling errors or changes in process dynamics occur). Figure 17.9 contrasts the traditional feedback control with IMC.

Model-predictive control (MPC) uses a model to predict process response over a longer period of time (typically at least as long as the open-loop response of the system). MPC applications include both single-input, single-output (SISO) systems, as well as multiple-input, multiple-output (MIMO) systems. Based on predictions about where the process is

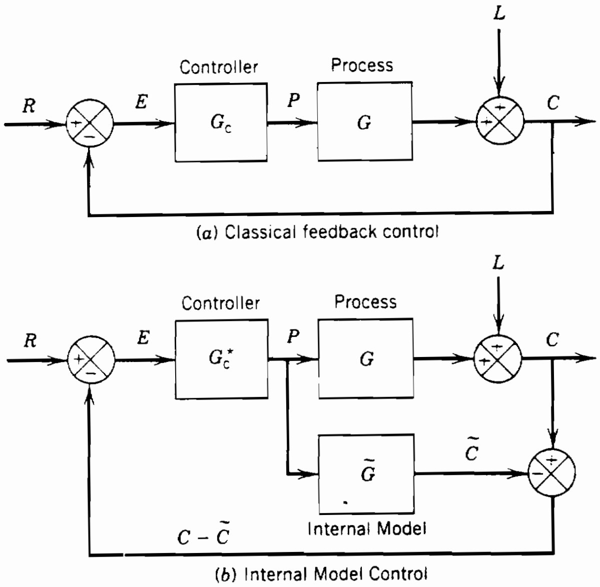


Figure 17.9. Block diagrams of (a) classical feedback control, and (b) internal model control.

headed, MPC can adjust numerous variables, such as controller tuning parameters, sampling time, predicted errors, prediction horizon (how far ahead we should predict process performance), and control horizon (how far ahead we should try to control). On the down side, MPC strategies are computationally intensive and difficult to implement in real-time.

Psichogios and Ungar use ANNs for both IMC and MPC. For the IMC neural network, they used a gradient-descent method to train it, taking approximately 1500 passes through the data set. The controller showed good disturbance rejection and no steady-state offset, but the response was oscillatory. Under MPC (which requires much more computation), the ANN controller regulated the process well and exhibited no oscillations.

Psichogios and Ungar applied both IMC and MPC on an ANN trained for a *linear* response. A linear model mapped the input-output pattern. They trained the ANN with data from this model, resulting in a "linear ANN model" rather than a nonlinear ANN emulating the real process. The result? Both ANNs failed to control the process, yielding either wild oscillation or substantial offset. Thus, Psichogios and Ungar identify some important trade-offs:

(1) training "nonlinear" ANNs to model an actual process is slow and cumbersome, but results in fairly good control;
(2) training "linear" ANNs based on a linear model is much more rapid (more than one order-of-magnitude faster), but results in unacceptable control;
(3) properly training an ANN requires significant amounts of data;
(4) researchers must address both stability and the ANN's tendency to "forget" training examples from the past for ANN-based control to be practical. (When ANNs are updated with more recent data, the new connection weights may not sufficiently reflect important properties of the process derived from past experience and data.)

d. Pattern-Based Control

Cooper at. al. (1990) utilize an ANN to recognize patterns, and propose adaptive control based on controller-error pattern recognition. They adjust the gain in the controller model and match the error response of the controller with a target pattern. When a disturbance arises, they use an ANN to determine if the disturbance can change the character of the process, or if the error patterns simply reflect the disturbance itself. They demonstrate this adaptive control using a traditional proportional-integral (PI) controller, as well as what they call a *generalized predictive controller* (GPC). A GPC is a self-tuning feedback controller that uses a model to estimate the values of measured process variables. Based on the difference between where the process actually is and where the prediction estimates it will be, the controller adjusts its parameters. The work of Cooper et. al. attempts to take advantage of the pattern-recognition capabilities of ANNs. In addition, the work applies only to single-loop systems where changes in process gain outweigh those in process time-constant or dead-time.

e. Problem Diagnosis Using Historical Data

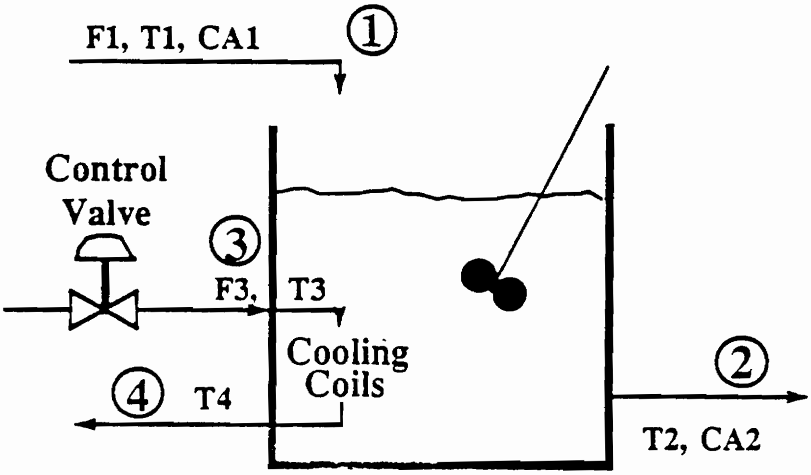
Roat et. al. (1990) have applied neural networks with what they call "system cultivation" in nonlinear control. *System cultivation*, as described by the authors, detects and diagnoses process and controller problems using historical data. They use hypothesis-feedback modeling (HFM), which attempts to relate hidden process problems and unmeasured disturbances to known and measured quantities. HFM operates on the historical database, i.e., the accumulation of the time series of process-variable measurements. HFM requires a ***reduced database***, which contains patterns of variation in the process and controller in an enhanced form. The information in this enhanced database then feeds into an ANN for training purposes. The ANN is trained using backpropagation. Once trained, the ANN can operate on future reduced databases. Since ANNs have excellent multivariable pattern-recognition properties, they can 1) discern and classify operating data corresponding to the normal operation, or 2) identify the type of problem if the operation is abnormal. Roat et. al. demonstrate the use of this technique in the control of a jacketed continuous stirred-tank reactor with a side reaction.

2. Control of a Continuous Stirred-Tank Reactor (CSTR)

Hoskins and Himmelblau (1990b) demonstrate the use of an ANN controller to operate a CSTR. Figure 17.10 shows a schematic diagram of this process.

The need for knowledge about the process to achieve control objectives poses a fundamental problem in process control. Control requires either an accurate process model or an analytical function (objective function) of the desired system behavior. Most processes, however, are nonlinear, and effectively modeling the process dynamics for control purposes is difficult. Hoskins and Himmelblau investigate an *Anderson ANN controller* (Barto et. al., 1983), which contains two ANNs: an *evaluation* network and an *action* network, as shown in Figure 17.11.

The evaluation network inputs a reinforcement signal and the process



Continuous Stirred Tank Reactor with Cooling

Reactor Streams 1 & 2

F1	Inlet flow rate, ft^3/min
T1	Inlet temperature, ^oF
CA1	Inlet concentration of reactant A, lb-moles/ft^3
T2	Outlet temperature, ^oF
CA2	Outlet concentration of reactant A, lb-moles/ft^3

Coolant Streams 3 & 4

F3	Inlet flow rate, ft^3/min
T3	Inlet temperature, ^oF
T4	Outlet temperature, ^oF

Figure 17.10. A schematic diagram of CSTR system.

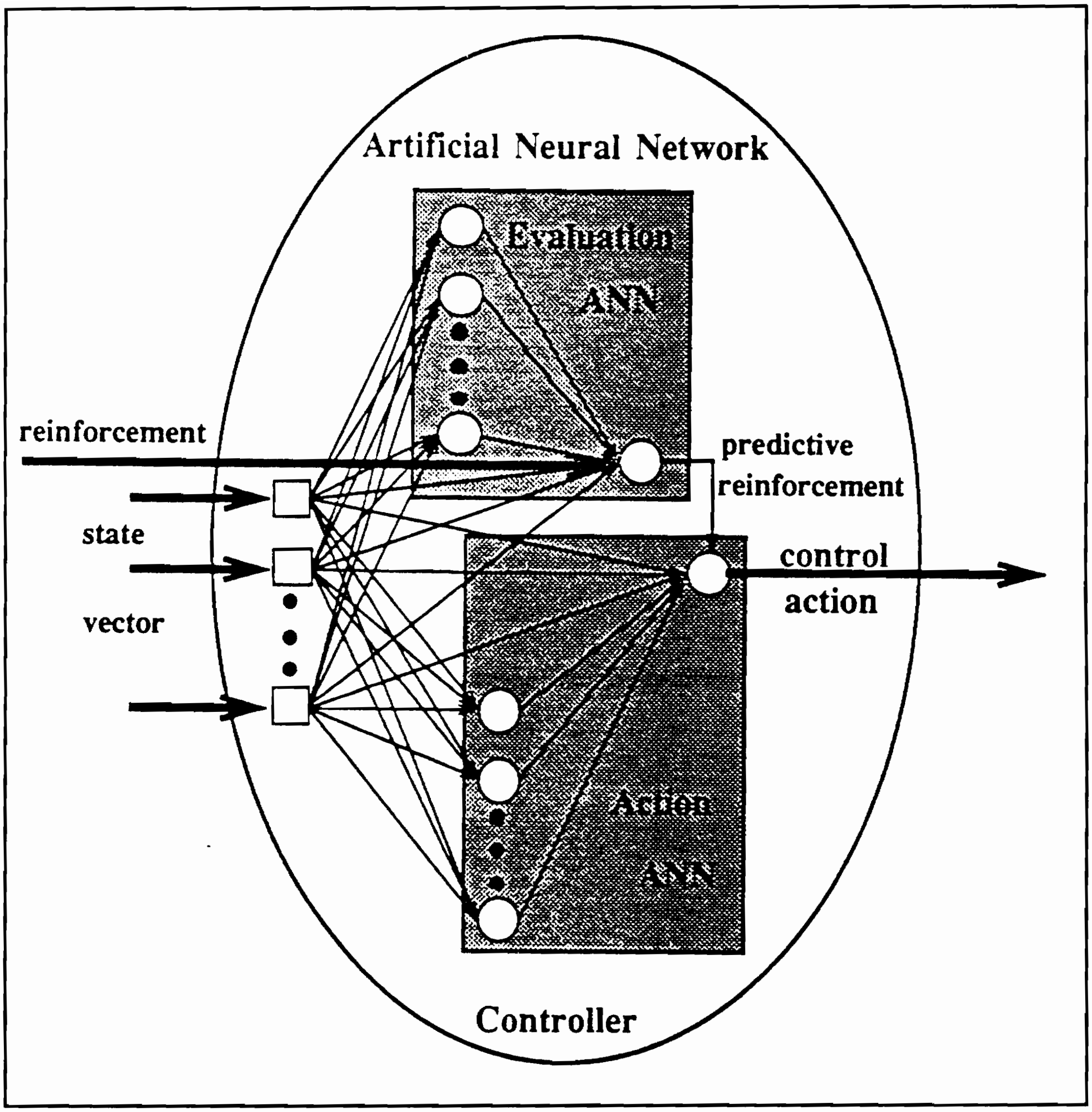


Figure 17.11. An Anderson ANN controller with an evaluation and action network.

data. From this information, the network determines whether or not the current process performance is acceptable. Based on these results, it outputs a predictive reinforcement signal that will in turn alter the controller output based on that predictive signal. The action network receives the process data from the plant and the predictive reinforcement signal from the evaluation network. The action network then computes a controller action based on these inputs. Learning in the evaluation network employs both reinforcement and backpropagation, with the reinforcement output signal itself serving as the error. Likewise, the action network uses both reinforcement and backpropagation learning. Hoskins and Himmelblau apply this controller to the operation of a jacketed CSTR, where the main goal is temperature control.

One difficulty with the approach is that the basis for the reinforcement criteria is unclear. Direct cause-and-effect relations between the reinforcement signal and controller performance do not exist. Many actions, all occurring simultaneously, affect the controller, and we cannot completely ascertain the reinforcement signal's role. Hoskins and Himmelblau's system did not perform up to the level of proportional-integral-derivative (PID) control, but it does demonstrate the usefulness of an ANN controller for an actual chemical process.

3. Qualitative Interpretation of Process Data

Whitely and Davis (1990), describe the use of ANNs for qualitative interpretation of quantitative process data. Though we can use knowledge-based systems (KBS) in process control, one key limitation is that the information processed by a KBS is generally *symbolic*, not numerical. Process data, on the other hand, is numerical, not symbolic, and translating numerical data into useful symbolic abstractions is difficult.

Whitely and Davis attempt to use ANNs to convert quantitative plant data into a qualitative interpretation. Specifically, they use an ANN to identify when a process variable is exhibiting cyclic behavior and when it is not. Thus, the ANN determines whether or not a process variable is cyclic with respect to time. They define a variable as cyclic if it exhibits a specific amplitude and frequency range for a sustained period of time. This definition itself is a limitation because it is a *very narrow* interpretation of a cyclic variable.

Through this work, Whitely and Davis identify some key challenges in using ANNs for qualitative interpretation. One problem is the continuity of the input parameter. For example, it is challenging to distinguish a "cyclic" pattern with the frequency, $\omega = 0.05001$ (cyclic, according to their definition) from $\omega = 0.04999$ (not cyclic) is challenging. Indeed, to get that level of discernment required 13.3 CPU hours of training time on a VAX 8550 mainframe computer -- not very practical.

The large number of possible input patterns presents another problem. Depending on the input patterns used for training, the network may perform very well in the narrow region it was trained for, but may not perform adequately outside of that region.

These two difficulties, continuity and the large number of input patterns, are inherent obstacles to using ANNs in qualitative interpretation, and must be reckoned with.

4. Sensor-Failure Detection

Naidu et. al. (1990) introduce the use of ANNs for sensor-failure detection. A diagnostic module in a control system detects failures in the actuators and sensors of that system. A mismatch between the model and the process may trigger false alarms, improperly indicating a sensor failure. If we then relax the criteria for firing the alarms, a critical sensor failure may go undetected. Consequently, one of the biggest problems in sensor-failure detection is modeling the process correctly. For a complex, nonlinear process, this task is extremely difficult.

Naidu et. al. attempt to overcome this problem by using ANNs to differentiate between patterns caused by sensor failures and those caused by process-model mismatch, noise, and disturbances. They use backpropagation to train the network, and compare the ANN's performance to two other fault-detection algorithms: finite integral squared-error (FISE) and nearest-neighbor (details of both appear in Nett et. al., 1988). They apply these algorithms to the internal-model control (IMC) structure for a first-order, linear, time-invariant process subjected to high model uncertainty. Their work demonstrates that the ANN gives a more accurate prediction of sensor failure than the two detection algorithms, partly because the ANNs can capture nonlinearities. In addition, ANNs can be trained on-line, and once trained, they require less calculation time than the algorithms.

Yao and Zafiriou (1990) further extend this sensor-failure detection using ANNs, but they do not use backpropagation through a standard perceptron network. Instead, they use the so-called *local receptive-field* *network* (LRFN), and demonstrate a way to prune redundant nodes in the network to improve efficiency. The LRFN shown in Figure 17.12 is a simpler network than the standard three-layer perceptron. The LRFN has just one layer, and is trained using both supervised and unsupervised learning. Yao and Zafiriou then use a technique called *singular-value decomposition* (Seborg et. al., 1989, pp.689-694) to remove redundant nodes.

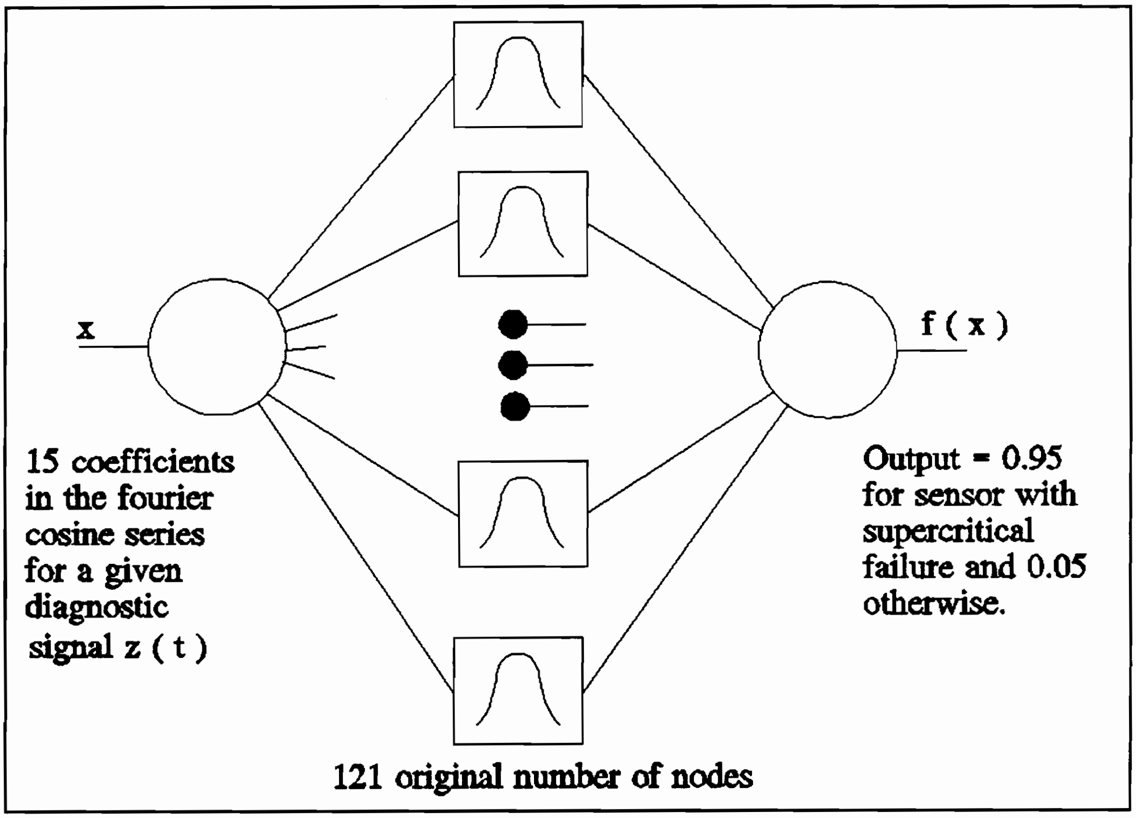

x
f (x)
15 coefficients in the fourier cosine series for a given diagnostic signal z (t)
Output = 0.95 for sensor with supercritical failure and 0.05 otherwise.
121 original number of nodes

Figure 17.12. The Local Receptive-Field Network (LRFN).

Based on this work, LRFNs hold promise for detecting sensor failure on-line. First, the networks detected failures accurately, and performed better than the backpropagation network used in Naidu et. al. (1990). Second, the LRFN (like any ANN), can be trained off-line, and then placed into on-line service. Since all the computational "cost" for ANNs occurs in the training phase and ANNs calculate results very quickly once trained, they are suitable for on-line implementation.

B. Fault Diagnosis

1. ANNs in Fault Diagnosis

ANNs are useful in fault diagnosis for three key reasons. First, through training the ANN can store knowledge about the process and learn directly from quantitative, historical fault information. We can train a network based on historically "normal" operation, and then compare that information to current operational data to determine faults.

Second, ANNs have the ability to filter noise and draw conclusions in the presence of noise. We can train ANNs to recognize important process information, and disregard noise, enabling then to function effectively in a noisy environment. This filtering capacity makes ANNs well-suited for on-line fault detection and diagnosis.

Finally, ANNs have the ability to identify causes and classify faults. Fault diagnosis requires pattern recognition, and we can train an ANN to identify "normal" patterns of operation and patterns where faults exist. Moreover, if the ANN detects a fault pattern, it can classify the fault and identify possible causes.

2. Boolean Fault Diagnosis

Venkatasubramanian and Chan (1989) introduce a method for using ANNs in fault diagnosis, and compare the results to a knowledge-based system. They diagnose faults in a fluidized, catalytic-cracking (FCC) unit, and identify eighteen symptoms (input nodes) and thirteen classifications of problems (output nodes). The hidden layer ranges from five to twenty-seven nodes. Importantly, both the input and output in this system are *Boolean*, i.e., either 0 (no fault) or 1 (fault). Thus we have an eighteen-dimension Boolean input vector, such as:

(1 0 0 0 1 0 1 0 0 1 0 1 0 0 1 0 0 0)

and a thirteen-dimension Boolean output vector, such as:

(0 0 0 0 0 0 1 0 0 0 0 0 0).

The goal is to have the network identify probable faults based on the input symptoms.

Venkatasubramanian and Chan use backpropagation training, and their system performs fairly well. It can identify the correct source of faults 94% to 98% of the time. Unfortunately, training the network requires over 8000 time steps. Also, the inputs are Boolean rather than real-time data. Thus, the "fault" and "no fault" classifications are already in place before the ANN takes over the analysis. The ANN simply maps a Boolean input to a Boolean output. Mapping continuous input variables is *far* more challenging. Aside from these limitations, Venkatasubramanian and Chan were among the first to successfully demonstrate the applicability of ANNs to pattern matching and fault diagnosis. The ANN approach has some definite advantages over traditional expert-system techniques. In principle:

- ANNs can be implemented in real-time, which has proved to be a challenging problem in expert systems.
- ANNs can directly utilize time-series data (or moving averages), while expert systems need some translation from numerical data into symbolic information.

However, while these statements are true *in principle*, implementing ANNs for real-time control is still a very challenging task.

3. Fault Diagnosis with Continuous Variables

Hoskins and Himmelblau (1990a) use ANNs for fault diagnosis with *continuous* variables as inputs. One limitation of their work, however, is that it applies to steady-state systems only. They apply ANNs to fault diagnosis in two problems: 1) three isothermal CSTRs operating in series, and 2) a chemical reactor catalytically converting heptane to toluene.

For the three CSTRs in series, Hoskins and Himmelblau use the ANN to identify six potential faults, as shown in Table 17.4. They assume that all sensors operate properly, and use the ANN to classify input patterns based on a scaler decision function. They train the network using backpropagation, and discuss the training efficiency based on the number of hidden nodes. With only six input patterns in the training set, the ANN misdiagnoses only 20% of the time. With twelve input patterns, the system diagnoses properly 100% of the time.

Table 17.4. Faults monitored in CSTR problem.

FAULT	DESCRIPTION	TYPE
A	Inlet Concentration, Component A	Low
B	Outlet Concentration, Component A	High
C	Inlet Flow Rate	Low
D	Inlet Flow Rate	High
E	Temperature	Low
F	Temperature	High

Hoskins and Himmelblau also use ANNs on a more sophisticated problem, fault diagnosis of a chemical reactor catalytically converting heptane to toluene. Here, they monitor five faults, shown in Table 17.5.

Table 17.5. Faults monitored in heptane-conversion problem.

FAULT	DESCRIPTION
1	Physical and/or chemical deterioration of catalyst, resulting in lower frequency factor.
2	Fouling of reactor heat-exchange surface, reducing h_1, the reactor heat-transfer coefficient.
3	Fouling of pre-heater heat-exchange surface, reducing h_2, the pre-heater heat-transfer coefficient.
4	Plugging of rector recycle line, leading to decreased volumetric flow to the reactor.
5	Plugging of rector recycle line, leading to decreased volumetric flow in the recycle stream.

This example clearly demonstrates that ANNs can convert *quantitative* numerical information in to *qualitative* interpretations.

Hoskins et. al. (1990c) extend this work to a much more complex process, one more typical of a unit we would see in a chemical plant. This process has a continuous stirred-tank reactor, a plug-flow reactor, and associated heat-exchange and separation equipment. Hoskins et. al. identify 19 process faults, typical for a small-to-medium sized unit in the chemical process industry. While they need a very large network requiring over 2000 time steps to train, the system seems to perform well. Typically, the system can correctly identify the fault with an 85% accuracy rate.

Another example of fault diagnosis with continuous variables is Venkatasubramanian et. al. (1990), where they also perform fault diagnosis on a CSTR. They identify six faults, and interestingly, choose nearly the exact same faults described in Hoskins and Himmelblau (1990a) and listed in Table 17.4. Not surprisingly, the two studies yield similar results. The ANN of Venkatasubramanian's et. al. performs well, with only a 3-7% error in identifying the faults. Training the network requires 417 time steps.

However, in the same work, Venkatasubramanian et. al. go one step further: they also perform fault diagnosis on a reactor-separator system. The network has eight input nodes corresponding to eight malfunctions. With 1000 time steps, the network correctly identifies six of the malfunctions. The network *fails* to identify two faults, however. To correctly identify the last two malfunctions, training requires 4000 time steps -- a 400% increase! This study reveals some important characteristics of ANNs:

- One of ANN's strengths is the ability to filter noise and still draw conclusions.
- That strength may become a *liability* when, as in this example, the network incorrectly views a faulty input pattern as noise.

4. Fault Diagnosis with Unsteady-State Systems

Vaidynathan and Venkatasubramanian (1990) expand on the work of Venkatasubramanian et. al. (1990) by moving to fault diagnosis in *dynamic*, unsteady-state systems. They use both raw time-series plant information and *average moving values* as input into the network. An average moving value takes raw data from the last *n* time series, and averages them to yield one value. They use a CSTR, and the same set of six fault classifications listed in Table 17.4. A key challenge in dynamic processes is choosing both the size of the "time window-width" (i.e., how much previous history is retained in the current calculation) and the time between data sampling. Vaidynathan and Venkatasubramanian determine these parameters empirically based on process time-constants. They choose a "window-width" of four sampling times. After 1000 time steps, the network can identify faults with less than 4% error.

Vaidynathan and Venkatasubramanian (1990) also perform *concurrent* fault analysis, i.e., identifying two faults occurring simultaneously. This task is much more challenging, and even goes beyond the ability of most human experts. Most fault-diagnosis strategies use a "fault tree" that does not take into account concurrent faults. This study tests six cases, again on the CSTR system, with the same fault pattern listed in Table 17.4. The ANN performs correctly in three cases, with errors in the other three. The results appear in Table 17.6. These results demonstrate that ANNs tend to *underreport* faults; i.e., they may view actual faults as noise and filter that noise, thereby failing to correctly report the fault.

Table 17.6. Results on concurrent faults test for the CSTR problem.

FAULTS INDUCED	DESCRIPTION	FAULTS IDENTIFIED
F3,F5	F3: Increase in Inlet Concentration F5: Increase in Inlet Temperature	F3, F5
F4,F5	F4: Decrease in Inlet Concentration F5: Increase in Inlet Temperature	F4, F5
F1,F5	F1: Increase in Inlet Flow Rate F5: Increase in Inlet Temperature	F1, F4, F5
F2,F5	F2: Decrease in Inlet Flow Rate F5: Increase in Inlet Temperature	F2, F5
F1,F6	F1: Increase in Inlet Flow Rate F6: Decrease in Inlet Temperature	F6
F2,F6	F2: Decrease in Inlet Flow Rate F6: Decrease in Inlet Temperature	F6

5. Alternatives to Backpropagation Learning for Fault Diagnosis

When used to train ANNs for fault diagnosis, backpropagation has a number of limitations:

(1) Backpropagation does not guarantee the most plausible or robust pattern recognizer possible.

(2) Based on the input data used for training, backpropagation networks may take on arbitrary characteristics, such as being accurate in local variable ranges subjected to extensive training, and inaccurate in ranges subjected to less training.

(3) Backpropagation networks lack a mechanism to detect when they may be in one of the "inaccurate" variable ranges that result from inadequate training.

To remedy some of these problems, Leonard and Kramer (1990) use *radial bias functions* in an alternate training method. Yao and Zafiriou (1990) also proposed the use of the radial bias function (see section 17.4A), and Figure 17.12 shows the network. Recall that the sigmoid threshold function, used in most backpropagation networks, is:

$$f(x) = \frac{1}{(1 + e^{-x})}$$

Instead of the sigmoid threshold function, Leonard and Kramer employ a Gaussian density function:

$$a_{hk} = \exp\left(\frac{-| X_h - X_k |}{\sigma_h^2} \right)$$

where a_{hk} is the activation of node h with given input x_k, x_h is the N-dimensional position of the center of the radial unit in the input space, and σ is a scaling parameter that determines the distance in the input space that the unit will have significant influence over. Figure 17.13 shows the Gaussian density function.

In this work, Leonard and Kramer claim that by using the radial bias function rather than the sigmoid function:

(1) the ANN can be made more robust and better able to handle test cases outside the range of training data;

(2) the ANN can estimate how close the test case is to the original classifier; and

(3) training is faster.

C. Process Modeling

Although most applications of ANNs in chemical engineering focus on process control and fault diagnosis, researchers are also working in dynamic (unsteady-state) process modeling. In fact, dynamic modeling could be considered a subset of process control since many advanced process control techniques require a process model.

Bhat and McAvoy (1990) have applied ANNs to dynamic modeling using a pH-controlled CSTR. The CSTR reacts acetic acid with sodium hydroxide to form sodium acetate. As the flow rate of sodium hydroxide fluctuates, the pH of the CSTR also fluctuates, but nonlinearly. The input is a "moving

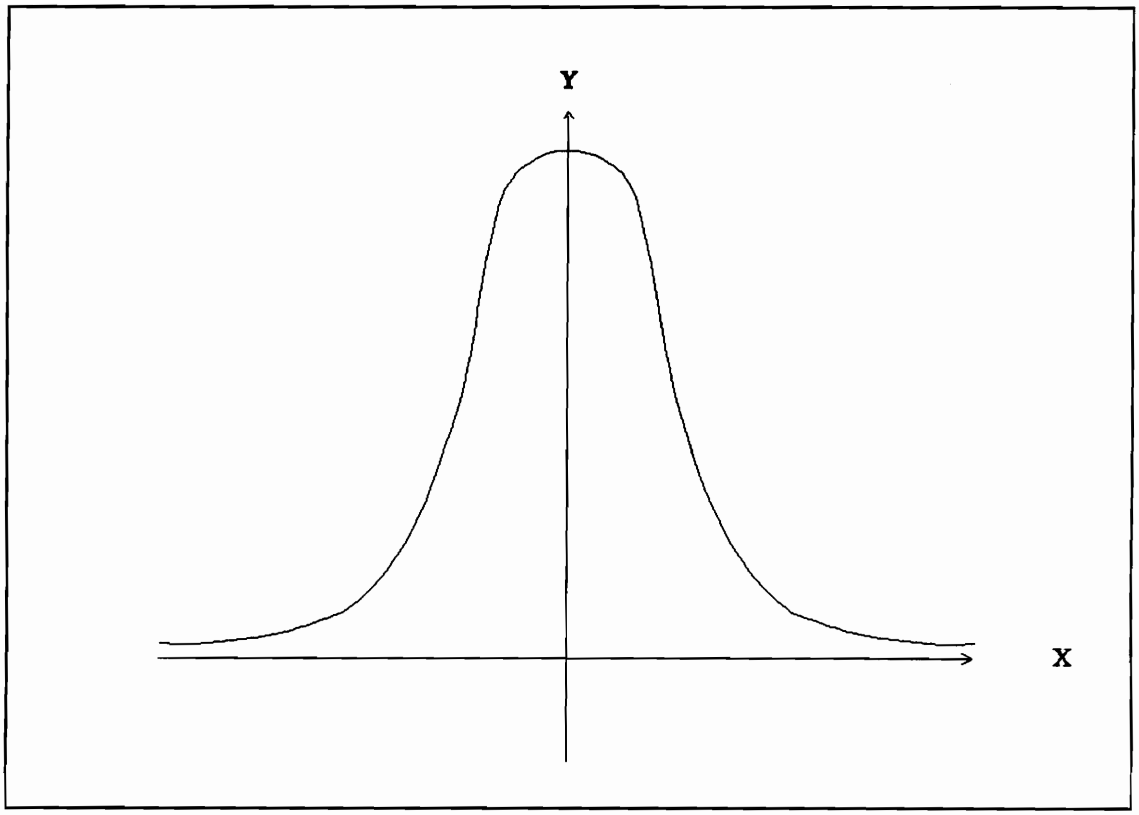


Figure 17.13. The Gaussian density function.

window" of both current and previous flow rate values of pH and sodium hydroxide. The "process window" is the input to the ANN, composed of data from the last three, four, or five sampling times. The sampling time is 0.4 minutes. They adjusted the number of input nodes based on the number of sampling times included in the input. The output from the ANN is the predicted pH. In addition to an ANN model, Bhat and McAvoy use some traditional dynamic modeling techniques and compare the two approaches. Their results demonstrate that the ANN predicted future pH more accurately, primarily because it could model system nonlinearities better.

Bhat et. al. (1990) include another modeling example, the steady-state modeling of a CSTR with reactions in series. They could predict results to less than 2% root-mean-squared error. On the downside, however, this relatively simple example took 10,000 iterations to fully train the ANN. Again, we see need to develop better training algorithms.

For an interesting application of the ANN to the modeling of electrochemical reaction kinetics, the reader may refer to Hudson et. al. (1990).

D. Process Forecasting

Forecasting is another new application of ANNs. Like process modeling, forecasting can also be viewed as a subset of process control. In forecasting, the objective is to predict the value of measured process variables in the next sampling time based on a history of noisy, "chaotic" data.

Ydstie (1990) demonstrates the use of ANNs for this purpose. In addition, he introduces a new training algorithm, called the *error-broadcast algorithm*. Ydstie attempts to address two fundamental tasks in this work:

(1) designing nonlinear, robust controllers based on ANNs with hidden layers; and

(2) modifying the tuning algorithm to achieve global parameter

convergence.

Ydstie's application uses the ANN to predict near-future process performance (within the next 1-5 sampling times). The error-broadcast algorithm is actually a "forward propagating" tool. Using previous parameter values, the training algorithm calculates the error and broadcasts it forward through the network. Based on predicted errors calculated as the training session passes through the network, the algorithm adjusts the weights. The mathematics are complicated; refer to Ydstie's paper directly for more information. Overall, the conclusions are general, and Ydstie successfully demonstrates the use of ANNs for process forecasting in a control loop. This type of "forward propagation" has not received as much attention as backpropagation among researchers, but does appear to offer some advantages.

Foster et. al. (1990) have demonstrated the use of ANNs for forecasting in short, noisy time series. They present two approaches: *direct forecasting*, where the ANN maps past data to future data for a single time series; and *network-based combining*, where a constrained network gives an optimal weighted combination of conventional forecasts.

In this work, Foster et. al. identify one problem associated with ANNs used in forecasting: a network trained from noisy past data performs poorly if that training captures spurious events inherent in the noise. The ANN effectively "over-fits" the data, and maps noise when it should not.

To eliminate this problem of over-fitting noise, Foster et. al. propose two changes. In direct forecasting, they change the network size, i.e., reduce the number of nodes. In network-based combining, they feed in the ANN time-smoothed information, such as the running average over a set of previous time periods, rather than raw data.

This study identifies the trade-offs between direct forecasting and network-based combining:

- *Expressive power*- direct forecasting is more flexible and has more expressive power, but this power can lead to overfitting.
- *Signal sensitivity*- direct forecasting is more sensitive to signals, but this sensitivity can in turn lead to noise sensitivity. Network-based combining uses abstracted, time-smoothed data, and therefore is less sensitive to signals and noise.

Overall, Foster et. el. conclude that the performance of direct forecasting using ANNs was *not* any better than traditional forecasting methods used today. The biggest problem is overfitting. Network-based combining, on the other hand, is superior to direct forecasting, and performs roughly equivalently to more traditional forecasting models.

E. Practice Problems

1. Develop an ANN for fault diagnosis using the process specifications

below. This ANN will have Boolean input and, likewise, the desired output is Boolean: 0 = no fault; 1 = fault. Refer to the model of Venkatasubramanian and Chan (1989) for details; we use their approach here. You will diagnose faults in a fluidized, catalytic-cracking (FCC) unit with eighteen symptoms (18 input nodes with values of 0 or 1) and thirteen elemental faults (13 output nodes with desired values of 0 or 1). Table 17.7 lists the representation of the input and output nodes.

Given the input and output nodes, we must relate symptoms with elemental faults. For example, if the H_2-to-CH_4 ratio is high (input node 15 = 1), and the amount of coke has increased (input node 14 = 1), then the likely fault is nickel poisoning of the catalyst (output node 11 = 1). Thus, for the ANN to successfully identify this fault, when input node 14 and 15 are one, and all other input nodes are zero, then output node 11 should be one, with all other output nodes equal to zero. The input-output pattern, then, is:

INPUT: (0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0)

OUTPUT: (0 0 0 0 0 0 0 0 0 0 1 0 0)

a. Write the computer program for the ANN topology that supports this problem. Include training and recall sections. Use the GDR algorithm for training. You will need 18 input nodes, and 13 output nodes. Design the ANN with 15 hidden-layer nodes.

b. Train the network with the sets of input-output information shown in Table 17.8. Stop iterating when the actual input is within 1% of the desired output (i.e., output value $\geq$ 0.99). There are 13 total input patterns.

Table 17.7. Classification of input and output nodes.

INPUT *(process symptoms)*		**OUTPUT** *(elemental faults)*
DESCRIPTION	**NODE**	**DESCRIPTION**
Fines content decreased slightly	1	Hole in reactor plenum
Rate of loss increasing	2	Dipleg damage
Fines content decreasing with time	3	Cyclone damage
BS&W analysis shows traces of refractory	4	Damage to regenerator grid
Regenerator grid ΔP has dropped	5	Plugged dipleg, or jammed trickle valve
Sudden high catalyst loss	6	Partial bed defluidization
Regenerator grid ΔP has increased	7	High regenerator velocity
Regenerator has abnormal temperature profiles	8	Catalyst attrition
Losses are high and steady	9	Vanadium poisoning of catalyst
Recent loss of regenerator air	10	Sodium poisoning of catalyst
Fines content increasing with time	11	Nickel poisoning of catalyst
High Vanadium on catalyst	12	Hydrothermal deactivation
High Sodium on catalyst	13	Thermal deactivation
Coke make has increased	14	---
Ratio of H_2 to CH_4 is high	15	---
Catalyst pore size has increased	16	---
Catalyst surface area has increased	17	---
Catalyst pore size remained constant	18	---

Table 17.8. Boolean training data for FCC fault-diagnosis problem.

								Node Number										
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
I:	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0:	1	0	0	0	0	0	0	0	0	0	0	0	0					
I:	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0:	0	1	0	0	0	0	0	0	0	0	0	0	0					
I:	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0:	0	0	1	0	0	0	0	0	0	0	0	0	0					
I:	0	0	0	0	1	0	0	0	1	0	0	0	0	0	0	0	0	0
0:	0	1	0	1	0	0	0	0	0	0	0	0	0					
I:	0	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0
0:	0	0	0	0	1	0	0	0	0	0	0	0	0					
I:	0	0	0	0	0	0	1	1	1	1	0	0	0	0	0	0	0	0
0:	0	0	0	0	0	1	0	0	0	0	0	0	0					
I:	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
0:	0	0	1	0	0	0	1	0	0	0	0	0	0					
I:	0	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0
0:	0	0	0	0	0	0	0	1	0	0	0	0	0					
I:	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	0	0	0
0:	0	0	0	0	0	0	0	0	1	0	0	0	0					
I:	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0
0:	0	0	0	0	0	0	0	0	0	1	0	0	0					
I:	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0
0:	0	0	0	0	0	0	0	0	0	0	1	0	0					
I:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0
0:	0	0	0	0	0	0	0	0	0	0	0	1	0					
I:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1
0:	0	0	0	0	0	0	0	0	0	0	0	0	1					

c. Now see if the network can predict *simultaneous* faults. Input the following vector to the ANN:

INPUT: (1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0)

This input vector *should* identify faults 1 and 2. What is the accuracy of the prediction? What do you recommend to improve the accuracy?

2. Let us develop an ANN to control an actual process. The most common type of ANN used in engineering applications has been the three-layer perceptron ANN of Figure 17.6, encoded with gradient-descent learning. Unfortunately, this ANN is poorly suited for real-time updating of the weights based on process dynamics. Below we discuss a new ANN, the *Adaptive Heuristic Critic*, used for on-line control.

Introduced by Barto et. al. (1983), the *Adaptive Heuristic Critic* (AHC) is a three-layer *feedforward* ANN that uses supervised learning with reinforcement in a unique topology. The output from the first layer feeds into *both* the second and third layers, and is adjusted using weights v_{ij} and w_{jk} respectively. Figure 17.14 shows the AHC topology. Note that the number of nodes in layer 2 must equal the number in layer 3.

The learning procedure for AHC is a type of reinforcement learning called *reinforcement adaptation*, and differs from standard error-correction learning. Each PE *does not* receive an individual, explicit estimate of its error as done in previous backpropagation examples.

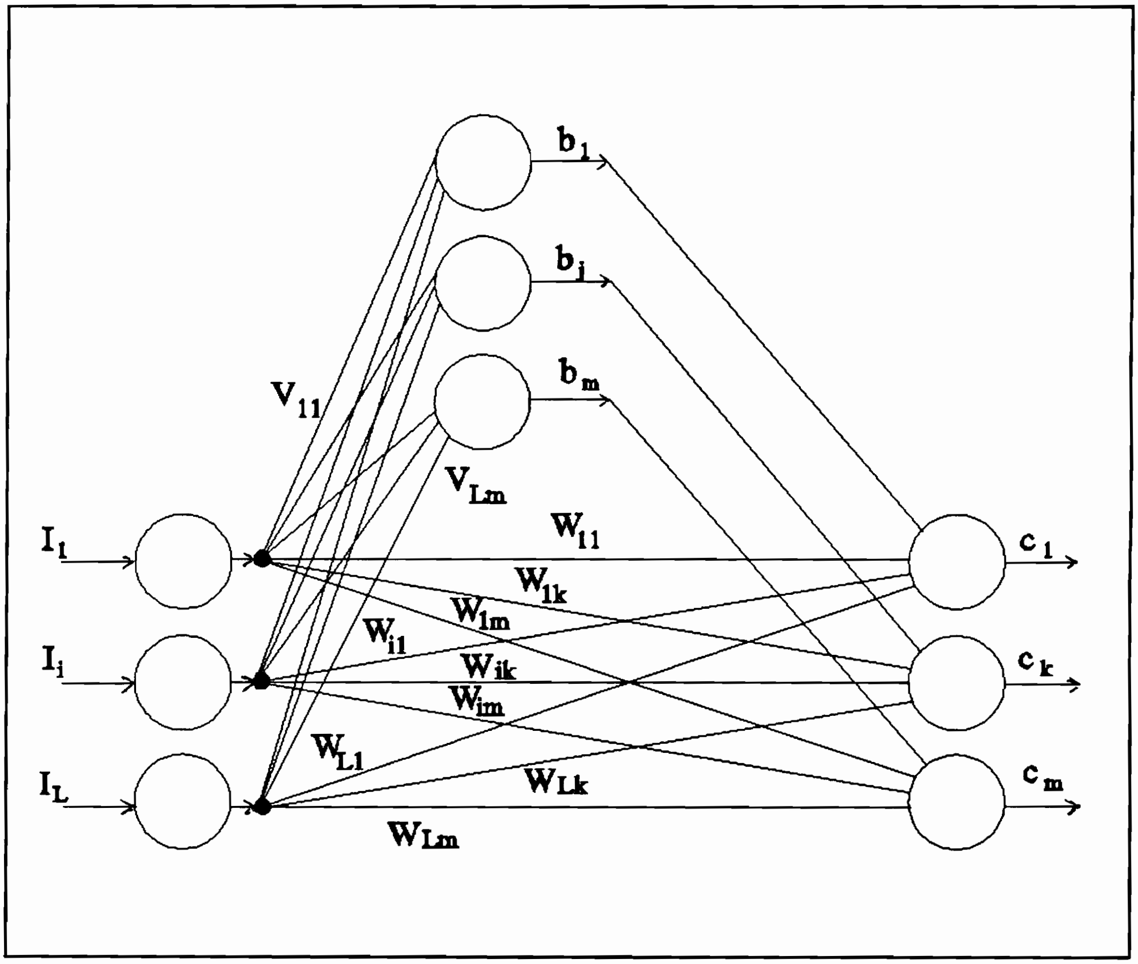


Figure 17.14. Topology of the Adaptive Heuristic Critic.

Instead, in AHC, all components receive the same single error estimation, and corrections in weights follow from there. As seen in Figure 17.14, the output from layer 2 goes directly into layer 3 with no weights. Thus, the PEs in layer 2 act as critics that provide a heuristically derived correction value to each PE in layer 3.

a. Write a computer program that follows the training and recall steps described below.

TRAINING

Step 1: Assign random values between [0, +1] to all v_{ij} and w_{ik} interconnections.

For m = 1, 2, ..., M input patterns, repeat steps 2-10 for each input pattern.

Step 2: Calculate the output from the first layer according to the equation:

$$a_i^m = I_i^m$$

where I_i^m is the input and a_i^m is the output, both for the i-th node on the input layer for training session m. Note that the first layer has no thresholds or sigmoid functions; it simply passes the input directly into the network.

Step 3: Calculate the output from the third layer using the equation:

$$c_k(t) = f\left(\sum_{i=1}^{m} a_i^m w_{ik}(t) + \gamma(t)\right)$$

where $c_k(t)$ is the output from the k-th node on the output layer at time t; $\gamma(t)$ is a random value in the interval [0,1]; and f(), instead of being a sigmoid function, is a bipolar step function, defined as:

$$f(x) = \begin{cases} +1 & if\, x>0 \\ -1 & if\, x\le 0 \end{cases}$$

Step 4: Calculate the *prediction*, $p_j(t)$, for the j-th node on layer two according to the equation:

$$p_j(t) = \sum_{i=1}^{n} v_{ij} a_i^m$$

where v_{ij} is the connection weight between the i-th node in layer one and the j-th node in layer two.

Step 5: Calculate the *internal reinforcement*, $b_j(t)$, according to the equation:

$$b_j(t) = r(t) + \mu p_j(t) - p_j(t-1)$$

where μ is a positive constant ($0 < \mu < 1$) controlling the internal reinforcement, and r(t) is the external reinforcement.

What is *external reinforcement*? The value of r(t) depends on the ANN's performance. If a "fail" situation occurs, where the controlled variable falls outside of a desired range, then $r(t) = -1$. If the controlled variable is within desired range, then $r(t) = 0$.

Step 6: Calculate the *eligibility*, $e_{ik}(t)$, of the change in weight w_{ik} at time t, according to the equation:

$$e_{ik}(t+1) = \delta\, e_{ik}(t) + (1-\delta)\, a_i^m c_k^m$$

where δ is a positive constant ($0 < \delta < 1$) controlling the eligibility's decay rate.

Step 7: Adjust the w_{ik}'s, the connection weights for layers one to three, according to the equation:

$$\Delta w_{ik} = \alpha\, b_k(t)\; e_{ik}(t)$$

where α is a positive constant controlling the learning rate; Δw_{ik} is the change in the weight connecting the i-th node in layer one and the k-th node in layer three; $e_{ik}(t)$ is the *eligibility* of the change w_{ik} at time t; and $b_k(t)$, called the *internal reinforcement*, is the output from the k-th node in layer two at time t.

Step 8: Calculate x_i, the weighted average of the i-th node's value in layer one, by the equation:

$$x_i(t + 1) = \epsilon\, x_i(t) + (1 - \epsilon)\, a_i^m$$

where ϵ is a positive constant controlling the decay rate ($0 < \epsilon < 1$).

Step 9: Adjust the weights v_{ij} according to the equation:

$$\Delta v_{ij} = \beta\, b_j(t)\; x_i(t)$$

Step 10: Increment t and return to step 2; continue to iterate through steps 2-9 until no poor performers exist (i.e., $r(t) = 1$ for a sufficiently long period of time).

RECALL

The AHC provides constantly updated values of the outputs from the network. Thus, we can perform recall easily using the equation:

$$c_k(t) = f\left(\sum_{i=1}^{m} a_i^m w_{ik}(t) + \gamma(t) \right)$$

The continuous updating and easy recall enables on-line training of the

AHC.

We will now use the AHC to control the temperature of water in a vessel. Note that the AHC uses an "on-off" type of control since the output is either +1 (turn the heat on) or -1 (turn the heat off). One benefit of the AHC is that we can start the controller up "naively"; the controller will take in process data on-line, and update its weights to properly control the process. *The system requires no prior tuning*; just plug it in and turn it on.

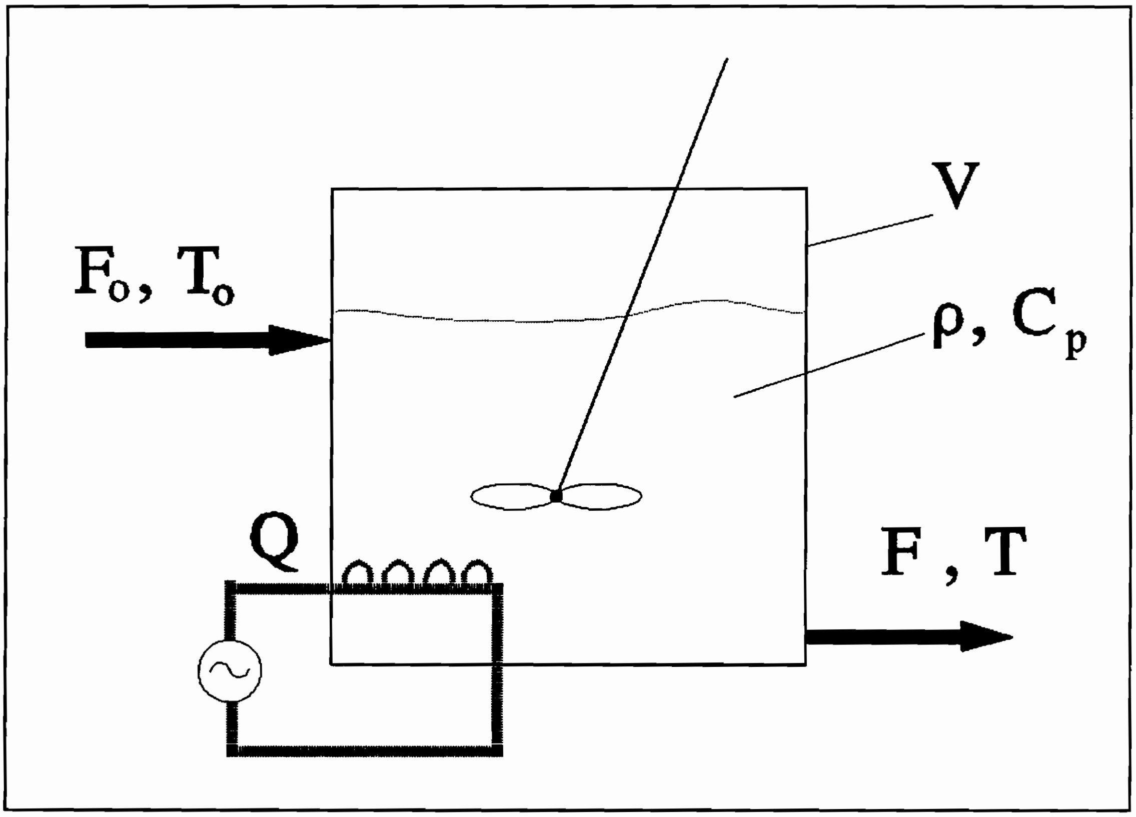


Figure 17.15. On-off control of temperature in a tank.

We use the AHC to control the temperature of the water exiting the tank shown in Figure 17.15. The flow rate into the tank is fixed at 2 m^3/s, and does not change as a function of time. The temperature into the tank *does* change as a function of time, and falls from a maximum value of 35 °C to 15 °C in a pulsating fashion:

$$T(t) = 35 + 15\sin(\frac{\pi t}{1000}) \quad \text{if} \quad \sin(\frac{\pi t}{1000}) < 0$$

$$T(t) = 35 \quad \text{if} \quad \sin(\frac{\pi t}{1000}) \geq 0$$

b. Develop the energy balance for the tank, and determine dT/dt as a function of F, V, T_o, T, Q, ρ, and C_p. Develop the numerical equation for T(t), using Euler's method.

c. Using the equations for T(t) and dT/dt as a model of the process, use the AHC developed in part (a) as a controller. The AHC has two inputs: T and dT/dt. It also has one output, c_1. If $c_1 = 1$, we turn the heat on. If $c_1 = -1$, we turn the heat off. To help, use the following values for the AHC parameters:

$$\alpha = 1.1 \qquad \beta = 0.0005 \quad \delta = 0.9 \qquad \mu = 0.95 \qquad \epsilon = 0.99$$

Recall that γ(t) is a "noise" function that generates random values on the interval [0,1]. Make γ(t) a Gaussian density function with a mean value of 0.5 and $\sigma = 0.125$. Start with a Δt of 0.5 seconds in the simulation, though you may find it valuable to experiment with the time increment. How well does the controller control the process? What do you suggest to improve performance? What does control theory suggest about on-off control for a first-order process?

17.4 COMMENTS AND CONCLUSIONS

While ANNs have many advantages, they are not a panacea. Before one embarks on an ANN project, it is wise to understand both the advantages and disadvantages of ANNs. Unfortunately, when we read published reports on the use of ANNs, authors tend to emphasize the *advantages* while de-emphasizing some of the *limitations* and associated *problems*. For this reason, we recommend getting a good understanding of ANNs through *hands-on* experience with minimal initial monetary investment. Through this hands-on experience, we will be better able to ascertain the appropriateness of ANNs for the specific application.

A. Advantages of Artificial Neural Networks

ANNs are simply another computer-modeling tool that offers some distinct advantages over some more well-known, traditional computer-modeling techniques. These advantages include:

(1) *Adaptive behavior*- ANNs have the ability to adapt, or *learn*, in response to their environment. They learn through training, where we give the ANN input-output patterns, and it adjusts itself to minimize the error.

(2) *Pattern-recognition properties*- ANNs perform multivariable

pattern recognition very well; indeed, this area is where ANNs will probably find the most use in engineering. In chemical engineering, process control and fault diagnosis involve extensive amounts of pattern-recognition. Not surprisingly, these application areas have received the most attention. Other engineering disciplines requiring pattern recognition are also seeing ANN applications. A classic example is in robotics, where ANNs find applications in both control and vision.

(3) *Filtering capacity: low sensitivity to noise and incomplete information*- It would be misleading to say that ANNs are insensitive to noise and incomplete information. However, compared to direct empirical, curve-fitted models, ANNs are definitely lower in sensitivity in a *relative* sense. Why? In empirical models, each independent variable usually plays a critical role. But in ANNs, each node incorporates only a microfeature of the problem; therefore, if the input into a node is incomplete or noisy, that poor input will not manifest itself as severely in an ANN. ANNs can deal with the imperfect world, generalize, and draw substantive conclusions more effectively than the more inflexible empirical models.

(4) *Automated abstraction*- ANNs can ascertain the essentials of relationships, and can do so automatically. We do not need the *domain expert* that knowledge-based systems require. Instead, through training with direct (and sometimes imprecise) numerical data, the ANN can

automatically determine cause-and-effect relations.

(5) *Potential for on-line use-* ANNs may take a very long time to train, but once trained, they can calculate results from a given input very quickly. Since a trained network may take less than a second to calculate results, it has the potential to be used on-line in a control system. Note, however, that at this point in time, the ANN must be trained off-line.

B. Limitations of Artificial Neural Networks

ANNs have a number of limitations that we should be aware of:

(1) *Very long training times-* Training can take long enough to make the ANN impractical. Most simple problems require at least 1000 time steps to train the network, and complex problems can require up to 75,000. Depending on this size of the network, this training could tie up a mainframe computer for 6-24 hours.

(2) *Large amount of training data-* If little input-output data exist on a problem or process, we may reconsider the use of ANNs, since they rely heavily on that data. Consequently, ANNs are best for problems with a large amount of historical data, or those that allow us to train the ANN with a separate simulator.

(3) *No guarantee of optimal results*- Backpropagation is a creative way to "tune" the network, but it does not guarantee that the network will operate properly. The training may "bias" the network, making it accurate in some operating zones, but inaccurate in others. In addition, we may inadvertently get trapped in "local minima" during training.

(4) *No guarantee of 100% reliability*- While this applies to all computational applications, this point is particularly true with ANNs. In fault-diagnosis applications, for example, on some faults, the ANN may misdiagnose only 1% of the time. On other faults *within the same problem*, it may misdiagnose 33% of the time. Importantly, we cannot tell ahead of time (using backpropagation training) what faults will be more prone to misdiagnosis than others. Thus, for a military application that requires near 100% reliability, we must show caution when using ANNs.

In addition to these limitations of ANNs, there can be practical problems that arise when using ANNs (Kramer and Leonard, 1990), particularly when using them for fault diagnosis and classification. These problems are "operational," and are associated with actually implementing the ANN. These problems can "crop up" to such an extent that they can significantly affect network performance. These difficulties are:

(1) *Use of undersized training sets.* For practical reasons, there may be limitations on the amount of data that is available for training the ANN. Examples of classes in relatively low probability density, for

example toward the outer boundaries of the class, may not be present in the training set. Consequently, new cases may fall into regions beyond the range of the training set. With undersized training sets, the networks can also fail to model the relative probability densities of the relations in regions well populated with training points, due to overfitting.

(2) *Shifts in parent distributions of the classes occur subsequent to training*. A process system is never static, and changes affecting the parent distributions of the fault classes may occur after the training set is assembled and the network is trained. These may include alterations in the process due to equipment degradation or maintenance actions, shifts in production levels or quality standards, or changes in exogenous conditions, such as the use of different raw material grades, diurnal and seasonal changes. Such process changes can cause new cases to fall outside of the range of the original training data.

(3) *Faulty sensors corrupt the incoming data*. Miscalibrated or malfunctioning sensors can corrupt the incoming data to the network. When sensors are out of service, substitute values must be used at the input of the network. Use of assumed or inaccurate values may place the input outside of the range of the original training data, forcing the network to extrapolate.

(4) *An example of a novel fault appears*. Because all possible faults

cannot be preenumerated, the training set may exclude certain failure modes. If a novel fault appears, it may fall into a new region of the input space. No classifier can diagnose novel fault types, but the classifier should give an indication that the fault is not one of the known types. Backpropagation network classifiers fail to provide such an indication, and classify the novel fault as one of the known faults, or worse yet, as normal. Consequently, the backpropagation network could miss the movement of the plant into a hazardous regime. This behavior is unsuitable when the cost of the misdiagnosis is high, for example in hazardous processes or economically sensitive applications.

(5) *The network is trained with synthetic data.* The training examples may be from a numerical simulation of the process, rather than the process itself. In this case, extrapolation may be required because of process-model mismatch. A similar situation arises when the training data is taken from a similar, but not identical process.

C. Closing Remarks

Artificial neural networks offer some distinct advantages over other forms of computing, including aspects of AI and traditional modeling. The wise engineer will know both the advantages and limitations before embarking on a long project. While we feel that ANNs will have an impact and may be "here to stay," we caution against excessive optimism. There are a number of practical problems with ANNs that must be overcome before they can be widely used. With respect to engineering, ANNs are still in their infancy, and researchers are only beginning to probe their uses and their limitations in the solutions of practical problems. Cases in the future will undoubtedly arise where ANNs simply have not advanced to the point that they can be applied to a specific engineering problem. Under those circumstances, we as engineers will simply have to wait and rely on the mathematicians to further refine ANNs.

NOTATIONS

a_i a numerical value representing the *activity* of a connection. It is the numerical output from the i-th node in a previous layer, and is used as the input into the following layer. Also used to represent activity of the output from the i-th node in layer 1 in a three-layer ANN.

b_j Output (activity) from the j-th node; in a three-layer ANN, it represents the output from the j-th node in layer 2.

c_k Output (activity) from the k-th node; in a three-layer ANN, it represents the output from the k-th node in layer 3.

d_n Desired output from an ANN.

e_j Calculated error (resulting from backpropagation) of the j-th node in the hidden layer.

E Total squared output error:

$$E = \sum_n \epsilon_n^2 = \sum_n (d_n - b_n)^2$$

f(x) Functional form for the calculation done by the ANN based on its input. Most ANNs use *sigmoid* (S-shaped) type functions.

F Tank exit flow rate, m^3/s

F_o Tank inlet flow rate, m^3/s

I_i Input vector into the ANN.

Q Heating-coil duty, watts.

v_{ij} Connectivity weights between the i-th node in layer 1 and the j-th node in layer 2.

V Tank volume, m^3.

w_{jk} Connectivity weights between the j-th node in layer 2 and the k-th node in layer 3.

t Time, s.

T_o Tank inlet temperature, °C.

T Tank exit temperature, °C.

T_{ij} Internal threshold value for the i-th node in layer j.

T_{ij}^{m} Internal threshold value for the i-th node in layer j for the m-th training session.

x_i Post-threshold input to the node, i.e., the total input to the node less the internal threshold value.

GREEK LETTERS

α Coefficient for the momentum term.

β Learning rate.

β_b Learning rate for layer B (i.e., layer 2, the hidden layer).

β_c Learning rate for layer C (i.e., layer 3, the output layer).

β_j Learning rate for the j-th node.

$\delta_{2j}{}^{m}$ The gradient-descent term for the j-th node in the hidden layer (layer 2) for training pattern m.

$\delta_{3k}{}^{m}$ The gradient-descent term for the k-th node in the output layer (layer 3) for training pattern m.

ϵ_n The error vector, the difference between the desired output from node n and the actual output from node n, i.e., $\epsilon_n = d_n - c_n$.

η The learning rate, used in gradient-descent learning.

REFERENCES

The number of publications on ANNs is growing rapidly, and it is getting difficult to keep up. First, some textbooks that are helpful are: 1) *Neural Computing: Theory and Practice* (Wasserman, 1989) - a good introductory and intermediate-level book on ANNs, and 2) *Artificial Neural Systems* (Simpson, 1990) - a good reference book, mathematically oriented, with numerous technical references. Unfortunately, at the time of this writing, virtually all of the AI textbooks do not discuss the growing importance of ANNs. An exception is Rich and Knight (1991). Chapter 18 of this book briefly discusses the concepts of ANNs in the context of "connectionist models."

In addition, a number of magazines and journals have had special issues dedicated to ANNs. Some of these editions are: 1) *IEEE Control Systems Magazine*, April 1988, 1989, and 1990, 2) *PC-AI* May/June 1990, 3)

IEEE Communications Magazine, November 1989. Finally, Computers and Chemical Engineering plans to publish a special issue on ANNs in 1991.

Anderson, C.W., "Learning to Control an Inverted Pendulum Using Neural Networks," *IEEE Control Systems Magazine*, 31, April (1989).

Barto, A., R. Sutton, and C. Anderson, "Neuron-Like Adaptive Elements That Can Solve Difficult Learning Control Problems," *IEEE Transactions on Systems, Man, and Cybernetics*, *SMC-13*, 834 (1983).

Bavarian, B. "Introduction to Neural Networks for Intelligent Control," *IEEE Control Systems Magazine*, 3, April (1988).

Bhagat, P., "An Introduction to Neural Nets," *Chemical Engineering Progress*, 55, August (1990).

Bhat, N. and T.J. McAvoy, "Use of Neural Nets for Dynamic Modeling and Control of Chemical Process Systems," *Comput. Chem. Eng.*, 14, 573 (1990).

Bhat, N., P.A. Minderman, Jr., T.J. McAvoy, and N.S. Wang, "Modeling Chemical Process Systems via Neural Computation," *IEEE Control Systems Magazine*, 24, April (1990).

Chen, F.C., "Backpropagation Neural Networks for NonLinear Self-Tuning Adaptive Control," *IEEE Control Systems Magazine*, 44, April (1990).

Cooper, D.J., R.F. Hinde, and L. Megan, "A Performance Feedback Neural Network for Pattern-Based Adaptive Process Control," *AIChE Annual Meeting*, Chicago IL, November (1990).

Ferrada, J.J., P.A. Grizzaffi, and I.W. Osborne-Lee, "Applications of Neural Networks in Chemical Engineering - Hybrid Systems," *AIChE Annual Meeting*, Chicago IL, November (1990).

Foster W., F. Collopy, and L. Ungar, "Neural Network Forecasting of Short, Noisy Time Series," *AIChE Annual Meeting*, Chicago IL, November (1990).

Guez, A., and J.L. Eilbert, M. Kam, "Neural Network Architecture for Control," *IEEE Control Systems Magazine*, 22, April (1988).

Hebb, D., *Organization of Behavior*, John Wiley, New York, NY (1949).

Hendler, J., "Editorial: On The Need for Hybrid Systems," *Connection Science*, 1, 227 (1989).

Hendler, J., "Marker-passing over Microfeatures: Towards a Hybrid

Symbolic/Connectionist Model," *Cognitive Science*, 13, 79 (1989).

Hoskins, J.C. and D.M. Himmelblau, "Artificial Neural Network Models of Knowledge Representation in Chemical Engineering," *Comput. Chem. Eng.*, 12, 881 (1988).

Hoskins, J.C. and D.M. Himmelblau, "Fault Detection and Diagnosis Using Artificial Neural Networks," pp. 123-160 in *Artificial Intelligence in Process Engineering*, M.L. Mavrovouniotis (Ed)., Academic Press, San Diego, CA, (1990a).

Hoskins, J.C. and D.M. Himmelblau, "Process Control Via Neural Networks and Reinforcement Learning," *AIChE Annual Meeting*, Chicago IL, November (1990b).

Hoskins, J.C. and K.M. Kaliyur, and D.M. Himmelblau, "Fault Diagnosis in Complex Chemical Plants Using Artificial Neural Networks," *AIChE Annual Meeting*, Chicago IL, November (1990c).

Hoskins, J.C. and K.M. Kaliyur, and D.M. Himmelblau, "Fault Diagnosis in Complex Chemical Plants Using Artificial Neural Networks," *AIChE J.*, **37**, 137, (1990d).

Hudson, J.L., M. Kube, R.A. Adomaitis, I.G. Kevrekidis, A.S. Lapedes, and

R.M. Farber, "Nonlinear Signal Processing and System Identification Application to Time Series from Electrochemical Reactions," *Chem. Eng. Sci.*, **45**, 2075 (1990).

Kramer, M.A. and Leonard, J.A., "Diagnosis Using Backpropagation Neural Networks - Analysis and Criticism," *Comput. Chem. Eng.*, **14**, 1323, (1990).

Kuperstein, M. and J. Rubinstein, "Implementation of an Adaptive Neural Controller for Sensory-Motor Coordination," *IEEE Control Systems Magazine*, 25, April (1989).

Leonard, J.A. and M.A. Kramer, "Limitations of the Backpropagation Approach to Fault Diagnosis and Improvement with the Radial Bias Function," *AIChE Annual Meeting*, Chicago IL, November (1990).

Liu, H. T. Iberall, and G.A. Bekey, "Neural Network Architecture for Robot Hand Control," *IEEE Control Systems Magazine*, 38, April (1989).

Mah, R.S.H., "Neural Nets," Letter to the Editor, *Chem. Eng. Prog.*, **87**, 6, January (1991).

Mavrovouniotis, M.L., "Hierarchical Neural Networks," *Comput. Chem Eng.*, special issue on Neural Networks, in press (1991).

Morari, M. and E. Zafiriou, *Robust Process Control*, Prentice-Hall, Englewood Cliffs, NJ (1989).

Naidu, S.R., E. Zafiriou, and T.J. McAvoy, "Use of Neural Networks for Sensor Failure Detection in a Control System," *IEEE Control Systems Magazine*, 49, April (1990).

Nett, C.N., C.A. Jacobson, A.T. Miller, "An Integrated Approach to Controls and Diagnostics: The 4-Parameter Controller," *Proc. of the American Control Conference*, 824, Atlanta, GA (1988).

Nguyen, D.H., and B. Windrow, "Neural Networks for Self-Learning Control Systems," *IEEE Control Systems Magazine*, 18, April (1990).

Passino, K.M., M.A. Sartori, and P.J. Antsaklis, "Neural Computing for Numeric-to-Symbolic Conversion in Control Systems," *IEEE Control Systems Magazine*, 44, April (1989).

Pollard, J.F., and D.B. Garrison, "Process Identification Using Neural Networks," *AIChE Annual Meeting*, Chicago IL, November (1990b).

Psaltis, D., A. Sideris, and A.A. Yamamura, "A Multilayered Neural Network Controller," *IEEE Control Systems Magazine*, 17, April (1988).

Psichogios, D.C. and L.H. Ungar, "Direct and Indirect Model-Based Control Using Artificial Neural Networks," *AIChE Annual Meeting*, Chicago IL, November (1990).

Rangwala, S.S., and D.A. Dornfield, "Learning and Optimization of Machining Operations Using Computing Abilities of Neural Networks," *IEEE Transactions on Systems, Man, and Cybernetics*, **19**, 299, March/April (1989).

Rich, E. and K. Knight, *Artificial Intelligence*, Chapter 18, pp. 487-525, McGraw-Hill, New York, NY (1991).

Rivera, D.E., M. Morari, and S. Skogestad, "Internal Model Control, 4. PID Controller Design," *Ind. Eng. Process Design Dev.*, **25**, 252 (1986).

Roat, S., A. Farell, and C. Moore, "Applications of Neural Networks and System Cultivation to a Nonlinear Optimal Control Algorithm," *AIChE Annual Meeting*, Chicago IL, November (1990).

Roth, M.W., "Survey of Neural Network Technology for Automatic Target Recognition," *IEEE Transactions on Neural Networks*, **1**, 28, March (1990).

Rumelhart, D., G. Hinton, and R. Williams, "Learning Representations by

Backpropagating Errors," *Nature*, **323**, 533 (1986).

Seborg, D.E., T.F. Edgar, and D.A. Mellichamp, *Process Dynamics and Control*, 272, John Wiley & Sons, New York, NY (1989).

Samdani, G., "Neural Nets," *Chemical Engineering*, **97**, 37, August (1990).

Sandon, P. and L. Uhr, "A Local Interaction Heuristic for Adaptive Networks," *Proc. of the IEEE International Conference on Neural Networks*, Volume I, p.317, San Diego, CA (1988).

Simpson, P.K., *Artificial Neural Systems*, Pergamon Press, New York, NY, (1990).

Stornetta W. and B. Huberman, "An Improved Three-Layer Backpropagation Algorithm," *Proc. of the First IEEE International Conference on Neural Networks*, Volume II, p.549, San Diego, CA (1987).

Vaidynathan, R. and V. Venkatasubramanian, "Process Fault Detection and Diagnosis Using Neural Networks - II. Dynamic Processes," *AIChE Annual Meeting*, Chicago IL, November (1990).

Venkatasubramanian, V., and K. Chan, "A Neural Network Methodology for Process Fault Diagnosis," *AIChE J.*, **35**, 1993 (1989).

Venkatasubramanian, V., R. Vaidynathan, and Y. Yamamoto, "Process Fault Detection and Diagnosis Using Neural Networks - I. Steady-State Processes," *Comput. Chem. Eng.*, **14**, 699 (1990).

Vogl, T. J. Mangis, A. Rigler, W. Zink, and D. Alkon, "Accelerating the Convergence of the Backpropagation Method," *Biological Cybernetics*, **59**, 257 (1988).

Wasserman, P.D., "Combined Backpropagation/Cauchy Machine," *Proc. of the International Neural Network Society*, Pergamon Press, New York, NY (1988).

Wasserman, P.D., *Neural Computing: Theory and Practice*, Van Nostrand Reinhold, New York, NY (1989).

Watanabe, K., I. Matsuura, M. Abe, M. Kubota, and D.M. Himmelblau, "Incipient Fault Diagnosis of Chemical Processes via Artificial Neural Networks," *AIChE J.*, **35**, 1803 (1989).

Weiland, A. and R. Leighton, "Shaping Schedules as a Method for Accelerated Learning," *Neural Networks Supplement: INNS Abstracts*, **1**, 231 (1988).

Whitely, J.R., and J.F. Davis, "Back-Propagation Neural Networks for

Qualitative Interpretation of Process Sensor Data," *AIChE Annual Meeting*, Chicago IL, November (1990).

Yao, S.C., and E. Zafiriou, "Control System Sensor Failure Detection via Networks of Localized Receptive Fields," *American Control Conf.*, San Diego, CA, June (1990).

Ydstie, B.E., "Forecasting and Control Using Adaptive Connectionist Networks," *Comput. Chem. Eng.*, **14**, 583 (1990).

Yu, W., and H. Teh, "An Error-Tolerant Environment of Multilayer Perceptrons with Controlled Learning," *Neural Networks Supplement: INNS Abstracts*, 1, 323 (1988).

APPENDIX A

AMERICAN STANDARD CODE FOR INFORMATION INTERCHANGE

(ASCII CODE)

APPENDIX A: ASCII CONVERSION CHART

Number	Character	Number	Character	Number	Character
001	☺	026	→	051	3
002	☻	027	←	052	4
003	♥	028	∟	053	5
004	♦	029	↔	054	6
005	♣	030	▲	055	7
006	♠	031	▼	056	8
007	•	032	(space)	057	9
008	◘	033	!	058	:
009	○	034	"	059	;
010	◙	035	#	060	<
011	♂	036	$	061	=
012	♀	037	%	062	>
013	♪	038	&	063	?
014	♫	039	'	064	@
015	☼	040	(	065	A
016	►	041	)	066	B
017	◄	042	*	067	C
018	↕	043	+	068	D
019	‼	044	,	069	E
020	¶	045	-	070	F
021	§	046	.	071	G
022	▬	047	/	072	H
023	↨	048	0	073	I
024	↑	049	1	074	J
025	↓	050	2	075	K

APPENDIX A: ASCII CONVERSION CHART

Number	Character	Number	Character	Number	Character
076	L	101	e	126	~
077	M	102	f	127	⌂
078	N	103	g	128	Ç
079	O	104	h	129	ü
080	P	105	i	130	é
081	Q	106	j	131	â
082	R	107	k	132	ä
083	S	108	l	133	à
084	T	109	m	134	å
085	U	110	n	135	ç
086	V	111	o	136	ê
087	W	112	p	137	ë
088	X	113	q	138	è
089	Y	114	r	139	ï
090	Z	115	s	140	î
091	[	116	t	141	ì
092	\	117	u	142	Ä
093	]	118	v	143	Å
094	^	119	w	144	É
095	_	120	x	145	æ
096	'	121	y	146	Æ
097	a	122	z	147	ô
098	b	123	{	148	ö
099	c	124	|	149	ò
100	d	125	}	150	û

APPENDIX A: ASCII CONVERSION CHART

Number	Character	Number	Character	Number	Character
151	ù	176	░	201	╔
152	ÿ	177	▒	202	╩
153	Ö	178	▓	203	╦
154	Ü	179	│	204	╠
155	¢	180	┤	205	═
156	£	181	╡	206	╬
157	¥	182	╢	207	╧
158	₧	183	╖	208	╨
159	ƒ	184	╕	209	╤
160	á	185	╣	210	╥
161	í	186	║	211	╙
162	ó	187	╗	212	╘
163	ú	188	╝	213	╒
164	ñ	189	╜	214	╓
165	Ñ	190	╛	215	╫
166	ª	191	┐	216	╪
167	º	192	└	217	┘
168	¿	193	┴	218	┌
169	⌐	194	─	219	█
170	¬	195	├	220	▄
171	½	196	─	221	▌
172	¼	197	┼	222	▐
173	¡	198	╞	223	▀
174	«	199	╟	224	α
175	»	200	╚	225	ß

APPENDIX A: ASCII CONVERSION CHART

Number	Character	Number	Character	Number	Character
226	Γ	236	∞	246	÷
227	π	237	φ	247	≈
228	Σ	238	ε	248	°
229	σ	239	∩	249	•
230	µ	240	≡	250	·
231	τ	241	±	251	√
232	Φ	242	≥	252	ⁿ
233	Θ	243	≤	253	²
234	Ω	244	⌠	254	■
235	δ	245	⌡	255	

APPENDIX B

EXSEP PROGRAM LISTING

WITH COMMENTS

MAIN MODULE

```
project "exsep"                         % required for Turbo Prolog only

global domains                          % required for Turbo Prolog only
        alpha = symbol
        alist = alpha*
        number = integer
        value = real
        vlist = value*
        str = string
/*****************************************************
 *    These global domains are for Toolbox predicates *
 *****************************************************/
  ROW, COL, LEN, ATTR   = INTEGER
  STRINGLIST = STRING*
  INTEGERLIST = INTEGER*
  KEY    = cr; esc; break; tab; btab; del; bdel; ctrlbdel; ins;
           end ; home ; fkey(INTEGER) ; up ; down ; left ; right ;
           ctrlleft; ctrlright; ctrlend; ctrlhome; pgup; pgdn;
           ctrlpgup; ctrlpgdn; char(CHAR) ; otherspec

global database - bypass                % required for Turbo Prolog only
        dbypass_amount(alist,alpha,value)
        dbypass_result(alist,alpha)
        dbypass_status(alpha)

global database - materials             % required for Turbo Prolog only
        flow(alpha,alpha,value)
        k_value(alpha,value)
        boiling_temp(alpha,value)
        initial_components(alist)
        initial_set(alist)
        corrosive(alpha)
        pseudoproduct(alist,alpha)

global database - table                 % required for Turbo Prolog only
        dsst(alist,alist,alpha,alist,alist,alpha,alpha,value,vlist,vlist,alpha,alpha,value)
        ddsst(alist,alist,alpha,alist,alist,alpha,alpha,value,vlist,vlist,alpha,alpha,value)
        dddsst(alist,alist,alpha,alist,alist,alpha,alpha,value,vlist,vlist,alpha,alpha,value)
        ddddsst(alist,alist,alpha,alist,alist,alpha,alpha,value,vlist,vlist,alpha,alpha,value)
        dddddsst(alist,alist,alpha,alist,alist,alpha,alpha,value,vlist,vlist,alpha,alpha,value)
        process_keys(alpha,alpha)
        process_product(alpha)
        sloppy_keys(alpha,alpha)

global database - sequence              % required for Turbo Prolog only
        dseparator(number,alist,vlist,alist,vlist)
        dstreamin(number,alist,vlist)
        dstreambypass(number,alpha,alist,vlist)

global predicates                       % required for Turbo Prolog only
        append_v(vlist,vlist,vlist) - (i,i,o)
        bypass(alist,alist) - (i,i)
        component_flow(alist,alpha,value) - (i,i,o)
        component_flow_list(alist,vlist,alist,alist) - (i,i,i,o)
        component_flow_set(alist,alist,vlist) - (i,i,o)
        decide_difficult_or_easy(value,alpha) - (i,o)
        delete_element(alpha,alist,alist) - (i,i,o)
        delete_elements(alist,alist,alist) - (i,i,o)
        delete_number(value,vlist,vlist) - (i,i,o)
        equal(alist,alist) - (i,i)
        ffactor(value,value,value) - (i,i,o)
        first_prod(alist,alist) - (i,o)
        flow_set(alpha,alist,vlist) - (i,i,o)
        flow_set_c(alpha,alist,vlist) - (i,i,o)
        get_split_flow(alist,value,alist,value) - (i,o,i,i)
        largest_flow(alist,alpha,alist) - (i,o,i)
        last_prod(alist,alist) - (i,o)
        length(alist,value) - (i,o)
        list_member(alist,alist) - (i,i)
        max(value,value,value) - (i,i,o)
        max(number,number,number) - (i,i,o)
        max_flist(vlist,value) - (i,o)
        max_flow(value,value,value) - (i,i,o)
        member(alpha,alist) - (i,i)
        pause(number) - (i)
        pause_escape
        positive_component_flow_list(alist,alist) - (i,o)
        product_flow(alpha,value,alist) - (i,o,i)
        product_flow_help(alpha,value,alist,value) - (i,o,i,i)
        product_flow_match(alist,value,alpha,alist) - (i,i,o,i)
        product_flow_set(alist,vlist,alist) - (i,o,i)
        reverse(alist,alist) - (i,o)
        selected_positive_component_flow_list(alist,alist,alist) - (i,i,o)
        separation_specification_table(alist,alist) - (i,i)
        set_above(alist,alpha,alist) - (i,i,o)
        set_below(alist,alpha,alist) - (i,i,o)
        split(alist,alist,alist,alist,alist,alist) - (i,i,o,o,o,o)
        split_above(alist,alpha,alist,alist) - (i,i,o,o)
        split_below(alist,alpha,alist,alist) - (i,i,o,o)
        sub_list(alist,alist) - (i,i)
        sum_component_flow(alist,value,alist) - (i,o,i)
        sum_flist(vlist,value) - (i,o)
        two_highest_flows(vlist,value,value) - (i,o,i)

database                                % required for Turbo Prolog only
        insmode
        lineinpstate(string,integer)
        lineinpflag

include "tpreds.pro"                    % required for Input/Output help
include "menu.pro"                      % required for Input/Output help
include "lineinp.pro"                   % required for Input/Output help
include "filename.pro"                  % required for Input/Output help

predicates                              % required for Turbo Prolog only
        append(alist,alist,alist)
        add_pseudo(number,alpha,alist,alist)
        build_pseudo(alpha,alist,alist)
        decide_cursor(number,number,number)
        decide_to_reprint_cam(alist,alist)
        develop_sequence(alist,alist)
        generate(alist,alist)
        initialize(alist,alist)
        plan(alist,alist)
        print_bottoms(alist,vlist,number)
        print_component_and_bypassflow(alist,vlist,number)
        print_component_and_streamflow(alist,vlist)
        print_component_line(alist)
        print_cam(number,alist,alist)
        print_cam_line(alist,alist)
        print_cam_driver(alist,alist,alpha)
        print_overhead(alist,vlist,number)
        print_row_flows(alpha,alist)
        readcomponents(number,alist,alist,number)
        readcorrosive(number,alist)
        readdata(number,number)
        readflows(alist,alist)
        read_flows_for_product(alpha,alist)
        read_k_values_and_boiling_points(alist)
        readproducts(number,alist,alist,number)
        readpseudo(number,alist)
        readpseudo_products(alist,number,number)
        report
        resolve_choice(number)
        resolve_corrosive(alpha,number)
        retrieve_and_print_bypass(number,number)
        retrieve_and_print_streamflow(number,number)
        reverse_dseparator_database(number,number)
        run
        save_to_file(number)
        sequence_print(number)
        test(alist,alist,alist,alist,alist,alist)

goal
        run.                                % Required for Turbo Prolog.
                                            % Makes program "self-executing"

/*****************************************************************************************************
        All computer code above this point is unique to Turbo Prolog and is required to get
        it to run. Below this point, we begin the actual Prolog relations in the MAIN
        MODULE. The relations below will run on almost any Prolog system.
*****************************************************************************************************/

clauses

        run:-
                repeat,
                initialize(List,CList),
                develop_sequence(List,CList),
                report,
                fail.
```

```
initialize(List,Clist):-
        makewindow(1,30,30,"Knowledge-Based Separation Sequencing",1,1,24,79),
        makewindow(2,113,113,"Component Assignment Matrix",2,1,23,79),
        makewindow(3,113,113,"Bypass Analysis",13,1,12,79),
        makewindow(4,29,29,"Separation Specification Table",2,1,23,79),
        makewindow(5,81,81,"Heuristic Analysis",2,1,23,79),
        makewindow(6,14,14,"Data input from keyboard",2,1,23,60),
        makewindow(7,14,14,"",21,45,4,35),
        shiftwindow(1),
        clearwindow,
        write(" \n\n"),
        write("         EEEEEEEEE  XXX       XXX   SSSSSSSS   EEEEEEEEE    PPPPPPPPP \n"),
        write("         EEEEEEEEE   XXX     XXX   SSSSSSSSS   EEEEEEEEE  PPPPPPPPPPP\n"),
        write("         EEE          XXX   XXX    SSS         EEE        PPP     PPP\n"),
        write("         EEE           XXX XXX     SSS         EEE        PPP     PPP\n"),
        write("         EEE            XXXXX      SSS         EEE        PPP     PPP\n"),
        write("         EEEEEE          XXX       SSSSSSSSS   EEEEEE     PPPPPPPPPP \n"),
        write("         EEE            XXXXX            SSS   EEE        PPP        \n"),
        write("         EEE           XXX XXX           SSS   EEE        PPP        \n"),
        write("         EEE          XXX   XXX          SSS   EEE        PPP        \n"),
        write("         EEEEEEEEE   XXX     XXX   SSSSSSSSS   EEEEEEEEE  PPP        \n"),
        write("         EEEEEEEEE  XXX       XXX  SSSSSSSS    EEEEEEEEE  PPP        \n"),
        cursor(21,14),
        write("Please enter desired choice for data loading."),
        menu(19,25,14,14,["File","Screen"],"Choose Method",2,Choice),
        resolve_choice(Choice),
        initial_set(List),
        initial_components(CList),
        asserta(dseparator(0,[],[],[],[]),sequence),!.

develop_sequence([_],_):-!.
develop_sequence(_,[_]):-!.
develop_sequence(List,CList):-
        plan(List,CList),
        generate(List,CList),
        test(List,CList,TopList,TopCList,BotList,BotCList),
        develop_sequence(TopList,TopCLIst),
        develop_sequence(BotList,BotCList).

plan(List,CList):-
        clearwindow,
                print_cam_driver(List,CList,"fresh"),
                bypass(List,Clist),
                decide_to_reprint_cam(List,CList),!.

        generate(List,CList):-
                separation_specification_table(List,CList).

        test(List,CList,TopList,TopCList,BotList,BotCList):-
                shiftwindow(5),
                clearwindow,
                split(List,CList,TopList,TopCList,BotList,BotCList),
                shiftwindow(1),
                clearwindow,!.

        report:-
                clearwindow,
                write("\nSeparator  Bypass Around  Stream Flow In      Overhead        Bottoms"),
                write("\n---------  -------------  ---------------  --------------
-------------\n"),
                retract(dseparator(0,_,_,_),sequence),
                reverse_dseparator_database(1,LastSepNum),
                sequence_print(LastSepNum),
                retractall(_,sequence),
                retractall(_,materials),
                shiftwindow(N),
                pause(N),
                !.

/*              Initializing Utilities                  */

        resolve_choice(Choice):-
                Choice = 1,
                readfilename(15,1,14,14,dat,"",Newfilename),
                consult(Newfilename,materials).

        resolve_choice(Choice):-
                Choice = 2,
                clearwindow,
                shiftwindow(6),
                write("\nHow many products are in the system?\n"),
                readint(NumProd),nl,
        write("\nHow many components are in the system?\n"),
        readint(NumComp),
        readdata(NumComp,NumProd),
        write("Do you wish to save this information to a file?\n"),
        menu(13,23,14,14,["Yes","No"],"Answer",2,Choicesave),
        save_to_file(Choicesave),
        removewindow.

readdata(NumComp,NumProd):-
        clearwindow,
        readcomponents(NumComp,CList,[],1),
        clearwindow,
        readproducts(NumProd,List,[],1),
        readflows(List,CList),
        clearwindow,
        read_k_values_and_boiling_points(CList),
        clearwindow,
        write("\n\n\n\n\nAre any components corrosive or hazardous?\n"),
        menu(13,23,14,14,["Yes","No"],"Answer",2,Choicecor),
        readcorrosive(Choicecor,CList),
        clearwindow,
        write("\n\n\nAre there any pseudoproducts?\n"),
        menu(13,23,14,14,["Yes","No"],"Answer",2,Choicepseudo),
        readpseudo(Choicepseudo,List).

readcomponents(NumComp,CList,RefList,Num):-
        Num <= NumComp,
        write("\nWhat is the name of component ",Num," ?\n"),
        readln(Comp),
        append(RefList,[Comp],NewRefList),
        NNum = Num + 1,
        readcomponents(NumComp,CList,NewRefList,NNum).
readcomponents(NumComp,CList,CList,Num):-
        Num > NumComp,
        assertz(initial_components(CList),materials).

readproducts(NumProd,List,RefList,Num):-
        Num <= NumProd,
        write("\nWhat is the name of product ",Num," ?\n"),
        readln(Prod),
        append(RefList,[Prod],NewRefList),
                NNum = Num + 1,
                readproducts(NumProd,List,NewRefList,NNum).
        readproducts(NumProd,List,List,Num):-
                Num > NumProd,
                assertz(initial_set(List),materials).

        readflows([],_).
        readflows([Head|Tail],CList):-
                clearwindow,
                read_flows_for_product(Head,CList),
                readflows(Tail,CList).

        read_flows_for_product(_,[]).
        read_flows_for_product(Prod,[Head|Tail]):-
                write("\nWhat is the flow of ",Head," in ",
Prod," ?\n"),
                readreal(X),
                assertz(flow(Prod,Head,X),materials),
                read_flows_for_product(Prod,Tail).

        read_k_values_and_boiling_points([]).
        read_k_values_and_boiling_points([Head|Tail]):-
                write("\nWhat is the normal boiling point (C) of ",Head," ?\n"),
                readreal(X),
                assertz(boiling_temp(Head,X),materials),
                write("\nWhat is the k-value of ",Head," ?\n"),
                readreal(Y),
                assertz(k_value(Head,Y),materials),
                read_k_values_and_boiling_points(Tail).

        readcorrosive(2,_).
        readcorrosive(_,[]).
        readcorrosive(1,[Head|Tail]):-
                write("\nIs ",Head," considered corrosive or hazardous?\n"),
                menu(13,23,14,14,["Yes","No"],"Answer",2,Choicecor),
                resolve_corrosive(Head,Choicecor),
                readcorrosive(2,Tail).

        resolve_corrosive(_,2).
        resolve_corrosive(Comp,1):-
                assertz(corrosive(Comp),materials).
        readpseudo(2,_).
        readpseudo(1,List):-
                write("\nHow many pseudoproducts are there?\n"),
                readint(NumPseudo),
                readpseudo_products(List,NumPseudo,1).

        readpseudo_products(_,NumPseudo,N):-
                N > NumPseudo.
        readpseudo_products(List,NumPseudo,N):-
                write("\nWhat is the name of the actual product?\n"),
                readln(Prod),
                build_pseudo(Prod,List,[]),
                NNumPseudo = N + 1,
                readpseudo_products(List,NumPseudo,NNumPseudo).

        build_pseudo(Prod,[],PseudoList):-
                assertz(pseudoproduct(PseudoList,Prod),materials).
        build_pseudo(Prod,[Head|Tail],PseudoList):-
                write("\nIs ",Head," a pseudoproduct of ",Prod," ?\n"),
                menu(13,23,14,14,["Yes","No"],"Answer",2,Choicepseudo),
                add_pseudo(Choicepseudo,Head,PseudoList,NPseudoList),
                build_pseudo(Prod,Tail,NPseudoList).

        add_pseudo(2,_,N,N).
        add_pseudo(1,Head,PseudoList,NPseudoList):-
                append(PseudoList,[Head],NPseudoList).

        save_to_file(2).
        save_to_file(1):-
                write("\nPlease enter file name\n"),
                readln(Ans),
                save(Ans,materials).

        append([],List,List).
        append([Head|Tail],List1,[Head|List]):-
                append(Tail,List1,List).

/*          End of Initializing Utilities                  */

        decide_to_reprint_cam(_,_):-
                dbypass_status("nobypasspossible").
        decide_to_reprint_cam(List,CList):-
                dbypass_status("bypasspossible"),
                print_cam_driver(List,CList,"bypass").

        print_cam_driver(List,CList,"bypass"):-
                shiftwindow(2),
                clearwindow,
                write("Component Assignment Matrix has been recalculated\ndue to bypass. You can
write it to the screen, printer, or not at all. Enter"),
                write("\nyour choice below:"),
                menu(16,25,14,14,["Screen","Printer","Do Not Display"],"Choose
Method",3,Choicecam),
                clearwindow,
                print_cam(Choicecam,List,CList),
                shiftwindow(1),!.

        print_cam_driver(List,CList,_):-
                shiftwindow(2),
                clearwindow,
                write("Component Assignment Matrix is complete. You can"),
                write("\nwrite it to the screen, printer, or not at all. Enter"),
                write("\nyour choice below:"),
                menu(16,25,14,14,["Screen","Printer","Do Not Display"],"Choose
Method",3,Choicecam),
                clearwindow,
                print_cam(Choicecam,List,CList),
                shiftwindow(1),!.

        print_cam(3,_,_).
        print_cam(2,List,CList):-
                writedevice(printer),
                write("     COMPONENT ASSIGNMENT MATRIX\n"),
                write("Products  Total Flow          Components\n\n"),
                reverse(List,Rlist),
                write("--------  ----------          ------------\n"),
                write("                         "),
                print_component_line(CList),
                print_cam_line(Rlist,CList),
                writedevice(screen),
                shiftwindow(N),
                pause(N).
```

```
print_cam(1,List,CList):-
        write("Products  Total Flow     Components"),nl,
        reverse(List,Rlist),
        write("--------  ----------    ------------"),nl,
        write("                              "),
        print_component_line(CList),
        print_cam_line(Rlist,CList),
        shiftwindow(N),
        pause(N).

print_component_line([]):- nl.
print_component_line([Head|Tail]):-
        write(Head,"      "),
        print_component_line(Tail).

print_cam_line(_,[]).
print_cam_line([],_).
print_cam_line([Head|Tail],CList):-
        product_flow(Head,Flow,CList),
        writef(" %4       %5.1f  ",Head,Flow),
        print_row_flows(Head,CList),nl,
        print_cam_line(Tail,CList).

print_row_flows(_,[]).
print_row_flows(Prod,[Head|Tail]):-
        flow(Prod,Head,Flow),
        writef("    %4.1f",Flow),
        print_row_flows(Prod,Tail).

reverse_dseparator_database(Num,LastSepNum):-
        dseparator(Num,TopCList,TopFList,BotCList,BotFList),
        retract(dseparator(Num,TopCList,TopFList,BotCList,BotFList),sequence),
        assertz(dseparator(Num,TopCList,TopFList,BotCList,BotFList),sequence),
        NNum = Num + 1,
        reverse_dseparator_database(NNum,LastSepNum),!.
reverse_dseparator_database(Num,LastSepNum):-
        LastSepNum = Num - 1,!.

sequence_print(LastSepNum):-
        dseparator(Num,TopCList,TopFList,BotCList,BotFList),
        cursor(Row,_),
        cursor(Row,4),
        write("S",Num),
        cursor(Row,12),
        retrieve_and_print_bypass(Num,BypassRow),
        cursor(Row,27),
        retrieve_and_print_streamflow(Num,StreamRow),
        cursor(Row,43),
        print_overhead(TopCList,TopFList,OvhdRow),
        cursor(Row,59),
        print_bottoms(BotCList,BotFList,BottomsRow),
        retract(dseparator(Num,TopCList,TopFList,BotCList,BotFList),sequence),
        max(BypassRow,StreamRow,X),
        max(OvhdRow,BottomsRow,Y),
        max(X,Y,Z),
        decide_cursor(Z,Num,LastSepNum),
        fail.
sequence_print(_).

retrieve_and_print_bypass(Num,BypassRow):-
        dstreambypass(Num,Prod,CList,FList),
        cursor(_,Column),
        Prod <> "none",
        write(Prod,": "),
        print_component_and_bypassflow(CList,FList,LastRow),
        retract(dstreambypass(Num,Prod,CList,FList),sequence),
        NRow = LastRow + 1,
        cursor(NRow,Column),
        retrieve_and_print_bypass(Num,BypassRow),!.
retrieve_and_print_bypass(Num,BypassRow):-
        dstreambypass(Num,"none",_,_),
        write("  none"),
        cursor(Row,_),
        BypassRow = Row + 1,!.
retrieve_and_print_bypass(_,BypassRow):-
        cursor(Row,_),
        BypassRow = Row,!.

retrieve_and_print_streamflow(Num,StreamRow):-
        dstreamin(Num,CList,FList),
        print_component_and_streamflow(CList,FList),
        cursor(Row,_),
        StreamRow = Row + 1,!.

print_component_and_bypassflow([],[],LastRow):-
        cursor(LastRow,_).
print_component_and_bypassflow([HeadC|TailC],[HeadF|TailF],LastRow):-
        cursor(R,C),
        writef("%4.1 %-6",HeadF,HeadC),
        NR = R + 1,
        cursor(NR,C),
        print_component_and_bypassflow(TailC,TailF,LastRow),!.

print_component_and_streamflow([],[]):- !.
print_component_and_streamflow([HeadC|TailC],[HeadF|TailF]):-
        cursor(R,C),
        writef("%4.1 %-9",HeadF,HeadC),
        NR = R + 1,
        cursor(NR,C),
        print_component_and_streamflow(TailC,TailF),!.

print_overhead([],[],OvhdRow):-
        cursor(Row,_),
        OvhdRow = Row,!.
print_overhead([TopHeadC|TopTailC],[TopHeadF|TopTailF],OvhdRow):-
        cursor(R,C),
        writef(" %4.1 %-8",TopHeadF,TopHeadC),
        NR = R + 1,
        cursor(NR,C),
        print_overhead(TopTailC,TopTailF,OvhdRow),!.

print_bottoms([],[],BottomsRow):-
        cursor(Row,_),
        BottomsRow = Row,!.
print_bottoms([BotHeadC|BotTailC],[BotHeadF|BotTailF],BottomsRow):-
        cursor(R,C),
        writef(" %4.1 %-8",BotHeadF,BotHeadC),
        NR = R + 1,
        cursor(NR,C),
        print_bottoms(BotTailC,BotTailF,BottomsRow),!.

decide_cursor(Z,_,_):-
        Z < 18,
              cursor(Z,1),!.
       decide_cursor(Z,Num,LastSepNum):-
              Z >= 18,
              Num < LastSepNum,
              shiftwindow(N),
              pause(N),
              clearwindow,
              write("\nSeparator  Bypass Around  Stream Flow In     Overhead        Bottoms"),
              write("\n---------  -------------  ---------------  --------------
-------------\n"),!.
       decide_cursor(_,Num,Num).
```

BYPASS MODULE

```
/*****************************************************************************************************
        The BYPASS MODULE determines if there is an all-component-inclusive stream that can
        be bypassed. It asks the user if bypass is desired. If bypass is done, the MODULE
        also recalculates the material balance before transferring control back to the MAIN
        MODULE.
*****************************************************************************************************

project "exsep"                                                      % Required for Turbo Prolog only

global domains                                                       % Required for Turbo Prolog only
        alpha = symbol
        aalist = symbol*
        number = integer
        value = real
        vvlist = real*
        str = string
/*                                                    *
 *  These global domains are for Toolbox predicates   *
 *                                                    */
  ROW, COL, LEN, ATTR   = INTEGER
  STRINGLIST = STRING*
  INTEGERLIST = INTEGER*
  KEY    = cr; esc; break; tab; btab; del; bdel; ctrlbdel; ins;
           end ; home ; fkey(INTEGER) ; up ; down ; left ; right ;
           ctrlleft; ctrlright; ctrlend; ctrlhome; pgup; pgdn;
           ctrlpgup; ctrlpgdn; char(CHAR) ; otherspec

global database - bypass                                             % Required for Turbo Prolog only
        dbypass_amount(aalist,alpha,value)
        dbypass_result(aalist,alpha)
        dbypass_status(alpha)

global database - materials                                          % Required for Turbo Prolog only
        flow(alpha,alpha,value)
        k_value(alpha,value)
        boiling_temp(alpha,value)
        initial_components(aalist)
        initial_set(aalist)
        corrosive(alpha)
        pseudoproduct(aalist,alpha)

global database - table                                     % Required for Turbo Prolog only
        dsst(aalist,aalist,alpha,aalist,aalist,alpha,alpha,value,vvlist,vvlist,alpha,alpha,value)
        ddsst(aalist,aalist,alpha,aalist,aalist,alpha,alpha,value,vvlist,vvlist,alpha,alpha,value)
        dddsst(aalist,aalist,alpha,aalist,aalist,alpha,alpha,value,vvlist,vvlist,alpha,alpha,value)
        ddddsst(aalist,aalist,alpha,aalist,aalist,alpha,alpha,value,vvlist,vvlist,alpha,alpha,value)
        dddddsst(aalist,aalist,alpha,aalist,aalist,alpha,alpha,value,vvlist,vvlist,alpha,alpha,
value)
        process_keys(alpha,alpha)
        process_product(alpha)
        sloppy_keys(alpha,alpha)

global database - sequence                                  % Required for Turbo Prolog only
        dseparator(number,aalist,vvlist,aalist,vvlist)
        dstreamin(number,aalist,vvlist)
        dstreambypass(number,alpha,aalist,vvlist)

global predicates                                           % Required for Turbo Prolog only
        append_v(vvlist,vvlist,vvlist) - (i,i,o)
        bypass(aalist,aalist) - (i,i)
        component_flow(aalist,alpha,value) - (i,i,o)
        component_flow_list(aalist,vvlist,aalist,aalist) - (i,i,i,o)
        component_flow_set(aalist,aalist,vvlist) - (i,i,o)
        decide_difficult_or_easy(value,alpha) - (i,o)
        delete_element(alpha,aalist,aalist) - (i,i,o)
        delete_elements(aalist,aalist,aalist) - (i,i,o)
        delete_number(value,vvlist,vvlist) - (i,i,o)
        equal(aalist,aalist) - (i,i)
        ffactor(value,value,value) - (i,i,o)
        first_prod(aalist,aalist) - (i,o)
        flow_set(alpha,aalist,vvlist) - (i,i,o)
        flow_set_c(alpha,aalist,vvlist) - (i,i,o)
        get_split_flow(aalist,value,aalist,value) - (i,o,i,i)
        largest_flow(aalist,alpha,aalist) - (i,o,i)
        last_prod(aalist,aalist) - (i,o)
        length(aalist,value) - (i,o)
        list_member(aalist,aalist) - (i,i)
        max(value,value,value) - (i,i,o)
        max(number,number,number) - (i,i,o)
        max_flist(vvlist,value) - (i,o)
        max_flow(value,value,value) - (i,i,o)
        member(alpha,aalist) - (i,i)
        pause(number) - (i)
        pause_escape
        positive_component_flow_list(aalist,aalist) - (i,o)
        product_flow(alpha,value,aalist) - (i,o,i)
        product_flow_help(alpha,value,aalist,value) - (i,o,i,i)
        product_flow_match(aalist,value,alpha,aalist) - (i,i,o,i)
        product_flow_set(aalist,vvlist,aalist) - (i,o,i)
        reverse(aalist,aalist) - (i,o)
        selected_positive_component_flow_list(aalist,aalist,aalist) - (i,i,o)
        separation_specification_table(aalist,aalist) - (i,i)
        set_above(aalist,alpha,aalist) - (i,i,o)
        set_below(aalist,alpha,aalist) - (i,i,o)
        split(aalist,aalist,aalist,aalist,aalist,aalist) - (i,i,o,o,o,o)
        split_above(aalist,alpha,aalist,aalist) - (i,i,o,o)
        split_below(aalist,alpha,aalist,aalist) - (i,i,o,o)
        sub_list(aalist,aalist) - (i,i)
        sum_component_flow(aalist,value,aalist) - (i,o,i)
        sum_flist(vvlist,value) - (i,o)
        two_highest_flows(vvlist,value,value) - (i,o,o)

database                                                   % Required for Turbo Prolog only
        insmode
        lineinpstate(string,integer)
        lineinpflag

include "tpreds.pro"                                       % Required for Input/Output help
include "menu.pro"                                         % Required for Input/Output help
include "lineinp.pro"                                      % Required for Input/Output help
include "filename.pro"                                     % Required for Input/Output help

predicates                                                 % Required for Turbo Prolog only
        bypassing_choice_list(aalist)
        bypass_flow_list(aalist,alpha,value,aalist,vvlist)
        bypass_flow_of_components(aalist,aalist,value,value,vvlist,vvlist)
        check_bypassing(aalist,aalist)
        check_bypassing_equal(aalist,aalist,aalist)
        flow_of_bypass_products(aalist,aalist,aalist,vvlist,vvlist)
        limiting_component(aalist,alpha,vvlist,vvlist)
        limiting_component_find(value,alpha,alpha,aalist,vvlist,vvlist)
        not_bypassed_flow_list(aalist,vvlist,vvlist,vvlist)
        print_component(aalist,vvlist)
        resolve_bypass_choice(number,aalist,alpha)
        resolve_choice_bypass_percent(number,value)
        solve_bypass_flow(aalist,aalist)
        update_flow_dbase_from_bypass(alpha,aalist,vvlist,aalist)
        update_pseudoproduct_flows(aalist,alpha,value)

/***************************************************************************************************
        All computer code above this point is unique to Turbo Prolog and is required to get
        it to run. Below this point, we begin the actual Prolog relations in the BYPASS
        MODULE. The relations below will run on almost any Prolog system.
***************************************************************************************************/

clauses

        bypass(List,CList):-
                shiftwindow(3),
                clearwindow,
                check_bypassing(List,CList),
                solve_bypass_flow(List,CList),
                write("\nThe bypass analysis is complete for this CAM."),nl,
                retractall(dbypass_result(_,_),bypass),
                retractall(dbypass_amount(_,_,_),bypass),
                clearwindow,
                shiftwindow(1),!.

        check_bypassing(_,[]).

        check_bypassing(List,CList):-
                asserta(dbypass_status("nobypasspossible"),bypass),
                check_bypassing_equal(List,List,CList),
                bypassing_choice_list(List).

        check_bypassing_equal(_,[],_).

        check_bypassing_equal(List,[Head|Tail],CList):-
                selected_positive_component_flow_list([Head],CList,ComponentsInProductStream),
                equal(ComponentsInProductStream,CList),
                assertz(dbypass_result(List,Head),bypass),
        check_bypassing_equal(List,Tail,CList).

check_bypassing_equal(List,[Head|Tail],CList):-
        pseudoproduct([Head|X],ActualProd),
        list_member([Head|X],List),
        selected_positive_component_flow_list([Head|X],CList,ComponentsInProductStream),
        equal(ComponentsInProductStream,CList),
        assertz(dbypass_result(List,ActualProd),bypass),
        check_bypassing_equal(List,Tail,CList).

check_bypassing_equal(List,[_|Tail],CList):-
        check_bypassing_equal(List,Tail,Clist).

bypassing_choice_list(List):-
        dbypass_result(List,Prod),
        asserta(dbypass_status("bypasspossible"),bypass),
        write("\nProduct ",Prod," is all-component_inclusive and subject to bypass.\n"),
        write("Do you wish to bypass this product?\n"),
        menu(9,27,14,14,["Bypass","No Bypass"],"Answer",2,Choicebypass),
        resolve_bypass_choice(Choicebypass,List,Prod),
        retract(dbypass_result(List,Prod),bypass),
        fail.

bypassing_choice_list(_):-
        dbypass_status("bypasspossible"),
        retractall(dbypass_status(_),bypass),
        asserta(dbypass_status("bypasspossible")).

bypassing_choice_list(_):-
        dbypass_status("nobypasspossible"),
        retractall(dbypass_status(_),bypass),
        asserta(dbypass_status("nobypasspossible")),
        write("\nBypass is not possible with this CAM.\n"),
        dseparator(Num,_,_,_,_),
        NNum = Num +1,
        assertz(dstreambypass(NNum,"none",[],[]),sequence),
        shiftwindow(N),
        pause(N).

resolve_bypass_choice(1,List,Prod):-
        write("Choose the percent bypass desired:"),nl,
                menu(9,27,14,14,["Enter manually","90 percent","100
percent"],"Answer",3,ChoiceBypassPercent),
                resolve_choice_bypass_percent(ChoiceBypassPercent,Percent),
                clearwindow,
                Fract = 0.01*Percent,
                assertz(dbypass_amount(List,Prod,Fract)),!.

        resolve_bypass_choice(2,_,Prod):-
                write("Product ",Prod," has been dropped from"),nl,
                write("bypass considerations for this CAM."),nl,
                dseparator(Num,_,_,_),
                NNum = Num +1,
                assertz(dstreambypass(NNum,"none",[],[]),sequence),!.

        resolve_choice_bypass_percent(3,Percent):-
                Percent = 100,!.

        resolve_choice_bypass_percent(2,Percent):-
                Percent = 90,!.

        resolve_choice_bypass_percent(1,Percent):-
                write("\nEnter the percent bypass you desire\n"),
                readreal(Percent),!.

        resolve_choice_bypass_percent(1,Percent):-
                clearwindow,
                write("\nInput error. Entry must be a real number."),
                resolve_choice_bypass_percent(1,Percent),!.

        solve_bypass_flow(List,CList):-
                component_flow_set(List,CList,FList),
                flow_of_bypass_products(List,List,CList,Flist,NFList),
                write(" Feed stream flow rate to separator is:\n"),
                dseparator(Num,_,_,_),
                NNum = Num +1,
                assertz(dstreamin(NNum,CList,NFList),sequence),
                write("Component  Flow"),nl,
                print_component(CList,NFList),
                shiftwindow(N),
                pause(N).
```

```
flow_of_bypass_products(_,[],_,N,N).

flow_of_bypass_products(List,[Head|Tail],NewCList,FList,NFList):-
        dbypass_amount(List,Head,Fract),
        component_flow_set([Head],NewCList,ProductFList),
        limiting_component(NewCList,Component,FList,ProductFList),
        flow(Head,Component,LimitingComponentFlow),
        BypassFlowofLimComp = Fract*LimitingComponentFlow,
        bypass_flow_list(List,Component,BypassFlowofLimComp,NewCList,BypassFlowList),
        not_bypassed_flow_list(NewCList,FList,BypassFlowList,NewFList),
        update_flow_dbase_from_bypass(Head,NewCList,BypassFlowList,List),
        write("Bypass flow to product ",Head," is:"),nl,
        write("Component   Flow Rate"),nl,
        print_component(NewClist,BypassFlowList),
        shiftwindow(N),
        pause(N),
        dseparator(Num,_,_,_,_),
        NNum = Num +1,
        assertz(dstreambypass(NNum,Head,NewCList,BypassFlowList),sequence),
        flow_of_bypass_products(List,Tail,NewCList,NewFList,NFList).

flow_of_bypass_products(List,[_|Tail],NewCList,FList,NFList):-
        dbypass_amount(List,ActualProd,Fract),
        not(member(ActualProd,List)),
        pseudoproduct(PList,ActualProd),
        component_flow_set(PList,NewCList,ProductFList),
        limiting_component(NewCList,Component,FList,ProductFList),
        component_flow(PList,Component,LimitingComponentFlow),
        BypassFlowofLimComp = Fract*LimitingComponentFlow,
        bypass_flow_list(List,Component,BypassFlowofLimComp,NewCList,BypassFlowList),
        not_bypassed_flow_list(NewCList,FList,BypassFlowList,NewFList),
        update_flow_dbase_from_bypass(ActualProd,NewCList,BypassFlowList,List),
        write("Bypass flow to product ",ActualProd," is:"),nl,
        write("Component   Flow Rate"),nl,
        print_component(NewClist,BypassFlowList),
        shiftwindow(N),
        pause(N),
        dseparator(Num,_,_,_,_),
        NNum = Num +1,
        assertz(dstreambypass(NNum,ActualProd,NewCList,BypassFlowList),sequence),
        retract(dbypass_amount(List,ActualProd,Fract),bypass),
        flow_of_bypass_products(List,Tail,NewCList,NewFList,NFList).

flow_of_bypass_products(List,[_|Tail],NewCList,NewFList,NFList):-
        flow_of_bypass_products(List,Tail,NewCList,NewFList,NFList).

limiting_component([Head1|Tail1],Component,[Head2|Tail2],[Head3|Tail3]):-
        Ratio1 = Head3/Head2,
        limiting_component_find(Ratio1,Head1,Component,Tail1,Tail2,Tail3).

limiting_component_find(_,Refcomponent,Component,[],[],[]):-
        Component = Refcomponent.

limiting_component_find(Refratio,Refcomponent,Component,[_|T1],[H2|T2],[H3|T3]):-
        Newratio = H3/H2,
        Newratio > Refratio,
        limiting_component_find(Refratio,Refcomponent,Component,T1,T2,T3).

limiting_component_find(Refratio,_,Component,[H1|T1],[H2|T2],[H3|T3]):-
        Newratio = H3/H2,
        Newratio <= Refratio,
        limiting_component_find(Newratio,H1,Component,T1,T2,T3).

bypass_flow_list(List,Component,BypassFlowofLimComp,NewCList,BypassFlowList):-
        component_flow(List,Component,CFlow),
        bypass_flow_of_components(NewCList,List,CFlow,BypassFlowofLimComp,BypassFlowList,[]
        ).

bypass_flow_of_components([],_,_,_,BypassFlowList,RefList):-
        BypassFlowList = RefList.

bypass_flow_of_components([Head|Tail],List,RefFlow1,RefFlow2,BypassFlowList,RefList):-
        component_flow(List,Head,TotalCFlow),
        BypassFlow = RefFlow2*TotalCFlow/RefFlow1,
        append_v(RefList,[BypassFlow],NewRefList),
        bypass_flow_of_components(Tail,List,RefFlow1,RefFlow2,BypassFlowList,NewRefList).

not_bypassed_flow_list([],[],[],[]).

not_bypassed_flow_list([_|Tail1],[Head2|Tail2],[Head3|Tail3],[Head4|Tail4]):-
        Head4 = Head2 - Head3,
        not_bypassed_flow_list(Tail1,Tail2,Tail3,Tail4).
```

```
print_component([],[]).

print_component([Head1|Tail1],[Head2|Tail2]):-
        writef("   %5.2       %5.2",Head1,Head2),nl,
        print_component(Tail1,Tail2).

update_flow_dbase_from_bypass(_,[],[],_).

update_flow_dbase_from_bypass(Prod,[Head1|Tail1],[Head2|Tail2],List):-
        member(Prod,List),
        flow(Prod,Head1,OldFlow),
        NewFlow = OldFlow - Head2,
        NewFlow >= 0.0001,
        retract(flow(Prod,Head1,OldFlow),materials),
        asserta(flow(Prod,Head1,NewFlow),materials),
        update_flow_dbase_from_bypass(Prod,Tail1,Tail2,List).

update_flow_dbase_from_bypass(Prod,[Head1|Tail1],[Head2|Tail2],List):-
        member(Prod,List),
        flow(Prod,Head1,OldFlow),
        NewFlow = OldFlow - Head2,
        NewFlow < 0.0001,
        retract(flow(Prod,Head1,OldFlow),materials),
        asserta(flow(Prod,Head1,0),materials),
        update_flow_dbase_from_bypass(Prod,Tail1,Tail2,List).

update_flow_dbase_from_bypass(ActualProd,[Head1|Tail1],[Head2|Tail2],List):-
        not(member(ActualProd,List)),
        pseudoproduct(PList,ActualProd),
        update_pseudoproduct_flows(PList,Head1,Head2),
        update_flow_dbase_from_bypass(ActualProd,Tail1,Tail2,List).

update_pseudoproduct_flows([],_,_).

update_pseudoproduct_flows([Head|_],Head1,Head2):-
        flow(Head,Head1,OldFlow),
        OldFlow > 0,
        NewFlow = OldFlow - Head2,
        NewFlow >= 0.0001,
        retract(flow(Head,Head1,OldFlow),materials),
        asserta(flow(Head,Head1,NewFlow),materials).
```

```
update_pseudoproduct_flows([Head|_],Head1,Head2):-
        flow(Head,Head1,OldFlow),
        OldFlow > 0,
        NewFlow = OldFlow - Head2,
        NewFlow >= 0,
        NewFlow < 0.0001,
        retract(flow(Head,Head1,OldFlow),materials),
        asserta(flow(Head,Head1,0),materials).

update_pseudoproduct_flows([_|Tail],Head1,Head2):-
        update_pseudoproduct_flows(Tail,Head1,Head2).
```

SST MODULE

```
/*******************************************************************
 *          SEPARATION SPECIFICATION TABLE MODULE                  *
 *                                                                 *
 * This module develops the SST. Its ultimate goal is to assertz*
 * the following fact to the Prolog database:                     *
 *                                                                 *
 *   (dsst(List,Clist,Split,Top,Bot,LK,HK,Delta,Listofd,Listofb,*
 *        Status,Ease,CES),table)                                  *
 *                                                                 *
 * where:                                                          *
 *       List = list of desired products, e.g., [p1,p2,p3,p4].    *
 *      CList = list of feed components, e.g., [a,b,c,d].         *
 *      Split = "sharp or "sloppy", depending on the split.       *
 *        Top = list of overhead products, e.g., [p1,p2] for      *
 *              a sloppy split, or [a,b] for a sharp split.       *
 *        Bot = list of bottoms products, e.g., [p3,p4] for a     *
 *              sloppy split, or [c,d] for a sharp split.         *
 *         LK = the light-key component in the split              *
 *         HK = the heavy-key component in the split              *
 *      Delta = normal boiling-point temperature difference       *
 *              between the light key and heavy key component,    *
 *              in F or C.                                         *
 *    Listofd = list of overhead recoveries for each component    *
 *              in CList, e.g., for [a,b,c,d] a Listofd could     *
 *              be [0.98,0.98,0.60,0.02].                          *
 *    Listofb = list of bottoms recoveries for each component in*
 *              CList, e.g., [0.02,0.02,0.40,0.98].                *
 *     Status = "feasible" or "infeasible", depending on          *
 *              component-recovery specification test and        *
 *              nonkey-component distribution test.               *
 *       Ease = difficult or easy, depending on the temperature *
 *              difference between the LK and HK components.     *
 *              This parameter applies to heuristic S2. If a     *
 *              split is labelled "difficult", it is considered *
 *              an essential last split by S2.                    *
 *        CES = the coefficient of ease of separation for the    *
 *              split.                                             *
*******************************************************************/
```

```
project "exsep"                                        % Required for Turbo Prolog only

global domains                                         % Required for Turbo Prolog only
     alpha = symbol
     slist = symbol*
     number = integer
     value = real
     rlist = real*
     str = string

/*********************************************************************
 *    These global domains are for Toolbox predicates               *
 *********************************************************************/

  ROW, COL, LEN, ATTR    = INTEGER
  STRINGLIST = STRING*
  INTEGERLIST = INTEGER*
  KEY    = cr; esc; break; tab; btab; del; bdel; ctrlbdel; ins;
        end ; home ; fkey(INTEGER) ; up ; down ; left ; right ;
        ctrlleft; ctrlright; ctrlend; ctrlhome; pgup; pgdn;
        ctrlpgup; ctrlpgdn; char(CHAR) ; otherspec

global database - bypass                               % Required for Turbo Prolog only
     dbypass_amount(slist,alpha,value)
     dbypass_result(slist,alpha)
     dypass_status(alpha)

global database - materials                            % Required for Turbo Prolog only
     flow(alpha,alpha,value)
     k_value(alpha,value)
     boiling_temp(alpha,value)
     initial_components(slist)
     initial_set(slist)
     corrosive(alpha)
     pseudoproduct(slist,alpha)

global database - table                                % Required for Turbo Prolog only
     dsst(slist,slist,alpha,slist,slist,alpha,alpha,value,rlist,rlist,alpha,alpha,value)
     ddsst(slist,slist,alpha,slist,slist,alpha,alpha,value,rlist,rlist,alpha,alpha,value)
     dddsst(slist,slist,alpha,slist,slist,alpha,alpha,value,rlist,rlist,alpha,alpha,value)
     ddddsst(slist,slist,alpha,slist,slist,alpha,alpha,value,rlist,rlist,alpha,alpha,value)
     dddddsst(slist,slist,alpha,slist,slist,alpha,alpha,value,rlist,rlist,alpha,alpha,value)
     process_keys(alpha,alpha)
     process_product(alpha)
     sloppy_keys(alpha,alpha)

global database - sequence                                  % Required for Turbo Prolog only
     dseparator(number,slist,rlist,slist,rlist)
     dstreamin(number,slist,rlist)
     dstreambypass(number,alpha,slist,rlist)

global predicates                                           % Required for Turbo Prolog only
     append_v(rlist,rlist,rlist) - (i,i,o)
     bypass(slist,slist) - (i,i)
     component_flow(slist,alpha,value) - (i,i,o)
     component_flow_list(slist,rlist,slist,slist) - (i,i,i,o)
     component_flow_set(slist,slist,rlist) - (i,i,o)
     decide_difficult_or_easy(value,alpha) - (i,o)
     delete_element(alpha,slist,slist) - (i,i,o)
     delete_elements(slist,slist,slist) - (i,i,o)
     delete_number(value,rlist,rlist) - (i,i,o)
     equal(slist,slist) - (i,i)
     ffactor(value,value,value) - (i,i,o)
     first_prod(slist,slist) - (i,o)
     flow_set(alpha,slist,rlist) - (i,i,o)
     flow_set_c(alpha,slist,rlist) - (i,i,o)
     get_split_flow(slist,value,slist,value) - (i,o,i,i)
     largest_flow(slist,alpha,slist) - (i,o,i)
     last_prod(slist,slist) - (i,o)
     length(slist,value) - (i,o)
     list_member(slist,slist) - (i,i)
     max(value,value,value) - (i,i,o)
     max(number,number,number) - (i,i,o)
     max_flist(rlist,value) - (i,o)
     max_flow(value,value,value) - (i,i,o)
     member(alpha,slist) - (i,i)
     pause(number) - (i)
     pause_escape
     positive_component_flow_list(slist,slist) - (i,o)
     product_flow(alpha,value,slist) - (i,o,i)
     product_flow_match(slist,value,alpha,slist) - (i,i,o,i)
     product_flow_set(slist,rlist,slist) - (i,o,i)
     reverse(slist,slist) - (i,o)
     selected_positive_component_flow_list(slist,slist,slist) - (i,i,o)
     separation_specification_table(slist,slist) - (i,i)
     set_above(slist,alpha,slist) - (i,i,o)
     set_below(slist,alpha,slist) - (i,i,o)
     split(slist,slist,slist,slist,slist,slist) - (i,i,o,o,o,o)
     split_above(slist,alpha,slist,slist) - (i,i,o,o)
     split_below(slist,alpha,slist,slist) - (i,i,o,o)
     sub_list(slist,slist) - (i,i)
     sum_component_flow(slist,value,slist) - (i,o,i)
     sum_flist(rlist,value) - (i,o)
     two_highest_flows(rlist,value,value) - (i,o,o)

database                                            % Required for Turbo Prolog only
     insmode
     lineinpstate(string,integer)
     lineinpflag

include "tpreds.pro"                                % Required for Input/Output help
include "menu.pro"                                  % Required for Input/Output help
include "lineinp.pro"                               % Required for Input/Output help
include "filename.pro"                              % Required for Input/Output help

predicates                                          % Required for Turbo Prolog only
     append(slist,slist,slist)
     assess_feasibility(alpha,alpha,alpha)
     calc_min_stages(slist,slist,alpha,alpha,value,value,value)
     calculate_ces(slist,slist,alpha,alpha,value,value,value,slist)
     calculate_ces_and_split_ease(slist,slist,alpha,alpha,slist,alpha,value,value,alpha)
     ces_log_term(slist,slist,alpha,alpha,value)
     check_print_sst(number)
     component_recovery_dandb(slist,slist,slist,rlist,rlist)
     component_recovery_run(slist,slist,slist,value,value,alpha)
     component_recovery_specification_test(slist,slist,slist,alpha)
     determine_potential_LKs(slist,slist,slist,slist,slist)
     determine_remaining_LKs(slist,slist,slist,slist,slist)
     get_delta_temp(alpha,alpha,value)
     initialize_keys(slist,slist)
     initialize_keys_for_nonkey_test(slist,slist)
     initialize_keys_for_nonkey_test_help(slist,slist)
     initialize_keys_help(slist,slist)
     initialize_products(slist)
     nonkey_component_distribution_test(slist,slist,slist,slist,alpha,alpha)
     nonkey_component_distribution_test_driver(slist,slist,slist,slist)
     print_doverb(rlist,rlist,slist)
     print_sst(alpha)
     sharp_split_dandb(slist,slist,slist,rlist,rlist)
     sharp_split_sst_driver(slist,slist)
     sharp_split_sst(slist,slist,alpha,alpha)
     shorten_list(alpha,slist,slist)
     sloppy_split_dandb(rlist,rlist,rlist)
     sloppy_split_sst(slist,alpha,slist)
     sloppy_split_sst_driver(slist,slist)
     solve_fenske_light(alpha,slist,value,rlist,value)
     solve_fenske_heavy(alpha,slist,value,rlist,value)
     solve_fenske_light_driver(slist,slist,slist,alpha,alpha,value,rlist,alpha)
     solve_fenske_heavy_driver(slist,slist,slist,alpha,alpha,value,rlist,alpha)
     solve_ratio(value,value,value)
     test_HHK_feasibility(slist,slist,slist,rlist,alpha)
     test_LLK_feasibility(slist,slist,slist,rlist,alpha)
     transfer_database

/***************************************************************************************************
        All computer code above this point is unique to Turbo Prolog and is required to get
        it to run. Below this point, we begin the actual Prolog relations in the SST
        MODULE. The relations below will run on almost any Prolog system.
***************************************************************************************************/


clauses

     append([],List,List).
     append([Head|Tail],List1,[Head|List]):-
          append(Tail,List1,List).

/****************************************************************
 * The separation_specification_table clause calls two main     *
 * drivers: sharp_split_sst_driver and sloppy_split_sst_driver. *
 * These drivers assess split feasibility, calculate the CES for*
 * feasible splits, and assert the result into the Prolog       *
 * databse via the assertz(dsst(... )) statement. The SST clause*
 * then reports the results to the user.                        *
 *********************************************************************/

    separation_specification_table(List,CList):-
         shiftwindow(4),
         clearwindow,
         write("\nCalculating Separation Specification Table...\n"),
         initialize_keys(CList,CList),
         sharp_split_sst_driver(List,CList),
         initialize_products(List),
         sloppy_split_sst_driver(List,CList),
         write("\nSST is complete. Enter choice for display\n"),
         menu(9,30,14,14,["Do not display","Print","Display to screen"],"SST Display",3,Choicesst),
         check_print_sst(Choicesst),
         clearwindow,
         shiftwindow(1),!.

    initialize_keys(L,[_|Tail]):-
         !,initialize_keys_help(L,Tail).

    initialize_keys_help(_,[]):-!.
    initialize_keys_help([H1|T1],[H2|T2]):-
         !,assertz(process_keys(H1,H2),table),
         initialize_keys_help(T1,T2).

/****************************************************************
 * The sharp_split_sst_driver clause is an iterative loop.   *
 * It traverses all process_keys(LK,HK) facts in the data-   *
 * base and retracts them. The net result is that every      *
 * sharp split is considered in the sharp_split_sst(List,    *
 * Clist,LK,HK) clause.                                       *
 ****************************************************************/

    sharp_split_sst_driver(List,CList):-
         process_keys(LK,HK),
         sharp_split_sst(List,CList,LK,HK),
         retract(process_keys(LK,HK)),
         fail.
    sharp_split_sst_driver(_,_):- !.

/****************************************************************
 * Heuristic M1: Favor ordinary distillation and avoid MSA's.*
 * However, if the relative volatility between the LK and HK *
 * components is < 1.10, do not use distillation. This clause*
 * asserts "extractive distillation" for status if the       *
 * relative volatility is < 1.10.                            *
 ************************************************************/

   sharp_split_sst(List,CList,LK,HK):-
         k_value(LK,KLK),
         k_value(HK,KHK),
         Alpha = KLK/KHK,
         Alpha <= 1.10,
         split_above(CList,LK,TopComponents,BotComponents),
         sharp_split_dandb(CList,TopComponents,BotComponents,Listofd,Listofb),
         get_delta_temp(LK,HK,Delta),
         assertz(ddsst(List,CList,"sharp",TopComponents,BotComponents,LK,HK,Delta,Listofd,Listofb,
"extractive distillation","neither",0),table),
         !.

   sharp_split_sst(List,CList,LK,HK):-
         get_delta_temp(LK,HK,Delta),
         split_above(CList,LK,TopComponents,BotComponents),
         get_split_flow(List,TopCompFlow,TopComponents,0),
         get_split_flow(List,BotCompFlow,BotComponents,0),
         ffactor(TopCompFlow,BotCompFlow,FFactor),
         CES = FFactor*Delta/log(2401),
         sharp_split_dandb(CList,TopComponents,BotComponents,Listofd,Listofb),
         decide_difficult_or_easy(Delta,Ease),
         assertz(ddsst(List,CList,"sharp",TopComponents,BotComponents,LK,HK,Delta,Listofd,Listofb,
"feasible",Ease,CES),table),
         !.

    get_delta_temp(LK,HK,Delta):-
         boiling_temp(LK,LKT),
         boiling_temp(HK,HKT),
         Delta = abs(HKT-LKT),!.

    sharp_split_dandb([],_,_,[],[]):-!.

    sharp_split_dandb([Head1|Tail1],TopComponents,BotComponents,[HeadD|TailD],[HeadB|TailB]):-
         member(Head1,TopComponents),!,
         HeadD = 0.98,
          HeadB = 0.02,
          sharp_split_dandb(Tail1,TopComponents,BotComponents,TailD,TailB).

     sharp_split_dandb([Head1¦Tail1],TopComponents,BotComponents,[HeadD¦TailD],[HeadB¦TailB]):-
          member(Head1,BotComponents),!,
          HeadD = 0.02,
          HeadB = 0.98,
          sharp_split_dandb(Tail1,TopComponents,BotComponents,TailD,TailB).

     initialize_products([_]):-!.

     initialize_products([H¦T]):-
          assertz(process_product(H),table),
          !,initialize_products(T).

     sloppy_split_sst_driver(List,CList):-
          process_product(Prod),
          sloppy_split_sst(List,Prod,CList),
          retract(process_product(Prod),table),
          fail.

     sloppy_split_sst_driver(_,_):-!.

     sloppy_split_sst(List,Prod,CList):-
          split_above(List,Prod,Top,Bot),
          component_recovery_specification_test(Top,Bot,CList,Status),
          Status = "infeasible",
          component_recovery_dandb(CList,Top,Bot,Listofd,Listofb),
          assertz(ddsst(List,Clist,"Sloppy",Top,Bot,"any","any",0,Listofd,Listofb,"infeasible",
"difficult",0),table),!.

     sloppy_split_sst(List,Prod,CList):-
          split_above(List,Prod,Top,Bot), % Component recovery specification
                                          % test is feasible from above
          determine_potential_LKs(Top,Bot,CList,CList,ListofLKs),
          not(equal(ListofLKs,[])),
          initialize_keys_for_nonkey_test(ListofLKs,CList),
          nonkey_component_distribution_test_driver(List,CList,Top,Bot),!.

/************************************************************
/* This third sloppy_split_sst clause is for when the list *
```

```
* of light keys (ListofLKs) is nil. In this case, all the *
* horizontal splits are in actuality sharp. Thus there's  *
* no true sloppy split that has a LK/HK.                  *
***********************************************************/
    sloppy_split_sst(_,_,_):-!.

    initialize_keys_for_nonkey_test([],_):-!.

    initialize_keys_for_nonkey_test([H|T],CList):-
         shorten_list(H,CList,NCList),
         initialize_keys_for_nonkey_test_help([H|T],NCList).

    shorten_list(X,[X|T],T):-!.

    shorten_list(X,[_|T],List):-
         !,shorten_list(X,T,List).

    initialize_keys_for_nonkey_test_help([],_):-!.

    initialize_keys_for_nonkey_test_help([H1|T1],[H2|T2]):-
         !,assertz(sloppy_keys(H1,H2),table),
         initialize_keys_for_nonkey_test_help(T1,T2).


    component_recovery_specification_test(Top,Bot,[Head|Tail],Status):-
         component_flow(Bot,Head,BotFlow),
         component_flow(Top,Head,TopFlow),
         0 < (TopFlow + BotFlow),
         D = TopFlow/(TopFlow + BotFlow),
         B = BotFlow/(TopFlow + BotFlow),
         component_recovery_run(Top,Bot,Tail,D,B,Status),
         !.

    component_recovery_run(_,_,[],_,_,"feasible").

    component_recovery_run(Top,Bot,[Head|Tail],D,B,Status):-
         component_flow(Top,Head,TopFlow),
         component_flow(Bot,Head,BotFlow),
         Total = TopFlow + BotFlow,
         Total > 0.0,
         NewD = TopFlow/Total,
     D >= NewD,
     NewB = BotFlow/Total,
     B <= NewB,
     component_recovery_run(Top,Bot,Tail,NewD,NewB,Status).

component_recovery_run(Top,Bot,[Head|_],D,_,Status):-
     component_flow(Top,Head,TopFlow),
     component_flow(Bot,Head,BotFlow),
     Total = TopFlow + BotFlow,
     Total > 0.0,
     NewD = TopFlow/Total,
     D < NewD,
     Status = "infeasible".

component_recovery_run(Top,Bot,[Head|_],_,B,Status):-
     component_flow(Top,Head,TopFlow),
     component_flow(Bot,Head,BotFlow),
     Total = TopFlow + BotFlow,
     Total > 0.0,
     NewB = BotFlow/Total,
     B > NewB,
     Status = "infeasible".

determine_potential_LKs(_,_,_,[],[]).

determine_potential_LKs(Top,Bot,CList,[Head1|_],[Head2|Tail2]):-
     component_flow(Top,Head1,TopFlow),
     TopFlow > 0.0,
     component_flow(Bot,Head1,BotFlow),
     BotFlow > 0.0,
     Ratio = TopFlow/BotFlow,
     Ratio < 49.0,
     Ratio > 0.02040816327,   % Head1 is a distributed component
     split_below(CList,Head1,ComponentsAbove,ComponentsBelowIncludingHead1),
     last_prod(ComponentsAbove,PotentialLK),
     not(equal(PotentialLK,[])),  % a component exists above Head1 that is non-distributed
     append([],PotentialLK,[Head2]),    % This is the most volatile LK
     determine_remaining_LKs(Top,Bot,CList,ComponentsBelowIncludingHead1,Tail2).

determine_potential_LKs(Top,Bot,CList,[Head1|_],[Head2|Tail2]):-
     component_flow(Top,Head1,TopFlow),
     TopFlow > 0.0,
     component_flow(Bot,Head1,BotFlow),
     BotFlow > 0.0,
     Ratio = TopFlow/BotFlow,
     Ratio < 49.0,
     Ratio > 0.02040816327,  % Head1 is a distributed component
     split_below(CList,Head1,ComponentsAbove,ComponentsBelowIncludingHead1),
     last_prod(ComponentsAbove,PotentialLK),
     equal(PotentialLK,[]),  % no components exist above Head1; Head1 is the most volatile LK
     append([],[Head1],[Head2]),    % This is the most volatile LK
     delete_element(Head1,ComponentsBelowIncludingHead1,ComponentsBelow),
     determine_remaining_LKs(Top,Bot,CList,ComponentsBelow,Tail2).

determine_potential_LKs(Top,Bot,CList,[_|Tail1],TempListLKs):-
   determine_potential_LKs(Top,Bot,CList,Tail1,TempListLKs).

determine_remaining_LKs(_,_,_,[_],Tail2):-
     Tail2 = [].

determine_remaining_LKs(Top,Bot,CList,[H1|T1],[H2|Tail2]):-
     component_flow(Top,H1,TopFlow),
     TopFlow > 0.0,
     component_flow(Bot,H1,BotFlow),
     BotFlow > 0.0,
     Ratio = TopFlow/BotFlow,
     Ratio < 49.0,
     Ratio > 0.02040816327,  % H1 is a distributed component
     first_prod(T1,PotentialHK),
     not(equal(PotentialHK,[])),  % a HK does exist
     H2 = H1,      % This is a LK
     determine_remaining_LKs(Top,Bot,CList,T1,Tail2).

determine_remaining_LKs(Top,Bot,CList,[_|T1],Tail2):-
     determine_remaining_LKs(Top,Bot,Clist,T1,Tail2).

nonkey_component_distribution_test_driver(List,CList,Top,Bot):-
     sloppy_keys(LK,HK),
     nonkey_component_distribution_test(List,CList,Top,Bot,LK,HK),
     retract(sloppy_keys(LK,HK),table),
     fail.
nonkey_component_distribution_test_driver(_,_,_,_):-!.
```

```
    nonkey_component_distribution_test(List,CList,Top,Bot,LK,HK):-
        calc_min_stages(Top,Bot,LK,HK,Nmin,LKdoverb,HKdoverb),
        solve_fenske_light_driver(CList,Top,Bot,LK,HK,Nmin,LLKdoverbList,LStatus),
        solve_fenske_heavy_driver(CList,Top,Bot,LK,HK,Nmin,HHKdoverbList,HStatus),
        append_v(LLKdoverbList,[LKdoverb,HKdoverb|HHKdoverbList],DoverBList),
        sloppy_split_dandb(DoverBList,Listofd,Listofb),
        assess_feasibility(LStatus,HStatus,Status),
        calculate_ces_and_split_ease(Top,Bot,LK,HK,CList,Status,Delta,CES,Ease),
        assertz(ddsst(List,Clist,"sloppy",Top,Bot,LK,HK,Delta,Listofd,Listofb,Status,Ease,
CES),table),!.

    calc_min_stages(Top,Bot,LK,HK,Nmin,Ratio1,Ratio3):-
        component_flow(Top,LK,TFlow1),
        TFlow1 > 0.0,
        component_flow(Bot,LK,BFlow1),
        component_flow(Top,HK,TFlow2),
        component_flow(Bot,HK,BFlow2),
        BFlow2 > 0.0,
        k_value(LK,KLK),
        k_value(HK,KHK),
        Alpha = KLK/KHK,
        solve_ratio(TFlow1,BFlow1,Ratio1),
        solve_ratio(BFlow2,TFlow2,Ratio2),
        Ratio3 = 1/Ratio2,
        Nmin = log(Ratio1*Ratio2)/log(Alpha),!.

    calc_min_stages(Top,_,LK,_,Nmin,0,0):-
        component_flow(Top,LK,TFlow1),
        TFlow1 <= 0.0,
        Nmin = 0.0.

    calc_min_stages(_,Bot,_,HK,Nmin,0,0):-
        component_flow(Bot,HK,BFlow2),
        BFlow2 <= 0.0,
        Nmin = 0.0.


/************************************************************
 * Fenske equation is feasible on the LLK side if no LLK *
 * components exist.                                      *
 ************************************************************/
```

```
    solve_fenske_light_driver(CList,_,_,LK,_,_,[],"feasible"):-
         set_above(CList,LK,LLKList),
         LLKList = [],!.

/***********************************************************
 * If LLK components do exist, a Fenske feasibility        *
 * analysis is required.                                   *
 ***********************************************************/


    solve_fenske_light_driver(CList,Top,Bot,LK,HK,Nmin,LLKdoverbList,LStatus):-
         set_above(CList,LK,LLKList),
         component_flow(Top,HK,TFLow1),
         component_flow(Bot,HK,BFLow1),
         solve_ratio(TFLow1,BFLow1,Ratio1),  % Ratio1 = (d/b)HK
         solve_fenske_light(HK,LLKList,Ratio1,LLKdoverbList,Nmin),
         test_LLK_feasibility(Top,Bot,LLKList,LLKdoverbList,LStatus).

/***********************************************************
 * The solve_fenske_light clause determines (d/b) for all *
 * LLK components.                                         *
 ***********************************************************/

    solve_fenske_light(_,[],_,[],_).
    solve_fenske_light(HK,[LLK|Tailc],Ratio1,[LLKdoverb|Tailr],Nmin):-
         k_value(HK,KHK),
         k_value(LLK,KLLK),
         Alpha = KLLK/KHK,
         E = exp(1),
         Constant = Nmin*log(Alpha)/log(E),
         LLKdoverb = Ratio1*exp(Constant),
         solve_fenske_light(HK,Tailc,Ratio1,Tailr,Nmin).

/***********************************************************
 * The test_LLK_feasibility clause has four cases:         *
 * First Clause: the base case; if it gets here, the split*
 *   is feasible.                                          *
 * Second Clause: sharp split desired on LLK; split is     *
 *   is feasible if calculated (d/b)LLK >= 0.93/0.07       *
 * Third Clause: nonsharp split desired on LLK; split is  *
 *   feasible if calculated (d/b)LLK is within +/- 20%     *
 *   of desired (d/b)LLK.                              *
 * Fourth Clause: split is infeasible if it fails clauses *
 *   two and three above. No more recursion needed.     *
 ************************************************************/

    test_LLK_feasibility(_,_,[],[],"feasible"):-!.

    test_LLK_feasibility(Top,Bot,[HC¦TC],[HR¦TR],LStatus):-
        component_flow(Top,HC,TFLow),
        component_flow(Bot,HC,BFLow),
        solve_ratio(TFLow,BFLow,Desireddoverb),
        Desireddoverb >= 49.0,
        HR >= 13.28571429,
        test_LLK_feasibility(Top,Bot,TC,TR,LStatus).

    test_LLK_feasibility(Top,Bot,[HC¦TC],[HR¦TR],LStatus):-
        component_flow(Top,HC,TFLow),
        component_flow(Bot,HC,BFLow),
        solve_ratio(TFLow,BFLow,Desireddoverb),
        Lowerbound= 0.8*Desireddoverb,
        Lowerbound <= HR,
        Upperbound = 1.2*Desireddoverb,
        Upperbound >= HR,
        test_LLK_feasibility(Top,Bot,TC,TR,LStatus).


    test_LLK_feasibility(_,_,_,_,"infeasible"):-!.


/************************************************************
 * Fenske equation is feasible on the HHK side if no HHK *
 * components exist.                                    *
 ************************************************************/


    solve_fenske_heavy_driver(CList,_,_,_,HK,_,[],"feasible"):-
        set_below(CList,HK,HHKList),
        HHKList = [],!.


/************************************************************
```

```
 * If HHK components do exist, a Fenske feasibility       *
 * analysis is required.                                  *
 ***********************************************************/


     solve_fenske_heavy_driver(CList,Top,Bot,LK,HK,Nmin,HHKdoverbList,HStatus):-
          set_below(CList,HK,HHKList),
          component_flow(Top,LK,TFLow1),
          component_flow(Bot,LK,BFLow1),
          solve_ratio(BFLow1,TFLow1,Ratio1),  % Ratio1 = (b/d)LK
          solve_fenske_heavy(LK,HHKList,Ratio1,HHKdoverbList,Nmin),
          test_HHK_feasibility(Top,Bot,HHKList,HHKdoverbList,HStatus).


/************************************************************
 * The solve_fenske_heavy clause determines (d/b) for all *
 * HHK components.                                        *
 ***********************************************************/


     solve_fenske_heavy(_,[],_,[],_).
     solve_fenske_heavy(LK,[HHK|Tailc],Ratio1,[HHKdoverb|Tailr],Nmin):-
          k_value(LK,KLK),
          k_value(HHK,KHHK),
          Alpha = KLK/KHHK,
          E = exp(1),
          Constant = Nmin*log(Alpha)/log(E),
          HHKdoverb = 1/(Ratio1*exp(Constant)),
          solve_fenske_heavy(LK,Tailc,Ratio1,Tailr,Nmin).


/************************************************************
 * The test_HHK_feasibility clause has four cases:        *
 * First Clause: the base case; if it gets here, the split*
 *   is feasible.                                         *
 * Second Clause: sharp split desired on HHK; split is    *
 *   is feasible if (b/d)HHK >= 0.93/0.07                 *
 * Third Clause: nonsharp split desired on HHK; split is  *
 *   feasible if (b/d)HHK is within +/- 20% of desired    *
 * Fourth Clause: split is infeasible if it fails clauses *
 *   two and three above. No more recursion needed.       *
 *****************************************************************/

    test_HHK_feasibility(_,_,[],[],"feasible"):-!.

    test_HHK_feasibility(Top,Bot,[HC¦TC],[HR¦TR],HStatus):-
         component_flow(Top,HC,TFLow),
         component_flow(Bot,HC,BFLow),
         solve_ratio(BFLow,TFLow,Desiredboverd),
         Desiredboverd >= 49.0,
         Calcboverd = 1/HR,
         Calcboverd >= 13.28571429,
         test_HHK_feasibility(Top,Bot,TC,TR,HStatus).

    test_HHK_feasibility(Top,Bot,[HC¦TC],[HR¦TR],HStatus):-
         component_flow(Top,HC,TFLow),
         component_flow(Bot,HC,BFLow),
         solve_ratio(BFLow,TFLow,Desiredboverd),
         Calcboverd = 1/HR,
         Lowerbound= 0.8*Desiredboverd,
         Lowerbound <= Calcboverd,
         Upperbound = 1.2*Desiredboverd,
         Upperbound >= Calcboverd,
         test_HHK_feasibility(Top,Bot,TC,TR,HStatus).

    test_HHK_feasibility(_,_,_,_,"infeasible"):-!.

    component_recovery_dandb([],_,_,[],[]).

    component_recovery_dandb([HC¦TC],Top,Bot,[HD¦TD],[HB¦TB]):-
         component_flow(Top,HC,TFLow),
         component_flow(Bot,HC,BFlow),
         solve_ratio(TFlow,BFlow,Ratio),
         HB = 1/(1+Ratio),
         HD = 1 - HB,
         component_recovery_dandb(TC,Top,Bot,TD,TB).

/****************************************************************
 * The sloppy_split_dandb clause is a utility that        *
 * separates the numerical value of d/b into the          *
 * fractional recoveries of d and b such that:            *
*         0<= d <= 1  and 0 <= b <= 1.                   *
**************************************************************/

    sloppy_split_dandb([],[],[]):-!.

    sloppy_split_dandb([H¦T],[HD¦TD],[HB¦TB]):-
         HD = H/(1+H),
         HB = 1 - HD,
         sloppy_split_dandb(T,TD,TB).

    assess_feasibility("feasible","feasible","feasible"):- !.
    assess_feasibility(_,_,"infeasible"):- !.

    calculate_ces_and_split_ease(_,_,LK,HK,_,"infeasible",Delta,0,"difficult"):-
         boiling_temp(LK,LKTemp),
         boiling_temp(HK,HKTemp),
         Delta = abs(HKTemp-LKTemp).

    calculate_ces_and_split_ease(Top,Bot,LK,HK,CList,"feasible",Delta,CES,Ease):-
         ces_log_term(Top,Bot,LK,HK,INVLTerm),
         calculate_ces(Top,Bot,LK,HK,INVLTerm,Delta,CES,CList),
         decide_difficult_or_easy(Delta,Ease).

    solve_ratio(Flow1,Flow2,Ratio):-
         Flow1 > 0.0,
         Flow2 > 0.0,
         Ratio = Flow1/Flow2.

    solve_ratio(Flow1,Flow2,Ratio):-
         Flow2 = 0.0,
         Flow1 > 0.0,
         Ratio = 49.

    solve_ratio(Flow1,Flow2,Ratio):-
         Flow1 = 0.0,
         Flow2 > 0.0,
         Ratio = 0.02040816327.

    /*  CES Log terms  */

/* 0
```

```
        0 0
        0 0
*/
    ces_log_term(Top,Bot,LK,HK,INVLTerm):-
        component_flow(Top,LK,TFlow1),
        TFlow1 =0.0,
        component_flow(Bot,LK,BFlow1),
        BFlow1 = 0.0,
        component_flow(Top,HK,TFlow2),
        TFlow2 = 0.0,
        component_flow(Bot,LK,BFlow2),
        BFlow2 = 0.0,
        INVLTerm = 0.0.

/* 1
        a 0
        0 0
*/
    ces_log_term(Top,Bot,LK,HK,INVLTerm):-
        component_flow(Top,LK,TFlow1),
        TFlow1 > 0.0,
        component_flow(Bot,LK,BFlow1),
        BFlow1 = 0.0,
        component_flow(Top,HK,TFlow2),
        TFlow2 = 0.0,
        component_flow(Bot,HK,BFlow2),
        BFlow2 = 0.0,
        INVLTerm = 0.0,
        write("Must transpose CAM, EXIT"),
        exit.

/* 2
        0 b
        0 0
*/
    ces_log_term(Top,Bot,LK,HK,INVLTerm):-
        component_flow(Top,LK,TFlow1),
        TFlow1 = 0.0,
        component_flow(Bot,LK,BFlow1),
        BFlow1 = 0.0,
        component_flow(Top,HK,TFlow2),
         TFlow2 > 0.0,
         component_flow(Bot,HK,BFlow2),
         BFlow2 = 0.0,
         INVLTerm = 0.0.

/* 3
         0 0
         0 b
*/
     ces_log_term(Top,Bot,LK,HK,INVLTerm):-
         component_flow(Top,LK,TFlow1),
         TFlow1 = 0.0,
         component_flow(Bot,LK,BFlow1),
         BFlow1 = 0.0,
         component_flow(Top,HK,TFlow2),
         TFlow2 = 0.0,
         component_flow(Bot,HK,BFlow2),
         BFlow2 > 0.0,
         INVLTerm = 0.0,
         write("Must transpose CAM, EXIT"),
         exit.

/* 4
         0 0
         a 0
*/
     ces_log_term(Top,Bot,LK,HK,INVLTerm):-
         component_flow(Top,LK,TFlow1),
         TFlow1 = 0.0,
         component_flow(Bot,LK,BFlow1),
         BFlow1 > 0.0,
         component_flow(Top,HK,TFlow2),
         TFlow2 = 0.0,
         component_flow(Bot,HK,BFlow2),
         BFlow2 = 0.0,
         INVLTerm = 0.0.

/* 5
         a b
         0 0
*/
```

```
    ces_log_term(Top,Bot,LK,HK,INVLTerm):-
        component_flow(Top,LK,TFlow1),
        TFlow1 > 0.0,
        component_flow(Bot,LK,BFlow1),
        BFlow1 = 0.0,
        component_flow(Top,HK,TFlow2),
        TFlow2 > 0.0,
        component_flow(Bot,HK,BFlow2),
        BFlow2 = 0.0,
        INVLTerm = 0.0.

/* 6
        a 0
        0 b
*/
    ces_log_term(Top,Bot,LK,HK,INVLTerm):-
        component_flow(Top,LK,TFlow1),
        TFlow1 > 0.0,
        component_flow(Bot,LK,BFlow1),
        BFlow1 = 0.0,
        component_flow(Top,HK,TFlow2),
        TFlow2 = 0.0,
        component_flow(Bot,HK,BFlow2),
        BFlow2 > 0.0,
        Ratio = 2401,
        INVLTerm = 1/log(Ratio).

/* 7
        a 0
        a 0
*/
    ces_log_term(Top,Bot,LK,HK,INVLTerm):-
        component_flow(Top,LK,TFlow1),
        TFlow1 > 0.0,
        component_flow(Bot,LK,BFlow1),
        BFlow1 > 0.0,
        component_flow(Top,HK,TFlow2),
        TFlow2 = 0.0,
        component_flow(Bot,HK,BFlow2),
        BFlow2 = 0.0,
        INVLTerm = 0.0.
```

```
/* 8
        0 b
        0 b
*/
    ces_log_term(Top,Bot,LK,HK,INVLTerm):-
        component_flow(Top,LK,TFlow1),
        TFlow1 = 0.0,
        component_flow(Bot,LK,BFlow1),
        BFlow1 = 0.0,
        component_flow(Top,HK,TFlow2),
        TFlow2 > 0.0,
        component_flow(Bot,HK,BFlow2),
        BFlow2 > 0.0,
        INVLTerm = 0.0.

/* 9
        0 b
        a 0
*/
    ces_log_term(Top,Bot,LK,HK,INVLTerm):-
        component_flow(Top,LK,TFlow1),
        TFlow1 = 0.0,
        component_flow(Bot,LK,BFlow1),
        BFlow1 > 0.0,
        component_flow(Top,HK,TFlow2),
        TFlow2 > 0.0,
        component_flow(Bot,HK,BFlow2),
        BFlow2 = 0.0,
        INVLTerm = 0.0,
        write("Must transpose CAM, EXIT"),
        exit.

/* 10
        0 0
        a b
*/
    ces_log_term(Top,Bot,LK,HK,INVLTerm):-
        component_flow(Top,LK,TFlow1),
        TFlow1 = 0.0,
        component_flow(Bot,LK,BFlow1),
        BFlow1 > 0.0,
          component_flow(Top,HK,TFlow2),
          TFlow2 = 0.0,
          component_flow(Bot,HK,BFlow2),
          BFlow2 > 0.0,
          INVLTerm = 0.0.

/* 11
          a b
          0 b
*/
     ces_log_term(Top,Bot,LK,HK,INVLTerm):-
          component_flow(Top,LK,TFlow1),
          TFlow1 > 0.0,
          component_flow(Bot,LK,BFlow1),
          BFlow1 = 0.0,
          component_flow(Top,HK,TFlow2),
          TFlow2 > 0.0,
          component_flow(Bot,HK,BFlow2),
          BFlow2 > 0.0,
          Ratio = 49*BFlow2/TFlow2,
          INVLTerm = 1/log(Ratio).

/* 12
          0 b
          a b
*/
     ces_log_term(Top,Bot,LK,HK,INVLTerm):-
          component_flow(Top,LK,TFlow1),
          TFlow1 = 0.0,
          component_flow(Bot,LK,BFlow1),
          BFlow1 > 0.0,
          component_flow(Top,HK,TFlow2),
          TFlow2 > 0.0,
          component_flow(Bot,HK,BFlow2),
          BFlow2 > 0.0,
          INVLTerm = 0.0,
          write("Must transpose CAM, EXIT"),
          exit.

/* 13
          a b
          a 0
*/
     ces_log_term(Top,Bot,LK,HK,INVLTerm):-
          component_flow(Top,LK,TFlow1),
          TFlow1 > 0.0,
          component_flow(Bot,LK,BFlow1),
          BFlow1 > 0.0,
          component_flow(Top,HK,TFlow2),
          TFlow2 > 0.0,
          component_flow(Bot,HK,BFlow2),
          BFlow2 = 0.0,
          INVLTerm = 0.0,
          write("Must transpose CAM, EXIT"),
          exit.

/* 14
          a 0
          a b
*/
     ces_log_term(Top,Bot,LK,HK,INVLTerm):-
          component_flow(Top,LK,TFlow1),
          TFlow1 > 0.0,
          component_flow(Bot,LK,BFlow1),
          BFlow1 > 0.0,
          component_flow(Top,HK,TFlow2),
          TFlow2 = 0.0,
          component_flow(Bot,HK,BFlow2),
          BFlow2 > 0.0,
          Ratio = 49*TFlow1/BFlow1,
          INVLTerm = 1/log(Ratio).

/* 15
          a b
          a b
*/
     ces_log_term(Top,Bot,LK,HK,INVLTerm):-
          component_flow(Top,LK,TFlow1),
          TFlow1 > 0.0,
          component_flow(Bot,LK,BFlow1),
          BFlow1 > 0.0,
          component_flow(Top,HK,TFlow2),
     TFlow2 > 0.0,
     component_flow(Bot,HK,BFlow2),
     BFlow2 > 0.0,
     Ratio = TFlow1*BFlow2/BFlow1/TFlow2,
     INVLTerm = 1/log(Ratio).

calculate_ces(Top,Bot,LK,HK,INVLTerm,Delta,CES,CList):-
     component_flow(Top,LK,TFlow1),
     TFlow1 > 0.0,
     component_flow(Bot,HK,BFlow2),
     BFlow2 > 0.0,
     get_split_flow(Top,TopFlow,CList,0),
     get_split_flow(Bot,BotFlow,CList,0),
     boiling_temp(LK,LKTemp),
     boiling_temp(HK,HKTemp),
     Delta = abs(HKTemp-LKTemp),
     ffactor(TopFlow,BotFlow,FFactor),
     CES = FFactor*Delta*INVLTerm.

calculate_ces(Top,_,LK,_,_,Delta,CES,_):-
     component_flow(Top,LK,TFlow1),
     TFlow1 <= 0.0,
     Delta = 0.0,
     CES = 0.0.

calculate_ces(_,Bot,_,HK,_,Delta,CES,_):-
     component_flow(Bot,HK,BFlow1),
     BFlow1 <= 0.0,
     Delta = 0.0,
     CES = 0.0.

check_print_sst(1):-
     transfer_database.

check_print_sst(2):-
     writedevice(printer),
     print_sst("P"),
     writedevice(screen).

check_print_sst(3):-
     clearwindow,
        print_sst("S").

    print_sst("P"):-
        ddsst(List,CList,Type,Top,Bot,LK,HK,Delta,Listofd,Listofb,Feasibility,Ease,CES),
        write("Split: ",Top,"/",Bot," LK/HK: ",LK,"/",HK,"   Status: ",Feasibility),nl,
        print_doverb(Listofd,Listofb,CList),
        k_value(LK,KLK),
        k_value(HK,KHK),
        Alpha = KLK/KHK,
        writef("LK/HK Temperature Difference (C): %-7.2 CES:%-7.2
        Alpha(LK,HK):%7.2",Delta,CES,Alpha),nl,
        write("------------------------------------------------------------------------"),nl,
        retract(ddsst(List,CList,Type,Top,Bot,LK,HK,Delta,Listofd,Listofb,Feasibility,Ease,
CES),table),
        assertz(dsst(List,CList,Type,Top,Bot,LK,HK,Delta,Listofd,Listofb,Feasibility,Ease,
CES),table),
        fail.

    print_sst("S"):-
        ddsst(List,CList,Type,Top,Bot,LK,HK,Delta,Listofd,Listofb,Feasibility,Ease,CES),
        write("Split: ",Top,"/",Bot," LK/HK: ",LK,"/",HK,"   Status: ",Feasibility),nl,
        print_doverb(Listofd,Listofb,CList),
        k_value(LK,KLK),
        k_value(HK,KHK),
        Alpha = KLK/KHK,
        writef("LK/HK Temperature Difference (C): %-7.2 CES:%-7.2
        Alpha(LK,HK):%7.2",Delta,CES,Alpha),nl,
        write("------------------------------------------------------------------------"),nl,nl,
        retract(ddsst(List,CList,Type,Top,Bot,LK,HK,Delta,Listofd,Listofb,Feasibility,Ease,
CES),table),
        assertz(dsst(List,CList,Type,Top,Bot,LK,HK,Delta,Listofd,Listofb,Feasibility,Ease,
CES),table),
        shiftwindow(N),
        pause(N),
        fail.

    print_sst(_).

    print_doverb([],[],[]):- nl,!.

    print_doverb([Headd|Taild],[Headb|Tailb],[HeadC|TailC]):-
```

```
          writef("(d/b)%-1 %4.2/%-4.2f",HeadC,Headd,Headb),nl,
          print_doverb(Taild,Tailb,TailC).

     transfer_database:-
          ddsst(List,CList,Type,Top,Bot,LK,HK,Delta,Listofd,Listofb,Feasibility,Ease,CES),
          retract(ddsst(List,CList,Type,Top,Bot,LK,HK,Delta,Listofd,Listofb,Feasibility,Ease,
CES),table),
          assertz(dsst(List,CList,Type,Top,Bot,LK,HK,Delta,Listofd,Listofb,Feasibility,Ease,
CES),table),
          fail.
     transfer_database.
```

SPLIT MODULE

```
/*******************************************************************
 * SPLIT MODULE --  This module implements the rank-ordered    *
 * heuristics of Nadgir and Liu, and heuristically chooses a   *
 * split that has been previously generated and assessed as    *
 * being technically feasible.                                 *
 *******************************************************************/

project "exsep"                                   % Required for Turbo Prolog only

global domains                                    % Required for Turbo Prolog only
     alpha = symbol
     mlist = symbol*
     number = integer
     value = real
     vvlist = real*
     str = string

/*******************************************************************
 *  These global domains are for Toolbox predicates            *
 *******************************************************************/

  ROW, COL, LEN, ATTR   = INTEGER
  STRINGLIST = STRING*
  INTEGERLIST = INTEGER*
  KEY    = cr; esc; break; tab; btab; del; bdel; ctrlbdel; ins;
        end ; home ; fkey(INTEGER) ; up ; down ; left ; right ;
        ctrlleft; ctrlright; ctrlend; ctrlhome; pgup; pgdn;
        ctrlpgup; ctrlpgdn; char(CHAR) ; otherspec

global database - bypass                          % Required for Turbo Prolog only
     dbypass_amount(mlist,alpha,value)
     dbypass_result(mlist,alpha)
     dbypass_status(alpha)

global database - materials                       % Required for Turbo Prolog only
     flow(alpha,alpha,value)
     k_value(alpha,value)
     boiling_temp(alpha,value)
     initial_components(mlist)
     initial_set(mlist)
     corrosive(alpha)
     pseudoproduct(mlist,alpha)

global database - table                                        % Required for Turbo Prolog only
     dsst(mlist,mlist,alpha,mlist,mlist,alpha,alpha,value,vvlist,vvlist,alpha,alpha,value)
     ddsst(mlist,mlist,alpha,mlist,mlist,alpha,alpha,value,vvlist,vvlist,alpha,alpha,value)
     dddsst(mlist,mlist,alpha,mlist,mlist,alpha,alpha,value,vvlist,vvlist,alpha,alpha,value)
     ddddsst(mlist,mlist,alpha,mlist,mlist,alpha,alpha,value,vvlist,vvlist,alpha,alpha,value)
     dddddsst(mlist,mlist,alpha,mlist,mlist,alpha,alpha,value,vvlist,vvlist,alpha,alpha,value)
     process_keys(alpha,alpha)
     process_product(alpha)
     sloppy_keys(alpha,alpha)

global database - sequence                                     % Required for Turbo Prolog only
     dseparator(number,mlist,vvlist,mlist,vvlist)
     dstreamin(number,mlist,vvlist)
     dstreambypass(number,alpha,mlist,vvlist)

global predicates                                              % Required for Turbo Prolog only
     append_v(vvlist,vvlist,vvlist) - (i,i,o)
     bypass(mlist,mlist) - (i,i)
     component_flow(mlist,alpha,value) - (i,i,o)
     component_flow_list(mlist,vvlist,mlist,mlist) - (i,i,i,o)
     component_flow_set(mlist,mlist,vvlist) - (i,i,o)
     decide_difficult_or_easy(value,alpha) - (i,o)
     delete_element(alpha,mlist,mlist) - (i,i,o)
     delete_elements(mlist,mlist,mlist) - (i,i,o)
     delete_number(value,vvlist,vvlist) - (i,i,o)
     equal(mlist,mlist) - (i,i)
     ffactor(value,value,value) - (i,i,o)
     first_prod(mlist,mlist) - (i,o)
     flow_set(alpha,mlist,vvlist) - (i,i,o)
     flow_set_c(alpha,mlist,vvlist) - (i,i,o)
     get_split_flow(mlist,value,mlist,value) - (i,o,i,i)
     largest_flow(mlist,alpha,mlist) - (i,o,i)
     last_prod(mlist,mlist) - (i,o)
     length(mlist,value) - (i,o)
     list_member(mlist,mlist) - (i,i)
     max(value,value,value) - (i,i,o)
     max(number,number,number) - (i,i,o)
     max_flist(vvlist,value) - (i,o)
     max_flow(value,value,value) - (i,i,o)
     member(alpha,mlist) - (i,i)
     pause(number) - (i)
     pause_escape
     positive_component_flow_list(mlist,mlist) - (i,o)
     product_flow(alpha,value,mlist) - (i,o,i)
     product_flow_help(alpha,value,mlist,value) - (i,o,i,i)
     product_flow_match(mlist,value,alpha,mlist) - (i,i,o,i)
     product_flow_set(mlist,vvlist,mlist) - (i,o,i)
     reverse(mlist,mlist) - (i,o)
     selected_positive_component_flow_list(mlist,mlist,mlist) - (i,i,o)
     separation_specification_table(mlist,mlist) - (i,i)
     set_above(mlist,alpha,mlist) - (i,i,o)
     set_below(mlist,alpha,mlist) - (i,i,o)
     split(mlist,mlist,mlist,mlist,mlist,mlist) - (i,i,o,o,o,o)
     split_above(mlist,alpha,mlist,mlist) - (i,i,o,o)
     split_below(mlist,alpha,mlist,mlist) - (i,i,o,o)
     sub_list(mlist,mlist) - (i,i)
     sum_component_flow(mlist,value,mlist) - (i,o,i)
     sum_flist(vvlist,value) - (i,o)
     two_highest_flows(vvlist,value,value) - (i,o,o)

database                                          % Required for Turbo Prolog only
     insmode
     lineinpstate(string,integer)
     lineinpflag

include "tpreds.pro"                              % Required for Input/Output help
include "menu.pro"                                % Required for Input/Output help
include "lineinp.pro"                             % Required for Input/Output help
include "filename.pro"                            % Required for Input/Output help

/*******************************************************************
 *  These predicates are unique to the SPLIT module.              *
 *******************************************************************/

predicates                                        % Required for Turbo Prolog only
     adjust_database(mlist,mlist,alpha,mlist,mlist,alpha,alpha)
     append(mlist,mlist,mlist)
     check_sharp_splits(mlist,mlist,mlist,alpha)
     check_sloppy_splits(mlist,mlist,mlist,alpha)
     choose_split(number)
     decide_split_choice(mlist,mlist,alpha,alpha,alpha,value,value,mlist,mlist,alpha,alpha,alpha,val
     ue,value,mlist,mlist,alpha,alpha,alpha,value,value,alpha)
     decide_split_path(number)
     determine_best_split_ratio(mlist,mlist,mlist,mlist,alpha,alpha,alpha,value,value,mlist,mlist,al
     pha,alpha,alpha,value,value,alpha)
     determine_sharp_products(mlist,mlist,mlist,mlist)
     determine_two_best_split_ratios(mlist,mlist,mlist,mlist,alpha,alpha,alpha,value,value,mlist,mli
     st,alpha,alpha,alpha,value,value)
     determine_whether_to_use_CES(value,value,alpha)
     get_split(mlist,mlist,alpha,mlist,mlist,alpha,alpha,value)
     line_write(mlist,mlist,alpha,alpha,value,value,value,value,alpha)
     max_ces_sharp(mlist,mlist,mlist,value,value,alpha)
     max_ces_sloppy(mlist,mlist,mlist,value,value,alpha)
     minimum(value,value,value)
     readjust_database(mlist,mlist)
     remove_split_from_choice(mlist,mlist,mlist,mlist,mlist,alpha)
     resulting_products(mlist,alpha,mlist,mlist,mlist,mlist,mlist,mlist)
     search_splits_for_max(mlist,mlist,mlist,mlist,mlist,value,value,alpha)
     solve_ratio(value,value,value)


/*********************************************************************************************************
     All computer code above this point is unique to Turbo Prolog and is required to get it to
     run. Below this point, we begin the actual Prolog relations in the SPLIT MODULE. The
     relations below will run on almost any Prolog system.
*********************************************************************************************************/


clauses

     append([],List,List).

     append([Head|Tail],List1,[Head|List]):-
          append(Tail,List1,List).

/******************************************************************
 * The split clause is the central control clause in the SPLIT *
 * module. It implements heuristics S1, S2, C1, and C2, chooses*
 * the best split based on the heuristics, and determines the  *
 * overhead and bottoms streams for the splitter.              *
 ***************************************************************/

/***************************************************************
 *         Base cases and trivial solutions                    *
 ***************************************************************/

    split([],_,[],[],[],[]):-!.                                    % rule 1
    split([_],_,[],[],[],[]):-!.                                   % rule 2
    split(_,[],[],[],[],[]):-!.                                    % rule 3
    split(_,[_],[],[],[],[]):-!.                                   % rule 4

/***************************************************************
 *         Implementation of heuristics M1 and M2              *
 ***************************************************************/

    split(List,CList,_,_,_,_):-                                    % rule 5
         clearwindow,
         write("**          Heuristic M1: Favor ordinary distillation.          **"),nl,
         write("******************************************************************"),nl,
         dsst(List,CList,Type,Top,Bot,LK,HK,Listofd,Listofb,Delta,"extractive
distillation",Ease,CES),
         k_value(LK,KLK),
         k_value(HK,KHK),
         Alpha = KLK/KHK,
         nl,write("Normal distillation for split ",Top,"/",Bot),nl,
         write("with ",LK,"/",HK," as LK/HK"),nl,
         write("and alpha(LK,HK) = "),
         writef("%5.2",Alpha),write(" is not recommended."),nl,
         write("Due to the low relative volatility, an alternate"),nl,
         write("separation process is recommended."),nl,
         shiftwindow(N),pause(N),
         retract(dsst(List,CList,Type,Top,Bot,LK,HK,Listofd,Listofb,Delta,"extractive
distillation",Ease,CES),table),
         assertz(ddsst(List,CList,Type,Top,Bot,LK,HK,Listofd,Listofb,Delta,"extractive
distillation",Ease,CES),table),
         fail.

   split(List,CList,_,_,_,_):-                                     % rule 6
         ddsst(List,CList,Type,Top,Bot,LK,HK,Listofd,Listofb,Delta,"extractive
distillation",Ease,CES),
          retract(ddsst(List,CLIst,Type,Top,Bot,LK,HK,Listofd,Listofb,Delta,"extractive
distillation",Ease,CES),table),
          assertz(dsst(List,CList,Type,Top,Bot,LK,HK,Listofd,Listofb,Delta,"extractive
distillation",Ease,CES),table),
          fail.



/*****************************************************************
 *           Implementation of heuristic M2                 *
 *****************************************************************/


/*   split(List,CLIst,_,_,_,_):-
          clearwindow,
          bubble_point,
          dew_point,
          ... implemented off-line ...*/

/*****************************************************************
 *         Implementation of heuristic S1                    *
 *****************************************************************/


      split(List,[H¦T],TopList,TopCList,BotList,BotCList):-                % rule 7
          clearwindow,
          write("** Heuristic S1: Remove corrosive and hazardours materials first. **"),nl,
          write("*********************************************************************"),nl,nl,
          corrosive(Element),
          CList = [H¦T],
          H = Element,
          dsst(List,CList,"sharp",Top,Bot,H,_,_,_,_,"feasible",_,_),
          write("Component ",Element," is corrosive."),nl,
          write("The recommended split is ",Top," / ",Bot,"\nDo you wish to make this split?\n"),
          menu(13,55,14,14,["Split","Do Not Split"],"Answer",2,Choicesplit),
          choose_split(Choicesplit),
          resulting_products(List,"sharp",Top,Bot,TopList,TopCList,BotList,BotCList),
          dseparator(Num,_,_,_,_),
          NNum = Num +1,
          component_flow_set(TopList,TopCList,TopFList),
          component_flow_set(BotList,BotCList,BotFList),
          asserta(dseparator(NNum,TopCList,TopFList,BotCList,BotFList),sequence),
     retractall(dsst(List,CList,_,_,_,_,_,_,_,_,_,_),table),
     retractall(ddsst(List,CList,_,_,_,_,_,_,_,_,_,_),table),!.

split(List,CList,TopList,TopCList,BotList,BotCList):-              % rule 8
     corrosive(Element),
     last_prod(CList,Bot),
     Bot = [H¦_],
     H = Element,
     dsst(List,CList,"sharp",Top,Bot,_,H,_,_,_,"feasible",_,_),
     write("Component ",Element," is corrosive."),nl,
     write("The recommended split is ",Top," / ",Bot,"\nDo you wish to make this split?\n"),
     menu(13,55,14,14,["Split","Do Not Split"],"Answer",2,Choicesplit),
     choose_split(Choicesplit),
     resulting_products(List,"sharp",Top,Bot,TopList,TopCList,BotList,BotCList),
     dseparator(Num,_,_,_,_),
     NNum = Num +1,
     component_flow_set(TopList,TopCList,TopFList),
     component_flow_set(BotList,BotCList,BotFList),
     asserta(dseparator(NNum,TopCList,TopFList,BotCList,BotFList),sequence),
     retractall(dsst(List,CList,_,_,_,_,_,_,_,_,_,_),table),
     retractall(ddsst(List,CList,_,_,_,_,_,_,_,_,_,_),table),!.

split(_,[Head¦Tail],_,_,_,_):-                                     % rule 9
     corrosive(Element),
     Element <> Head,
     member(Element,[Head¦Tail]),
     last_prod([Head¦Tail],Bot),
     Bot = [H¦_],
     Element <> H,
     write("Component ",Element," is corrosive."),nl,
     write("However, it cannot be isolated in a sharp split."),nl,
     write("It is recommended to proceed with other heuristics."),
     shiftwindow(N),
     pause(N),
     fail.


split(_,CList,_,_,_,_):-                                           % rule 10
     corrosive(Element),
     not(member(Element,CList)),
     write("No corrosive elements exist."),nl,
          write("It is recommended to apply other heurstics"),nl,
          shiftwindow(N),
          pause(N),
          fail.

/*******************************************************
 * Heuristic S2: Perform difficult separations last.  *
 * This has already been implented in the dsst state- *
 * ment when constructing SST. It has the qualifier   *
 * "easy" or "difficult" based on boiling point dif-  *
 * ference less than 20 degrees C. These results are  *
 * merely shown to the user below.                    *
 ******************************************************/

     split(List,CList,_,_,_,_):-                                      % rule 11
          clearwindow,shiftwindow(N),
          write("**       Heuristic S2: Perform difficult separations last.        **"),nl,
          write("**********************************************************************"),nl,nl,
          dsst(List,CList,Type,Top,Bot,LK,HK,Delta,Listofd,Listofb,"feasible","difficult",CES),
          retract(dsst(List,CList,Type,Top,Bot,LK,HK,Delta,Listofd,Listofb,"feasible","difficult",
CES),table),
          assertz(ddsst(List,CList,Type,Top,Bot,LK,HK,Delta,Listofd,Listofb,"feasible","difficult",
CES),table),
          write("Split ",Top,"/",Bot," with ",LK,"/",HK,"as LK/HK"),nl,
          write("has a normal boiling point temperature difference of ",Delta),nl,
          write("and has been identified as an essential last split by heuristic S1."),nl,nl,
          pause(N),
          fail.

     split(List,CList,_,_,_,_):-                                      % rule 12
          not(ddsst(List,CList,_,_,_,_,_,_,_,_,"feasible","difficult",_)),
          write("No splits are particularly difficult."),nl,
          write("Heuristic S1 does not apply, and no splits\nare identified as essential last
splits."),nl,
          shiftwindow(N),
          pause(N),
          fail.

     split(List,CList,_,_,_,_):-                                      % rule 13
          ddsst(List,CList,Type,Top,Bot,LK,HK,Delta,Listofd,Listofb,"feasible","difficult",CES),
          retract(ddsst(List,CList,Type,Top,Bot,LK,HK,Delta,Listofd,Listofb,"feasible","difficult",
CES),table),
         assertz(dsst(List,CList,Type,Top,Bot,LK,HK,Delta,Listofd,Listofb,"feasible","difficult",
CES),table),
         fail.

/*******************************************************
 * Heuristic C1: Remove most plentiful product first. *
 * This section retracts all splits from SST (with    *
 * interactive user approval) if they do not fully    *
 * remove the most plentiful product. After           *
 * retraction, the clause fails to force splitting of *
 * remaining splits via heuristic C2.                 *
 *******************************************************/

    split(List,CList,_,_,_,_):-                                          % rule 14
         clearwindow,
         write("**     Heuristic C1: Remove the most plentiful product first.     **"),nl,
         write("**********************************************************************"),nl,nl,
         product_flow_set(List,FList,CList),
         two_highest_flows(Flist,MaxFlow,SecondFlow),
         MaxFlow/SecondFlow <= 1.2,
         write("No product is particularly dominant in "),nl,
         write("flow rate. Heuristic C1 does not apply."),nl,
         shiftwindow(N),
         pause(N),
         fail.

    split(List,CList,_,_,_,_):-                                          % rule 15
         product_flow_set(List,FList,CList),
         two_highest_flows(Flist,MaxFlow,SecondFlow),
         MaxFlow/SecondFlow > 1.2,
         sum_flist(Flist,TFlow),
         MaxFlow/TFlow <= 0.285,
         write("No product is particularly dominant in "),nl,
         write("flow rate. Heuristic C1 does not apply."),nl,
         shiftwindow(N),
         pause(N),
         fail.

    split(List,CList,_,_,_,_):-                                          % rule 16
         product_flow_set(List,FList,CList),
          two_highest_flows(Flist,MaxFlow,SecondFlow),
          MaxFlow/SecondFlow > 1.2,
          sum_flist(Flist,TFlow),
          MaxFlow/TFlow > 0.285,
          product_flow_match(List,MaxFlow,Prod,CList),
          check_sloppy_splits(List,List,CList,Prod),
          selected_positive_component_flow_List([Prod],CList,CompList),
          not(equal(CList,CompList)),
          check_sharp_splits(List,CList,CList,Prod),
          fail.

     split(List,CList,_,_,_,_):-                                          % rule 17
          product_flow_set(List,FList,CList),
          two_highest_flows(Flist,MaxFlow,SecondFlow),
          MaxFlow/SecondFlow > 1.2,
          sum_flist(Flist,TFlow),
          MaxFlow/TFlow > 0.285,
          product_flow_match(List,MaxFlow,Prod,CList),
          selected_positive_component_flow_list([Prod],CList,CompList),
          equal(CList,CompList),
          write("\nNote: product ",Prod," is the most plentiful. It is also"),
          write("\nall-component-inclusive. Therefore, heuristic C1 will not \nbe applied to sharp
splits here."),
          shiftwindow(N),
          pause(N),
          fail.

/*************************************************************
 * Heuristic C2: Favor a 50/50 split. This split closest to   *
 * 50/50 will be chosen. If another split is within 30% of a  *
 * 50/50 split, then the CES will also be used.               *
 *************************************************************/

     split(List,CList,TopList,TopCList,BotList,BotCList):-                % rule 18
          clearwindow,
          write("**               Heuristic C2: Favor a 50/50 Split             **"),nl,
          write("*****************************************************************"),nl,
          write("                          Alpha  Distillate  Bottoms  Molar Split      "),nl,
          write("   Split       LK    HK   LK,HK     Flow       Flow     Ratio    CES"),nl,
          write("------------  ----  ----  -------  ---------  ---------  -----  -----"),nl,
          determine_two_best_split_ratios(List,CList,Top1,Bot1,Type1,LK1,HK1,Ratio1,CES1,Top2,Bot2,
Type2,LK2,HK2,Ratio2,CES2),
          Testratio = 1.3*Ratio1,
          determine_whether_to_use_CES(Testratio,Ratio2,Answer),
          decide_split_choice(Top1,Bot1,Type1,LK1,HK1,Ratio1,CES1,Top2,Bot2,Type2,LK2,HK2,Ratio2,
CES2,Top,Bot,Type,LK,HK,Ratio,CES,Answer),
          X = 100/(Ratio + 1),
          Y = 100 - X,
          max(X,Y,D),
          B = 100 - D,
          clearwindow,
          write("\n\nSplit ",Top,"/",Bot," with:"),nl,
          write("    -- components ",LK,"/",HK," as LK/HK"),nl,
          write("    -- a CES = "),writef("%5.2",CES),nl,
          write("    -- a molar split ratio = "),writef("%2.0",D),write("/"),writef("%2.0",B),nl,
          write("is the recommended split. Do you\nwish to make this split?\n"),
          menu(13,55,14,14,["Split","Do Not Split"],"Answer",2,Choicesplit),
          choose_split(Choicesplit),
          resulting_products(List,Type,Top,Bot,TopList,TopCList,BotList,BotCList),
          dseparator(Num,_,_,_,_),
          NNum = Num +1,
          component_flow_set(TopList,TopCList,TopFList),
          component_flow_set(BotList,BotCList,BotFList),
          asserta(dseparator(NNum,TopCList,TopFList,BotCList,BotFList),sequence),
          retractall(dsst(List,CList,_,_,_,_,_,_,_,_,_,_),table),
          retractall(ddsst(List,CList,_,_,_,_,_,_,_,_,_,_),table),!.

/*********************************************************
 * Heuristic S2: Do difficult splits last. Difficult    *
 * separations are in dsst statement. This section splits*
 * difficult separations after other splits are complete.*
 * If more than one difficult separation exist, then the *
 * one with the maximum CES is chosen first.             *
 *********************************************************/

    split(List,CList,TopList,TopCLIst,BotList,BotCList):-               % rule 19
         max_ces_sloppy(List,List,CList,0,CESsloppy,"difficult"),
         max_ces_sharp(List,CList,CList,0,CESsharp,"difficult"),
         max(CESsharp,CESsloppy,CES),
         CES > 0.0,
         dsst(List,CList,Type,Top,Bot,LK,HK,_,_,_,"feasible","difficult",CES),
         write("Difficult separation"),nl,
        write("Split ",Top,"/",Bot),nl,
        write("with ",LK,"/",HK," as LK/HK"),nl,
        write("and CES = ",CES),nl,
        write("is the recommended split. Do you\n wish to make this split?\n"),
        menu(13,55,14,14,["Split","Do Not Split"],"Answer",2,Choicesplit),
        choose_split(Choicesplit),
        resulting_products(List,Type,Top,Bot,TopList,TopCList,BotList,BotCList),
        dseparator(Num,_,_,_,_),
        NNum = Num +1,
        component_flow_set(TopList,TopCList,TopFList),
        component_flow_set(BotList,BotCList,BotFList),
        asserta(dseparator(NNum,TopCList,TopFList,BotCList,BotFList),sequence),
        retractall(dsst(List,CList,_,_,_,_,_,_,_,_,_,_),table),
        retractall(ddsst(List,CList,_,_,_,_,_,_,_,_,_,_,_),table),!.

/*     This section is if no splits above were executed.  */

    split(List,CList,TopList,TopCList,BotList,BotCList):-               % rule 20
        clearwindow,
        write("Heursitics complete. No splits have yet been\nchosen. Heuristic search can be
reinitiated\nor a split can be chosen maually. Enter your choice.\n"),
        menu(13,23,14,14,["Heuristically","Manually"],"Search Technique",2,Choicesplit),
        decide_split_path(Choicesplit),
        dsst(List,CList,Type,Top,Bot,LK,HK,Delta,Listofd,Listofb,"feasible",Ease,CES),
        retract(dsst(List,CList,Type,Top,Bot,LK,HK,Delta,Listofd,Listofb,"feasible",Ease,
CES),table),
        assertz(ddsst(List,CList,Type,Top,Bot,LK,HK,Delta,Listofd,Listofb,"feasible",Ease,
CES),table),
        write("\nIf desired, ",Type," split",Top,"/",Bot,"\nwith a CES = "),
        writef("%5.2",CES),
        write(" can be done. Please enter your choice.\n"),
        menu(13,55,14,14,["Split","Do Not Split"],"Answer",2,SecondChoicesplit),
        choose_split(SecondChoicesplit),
        resulting_products(List,Type,Top,Bot,TopList,TopCList,BotList,BotCList),
        dseparator(Num,_,_,_,_),
        NNum = Num +1,
        component_flow_set(TopList,TopCList,TopFList),
        component_flow_set(BotList,BotCList,BotFList),
        asserta(dseparator(NNum,TopCList,TopFList,BotCList,BotFList),sequence),
        retractall(dsst(List,CList,_,_,_,_,_,_,_,_,_,_),table),
        retractall(ddsst(List,CList,_,_,_,_,_,_,_,_,_,_,_),table),!.
```

```
/*     This section is if no manual split is chosen. The *
 *     above rule would then fail                        */

    split(List,CList,_,_,_,_):-                                     % rule 21
        clearwindow,
        write("\n\nNo splits were chosen manually.\nThe heuristic search will be
reinitiated...\n"),
        ddsst(List,CList,Type,Top,Bot,LK,HK,Delta,Listofd,Listofb,"feasible",Ease,CES),
        retract(ddsst(List,CList,Type,Top,Bot,LK,HK,Delta,Listofd,Listofb,"feasible",Ease,
CES),table),
        assertz(dsst(List,CList,Type,Top,Bot,LK,HK,Delta,Listofd,Listofb,"feasible",Ease,
CES),table),
        fail.

    split(List,CList,TopList,TopCList,BotList,BotCList):-           % rule 22
        split(List,CList,TopList,TopCList,BotList,BotCList).

/************************************************************
 *              Heuristic C1 Utilities                      *
 ************************************************************/

    check_sloppy_splits(_,[],_,_):- !.

    check_sloppy_splits(List,[Head|Tail],CList,Prod):-
        Head = Prod,
        check_sloppy_splits(List,Tail,CList,Prod),!.

    check_sloppy_splits(List,[Prod1,Prod2|Tail],CList,Prod):-
        Prod1 <> Prod,
        Prod2 <> Prod,
        split_above(List,Prod1,Top,Bot),
        remove_split_from_choice(List,CList,CLIst,Top,Bot,Prod),
        check_sloppy_splits(List,[Prod2|Tail],CList,Prod),!.

    check_sloppy_splits(List,[Prod1,Prod2|Tail],CList,Prod):-
        Prod1 <> Prod,
        Prod2 =  Prod,
        check_sloppy_splits(List,[Prod2|Tail],CList,Prod),!.

    check_sloppy_splits(List,[Prod1|Tail],CList,Prod):-
        Prod1 <> Prod,
          Tail = [],
          check_sloppy_splits(List,Tail,CList,Prod),!.

     remove_split_from_choice(_,_,[],_,_,_):- !.

     remove_split_from_choice(List,CLIst,[Head|Tail],Top,Bot,Prod):-
          dsst(List,CList,Type,Top,Bot,Head,HK,Delta,Listofd,Listofb,Feasibility,Ease,CES),
          write("\n\nMost plentiful product is ",Prod,". The",Type),
          write("\nsplit ",Top," / ",Bot," with ",Head,"/",HK,"as LK/HK"),
          write("\nviolates heuristic C1: remove the most plentiful"),
          write("\nproduct first. Do you wish to remove this"),
          write("\nsplit from consideration or maintain it?"),
          menu(13,55,14,14,["Remove","Maintain"],"Answer",2,RChoicesplit),
          choose_split(RChoicesplit),
          retract(dsst(List,CList,Type,Top,Bot,Head,HK,Delta,Listofd,Listofb,Feasibility,Ease,
CES),table),
          assertz(ddsst(List,Clist,Type,Top,Bot,Head,HK,Delta,Listofd,Listofb,Feasibility,Ease,
CES),table),
          remove_split_from_choice(List,CLIst,Tail,Top,Bot,Prod),!.

    remove_split_from_choice(List,CList,[_|Tail],Top,Bot,Prod):-
          remove_split_from_choice(List,CList,Tail,Top,Bot,Prod),!.

     check_sharp_splits(_,[_],_,_):- !.

     check_sharp_splits(List,[Compl|Tail],CList,Prod):-
          flow(Prod,Compl,CompFlow),
          CompFlow = 0,
          split_below(CList,Compl,TopComp,_),
          sum_component_flow(TopComp,TopCompFlow,[Prod]),
          TopCompFlow = 0,
          check_sharp_splits(List,Tail,CList,Prod),!.
     check_sharp_splits(List,[Compl|Tail],CList,Prod):-
          flow(Prod,Compl,CompFlow),
          CompFlow = 0,
          split_above(CList,Compl,_,BotComp),
          sum_component_flow(BotComp,BotCompFlow,[Prod]),
          BotCompFlow = 0,
          check_sharp_splits(List,Tail,CList,Prod),!.

     check_sharp_splits(List,[Compl|Tail],CList,Prod):-
```

```
        flow(Prod,Compl,CompFlow),
        CompFlow = 0,
        split_above(CList,Compl,TempTopComp,BotComp),
        sum_component_flow(BotComp,BotCompFlow,[Prod]),
        BotCompFlow <> 0,
        delete_element(Compl,TempTopComp,TopComp),
        sum_component_flow(TopComp,TopCompFlow,[Prod]),
        TopCompFlow <> 0,
        dsst(List,CList,"sharp",Top,Bot,Compl,HK,Delta,Listofd,Listofb,Feasibility,Ease,CES),
        write("\n\nMost plentiful product is ",Prod,". The sharp"),
        write("\nsplit ",Top," / ",Bot," with ",Compl,"/",HK,"as LK/HK"),
        write("\nviolates heuristic C1: remove the most plentiful"),
        write("\nproduct first. Do you wish to remove this"),
        write("\nsplit from consideration or maintain it?"),
        menu(13,55,14,14,["Remove","Maintain"],"Answer",2,RChoicesplit),
        choose_split(RChoicesplit),
        retract(dsst(List,CList,"sharp",Top,Bot,Compl,HK,Delta,Listofd,Listofb,Feasibility,Ease,
CES),table),
        assertz(ddsst(List,CList,"sharp",Top,Bot,Compl,HK,Delta,Listofd,Listofb,Feasibility,Ease,
CES),table),
        check_sharp_splits(List,Tail,CList,Prod),!.

    check_sharp_splits(List,[Compl|Tail],CList,Prod):-
        flow(Prod,Compl,CompFlow),
        CompFlow <> 0,
        split_above(CList,Compl,_,BotComp),
        sum_component_flow(BotComp,BotCompFlow,[Prod]),
        BotCompFlow = 0,
        check_sharp_splits(List,Tail,CList,Prod),!.

    check_sharp_splits(List,[Compl|Tail],CList,Prod):-
        flow(Prod,Compl,CompFlow),
        CompFlow <> 0,
        split_above(CList,Compl,_,BotComp),
        sum_component_flow(BotComp,BotCompFlow,[Prod]),
        BotCompFlow <> 0,
        dsst(List,CList,"sharp",Top,Bot,Compl,HK,Delta,Listofd,Listofb,Feasibility,Ease,CES),
        write("\n\nMost plentiful product is ",Prod,". The sharp"),
        write("\nsplit ",Top," / ",Bot," with ",Compl,"/",HK,"as LK/HK"),
        write("\nviolates heuristic C1: remove the most plentiful"),
        write("\nproduct first. Do you wish to remove this"),
          write("\nsplit from consideration or maintain it?"),
          menu(13,55,14,14,["Remove","Maintain"],"Answer",2,RChoicesplit),
          choose_split(RChoicesplit),
          retract(dsst(List,CList,"sharp",Top,Bot,Compl,HK,Delta,Listofd,Listofb,Feasibility,Ease,
CES),table),
          assertz(ddsst(List,CList,"sharp",Top,Bot,Compl,HK,Delta,Listofd,Listofb,Feasibility,Ease,
CES),table),
          check_sharp_splits(List,Tail,CList,Prod),!.

     check_sharp_splits(List,[_|Tail],CList,Prod):-
          check_sharp_splits(List,Tail,CList,Prod),!.



/*****************************************************************
 *   Heurstic C2 Utilities                                       *
 *****************************************************************/


     determine_two_best_split_ratios(List,CList,Top1,Bot1,Type1,LK1,HK1,Ratio1,CES1,Top2,Bot2,
Type2,LK2,HK2,Ratio2,CES2):-
        determine_best_split_ratio(List,CList,[],[],"none","none","none",100,0,Top1,Bot1,Type1,
LK1,HK1,Ratio1,CES1,"yes"),
        adjust_database(List,CList,Type1,Top1,Bot1,LK1,HK1),
        determine_best_split_ratio(List,CList,[],[],"none","none","none",100,0,Top2,Bot2,Type2,
LK2,HK2,Ratio2,CES2,"no"),
        readjust_database(List,CList), !.

     adjust_database(List,CList,_,_,_,_,_):-
           dsst(List,CList,Type,Top,Bot,LK,HK,Delta,Listofd,Listofb,"feasible","easy",CES),
           assertz(ddddsst(List,CList,Type,Top,Bot,LK,HK,Delta,Listofd,Listofb,"feasible","easy",
CES),table),
           assertz(dddsst(List,CList,Type,Top,Bot,LK,HK,Delta,Listofd,Listofb,"feasible","easy",
CES),table),
           retract(dsst(List,CList,Type,Top,Bot,LK,HK,Delta,Listofd,Listofb,"feasible","easy",
CES),table),
           fail.

     adjust_database(List,CList,_,_,_,_,_):-
             dddsst(List,CList,Type,Top,Bot,LK,HK,Delta,Listofd,Listofb,"feasible","easy",CES),
             assertz(dsst(List,CList,Type,Top,Bot,LK,HK,Delta,Listofd,Listofb,"feasible","easy",
CES),table),
             retract(dddsst(List,CList,Type,Top,Bot,LK,HK,Delta,Listofd,Listofb,"feasible","easy",
CES),table),
             fail.

     adjust_database(List,CList,Type,Top,Bot,LK,HK):-
             dsst(List,CList,Type,Top,Bot,LK,HK,Delta,Listofd,Listofb,"feasible","easy",CES),
             retract(dsst(List,CList,Type,Top,Bot,LK,HK,Delta,Listofd,Listofb,"feasible","easy",
CES),table),!.

     readjust_database(List,CList):-
             dsst(List,CList,Type,Top,Bot,LK,HK,Delta,Listofd,Listofb,"feasible","easy",CES),
             retract(dsst(List,CList,Type,Top,Bot,LK,HK,Delta,Listofd,Listofb,"feasible","easy",
CES),table),
        fail.

     readjust_database(List,CList):-
             ddddsst(List,CList,Type,Top,Bot,LK,HK,Delta,Listofd,Listofb,"feasible","easy",CES),
             assertz(dsst(List,CList,Type,Top,Bot,LK,HK,Delta,Listofd,Listofb,"feasible","easy",
CES),table),
             retract(ddddsst(List,CList,Type,Top,Bot,LK,HK,Delta,Listofd,Listofb,"feasible","easy",
CES),table),
             fail.

     readjust_database(_,_):- !.

     determine_whether_to_use_CES(Testratio,Ratio2,"no"):-
          Testratio < Ratio2, !.

     determine_whether_to_use_CES(Testratio,Ratio2,"yes"):-
          Testratio >= Ratio2, !.

     determine_best_split_ratio(List,CList,_,_,_,_,_,TempRatio,_,Top,Bot,Type,LK,HK,Ratio
,CES,Writeoption):-
          get_split(List,CList,TestType,TestTop,TestBot,TestLK,TestHK,TestCES),
          resulting_products(List,TestType,TestTop,TestBot,TestTopList,TestTopCList,TestBotList,
TestBotCList),
          get_split_flow(TestTopList,TestTFLow,TestTopCList,0),
          get_split_flow(TestBotList,TestBFlow,TestBotCList,0),
          solve_ratio(TestTFLow,TestBFlow,Ratio1),
          solve_ratio(TestBFLow,TestTFlow,Ratio2),
          max(Ratio1,Ratio2,Testratio),
          line_write(TestTop,TestBot,TestLK,TestHK,TestTFLow,TestBFlow,Ratio1,TestCES,Writeoption),
          Testratio < TempRatio,
          determine_best_split_ratio(List,CList,TestTop,TestBot,TestType,TestLK,TestHK,TestRatio,
TestCES,Top,Bot,Type,LK,HK,Ratio,CES,Writeoption).

     determine_best_split_ratio(List,CList,TestTop,TestBot,TestType,TestLK,TestHK,TestRatio,TestCES,
Top,Bot,Type,LK,HK,Ratio,CES,Writeoption):-
          dsst(List,CList,_,_,_,_,_,_,_,_,"feasible","easy",_),
          determine_best_split_ratio(List,CList,TestTop,TestBot,TestType,TestLK,TestHK,TestRatio,
TestCES,Top,Bot,Type,LK,HK,Ratio,CES,Writeoption).

     determine_best_split_ratio(List,CList,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_):-
        dddddsst(List,CList,Type,Top,Bot,LK,HK,Delta,Listofd,Listofb,Feasibility,Ease,CES),
        retract(dddddsst(List,CList,Type,Top,Bot,LK,HK,Delta,Listofd,Listofb,Feasibility,Ease,
CES),table),
        assertz(dsst(List,CList,Type,Top,Bot,LK,HK,Delta,Listofd,Listofb,Feasibility,Ease,
CES),table),
           fail.

    determine_best_split_ratio(_,_,Top,Bot,Type,LK,HK,Ratio,CES,Top,Bot,Type,LK,HK,Ratio,CES,_):- !.

    line_write(Top,Bot,LK,HK,TFLow,BFLow,Ratio,CES,"yes"):-
          shiftwindow(N),
          X = 100/(1 + Ratio),
          Y = 100 - X,
          max(X,Y,Z1),
          minimum(X,Y,Z2),
          k_value(LK,KLK),
          k_value(HK,KHK),
          Alpha = KLK/KHK,
          nl,write(Top,"/",Bot),nl,
          write("                    ",LK,"    ",HK),
          writef("%9.2",Alpha),
          writef("%10.1",TFLow),
          writef("%11.1",BFLow),
          writef("%7.0",Z1),write("/"),writef("%2.0",Z2),
          writef("%8.2",CES),
          pause(N), !.

     line_write(_,_,_,_,_,_,_,_,"no"):- !.

     get_split(List,CList,Type,Top,Bot,LK,HK,CES):-
```

```
          dsst(List,CList,Type,Top,Bot,LK,HK,Delta,Listofd,Listofb,"feasible","easy",CES),
          retract(dsst(List,CList,Type,Top,Bot,LK,HK,Delta,Listofd,Listofb,"feasible","easy",
CES),table),
          assertz(dddddsst(List,CList,Type,Top,Bot,LK,HK,Delta,Listofd,Listofb,"feasible","easy",
CES),table), !.

     decide_split_choice(Top,Bot,Type,LK,HK,Ratio,CES,_,_,_,_,_,_,Top,Bot,Type,LK,HK,Ratio,CES,
"no"):- !.

     decide_split_choice(Top,Bot,Type,LK,HK,Ratio,CES1,_,_,_,_,_,_,CES2,Top,Bot,Type,LK,HK,Ratio,
CES1,"yes"):-
       CES1 >= CES2, !.

     decide_split_choice(_,_,_,_,_,_,CES1,Top,Bot,Type,LK,HK,Ratio,CES2,Top,Bot,Type,LK,HK,Ratio,
CES2,"yes"):-
       CES1 < CES2, !.

     resulting_products(_,Type,Top,Bot,TopList,TopCList,BotList,BotCList):-
          Type = "sloppy",
          TopList = Top,
          BotList = Bot,
          positive_component_flow_list(TopList,TopCList),
          positive_component_flow_list(BotList,BotCList).

     resulting_products(List,Type,Top,Bot,TopList,TopCList,BotList,BotCList):-
          Type = "sharp",
          TopCList = Top,
          BotCList = Bot,
          determine_sharp_products(List,TopCList,[],TopList),
          determine_sharp_products(List,BotCList,[],BotList).
          determine_sharp_products([],_,R,R).

     determine_sharp_products([Head|Tail],CList,RefList,TorBList):-
          product_flow(Head,PFlow,CList),
          PFlow > 0,
          append(RefList,[Head],NewRefList),
          determine_sharp_products(Tail,CList,NewRefList,TorBList).

     determine_sharp_products([Head|Tail],CList,RefList,TorBList):-
          product_flow(Head,PFlow,CList),
          PFlow <= 0,
          determine_sharp_products(Tail,CList,RefList,TorBList).

/* End of Heuristic C2 Utilities  */

     decide_split_path(1):- fail.
     decide_split_path(2):-nl,!.

     choose_split(2):- fail.
     choose_split(1):-nl,!.

     max_ces_sloppy(_,[_],_,TempCES,TempCES,_):-!.
     max_ces_sloppy(List,[Head|Tail],CList,TempCES,CESsloppy,Ease):-
            split_above(List,Head,Top,Bot),
            search_splits_for_max(List,CList,CList,Top,Bot,TempCES,TestCES,Ease),
            max_ces_sloppy(List,Tail,Clist,TestCES,CESsloppy,Ease),!.
     max_ces_sloppy(List,[_|Tail],CList,TempCES,CESsloppy,Ease):-
            max_ces_sloppy(List,Tail,Clist,TempCES,CESsloppy,Ease),!.

     search_splits_for_max(_,_,[],_,_,CES,CES,_):- !.
     search_splits_for_max(List,CList,[Head|Tail],Top,Bot,TempCES,CESsloppy,Ease):-
          dsst(List,CList,_,Top,Bot,Head,_,_,_,_,"feasible",Ease,TestCES),
          TestCES > TempCES,
          search_splits_for_max(List,CList,Tail,Top,Bot,TestCES,CESsloppy,Ease).
     search_splits_for_max(List,CList,[Head|Tail],Top,Bot,TempCES,CESsloppy,Ease):-
          dsst(List,CList,_,Top,Bot,Head,_,_,_,_,"feasible",Ease,TestCES),
          TestCES <= TempCES,
          search_splits_for_max(List,CList,Tail,Top,Bot,TempCES,CESsloppy,Ease).

     max_ces_sharp(_,_,[],CES,CES,_):- !.

     max_ces_sharp(List,CList,[Head|Tail],TempCES,CESsharp,Ease):-
          split_above(List,Head,TopComp,BotComp),
          dsst(List,CList,"sharp",TopComp,BotComp,_,_,_,_,_,"feasible",Ease,TestCES),
          TestCES > TempCES,
          max_ces_sharp(List,CList,Tail,TestCES,CESsharp,Ease).

     max_ces_sharp(List,CList,[Head|Tail],TempCES,CESsharp,Ease):-
          split_above(List,Head,TopComp,BotComp),
          dsst(List,CList,"sharp",TopComp,BotComp,_,_,_,_,_,"feasible",Ease,TestCES),
          TestCES <= TempCES,
          max_ces_sharp(List,CList,Tail,TempCES,CESsharp,Ease).
max_ces_sharp(List,CList,[_|Tail],TempCES,CESsharp,Ease):-
     max_ces_sharp(List,CList,Tail,TempCES,CESsharp,Ease).

solve_ratio(F1,F2,R):-
     F1 > 0.0,
     F2 > 0.0,
     R = F1/F2,!.

solve_ratio(F1,F2,R):-
     F2 = 0.0,
     F1 > 0.0,
     R = 49.0,!.



solve_ratio(F1,F2,R):-
     F1 = 0.0,
     F2 > 0.0,
     R = 0.02040816327,!.

minimum(X,Y,X):- X < Y, !.
minimum(X,Y,Y):- X >= Y, !.
```

UTILITY MODULE

```
/****************************************************************************************************
     The central purpose of the UTILITY MODULE is to support the rest of the program with
     frequently-used list and numerical processing "utility" relations. By locating all the
     utility predicates in this "library", we avoid the needless duplication of relations.
****************************************************************************************************/

project "exsep"                                                    % Required for Turbo Prolog only

global domains                                                     % Required for Turbo Prolog only
     alpha = symbol
     klist = alpha*
     number = integer
     value = real
     vlist = value*
     str = string

/******************************************************
 *    Global domains for Toolbox Predicates           *
 ******************************************************/
  ROW, COL, LEN, ATTR   = INTEGER
  STRINGLIST = STRING*
  INTEGERLIST = INTEGER*
  KEY    = cr; esc; break; tab; btab; del; bdel; ctrlbdel; ins;
        end ; home ; fkey(INTEGER) ; up ; down ; left ; right ;
        ctrlleft; ctrlright; ctrlend; ctrlhome; pgup; pgdn;
        ctrlpgup; ctrlpgdn; char(CHAR) ; otherspec


global database - bypass                                           % Required for Turbo Prolog only
     dbypass_amount(klist,alpha,value)
     dbypass_result(klist,alpha)
     dypass_status(alpha)

global database - materials                                        % Required for Turbo Prolog only
     flow(alpha,alpha,value)
     k_value(alpha,value)
     boiling_temp(alpha,value)
     initial_components(klist)
```

```
    initial_set(klist)
    corrosive(alpha)
    pseudoproduct(klist,alpha)

global database - table                                    % Required for Turbo Prolog only
    dsst(klist,klist,alpha,klist,klist,alpha,alpha,value,vlist,vlist,alpha,alpha,value)
    ddsst(klist,klist,alpha,klist,klist,alpha,alpha,value,vlist,vlist,alpha,alpha,value)
    dddsst(klist,klist,alpha,klist,klist,alpha,alpha,value,vlist,vlist,alpha,alpha,value)
    ddddsst(klist,klist,alpha,klist,klist,alpha,alpha,value,vlist,vlist,alpha,alpha,value)
    dddddsst(klist,klist,alpha,klist,klist,alpha,alpha,value,vlist,vlist,alpha,alpha,value)
    process_keys(alpha,alpha)
    process_product(alpha)
    sloppy_keys(alpha,alpha)


global database - sequence                                 % Required for Turbo Prolog only
    dseparator(number,klist,vlist,klist,vlist)
    dstreamin(number,klist,vlist)
    dstreambypass(number,alpha,klist,vlist)

global predicates                                          % Required for Turbo Prolog only
    append_v(vlist,vlist,vlist) - (i,i,o)
    bypass(klist,klist) - (i,i)
    component_flow(klist,alpha,value) - (i,i,o)
    component_flow_list(klist,vlist,klist,klist) - (i,i,i,o)
    component_flow_set(klist,klist,vlist) - (i,i,o)
    decide_difficult_or_easy(value,alpha) - (i,o)
    delete_element(alpha,klist,klist) - (i,i,o)
    delete_elements(klist,klist,klist) - (i,i,o)
    delete_number(value,vlist,vlist) - (i,i,o)
    equal(klist,klist) - (i,i)
    ffactor(value,value,value) - (i,i,o)
    first_prod(klist,klist) - (i,o)
    flow_set(alpha,klist,vlist) - (i,i,o)
    flow_set_c(alpha,klist,vlist) - (i,i,o)
    get_split_flow(klist,value,klist,value) - (i,o,i,i)
    largest_flow(klist,alpha,klist) - (i,o,i)
    last_prod(klist,klist) - (i,o)
    length(klist,value) - (i,o)
    list_member(klist,klist) - (i,i)
    max(value,value,value) - (i,i,o)
    max(number,number,number) - (i,i,o)
    max_flist(vlist,value) - (i,o)
    max_flow(value,value,value) - (i,i,o)
    member(alpha,klist) - (i,i)
    pause(number) - (i)
    pause_escape
    positive_component_flow_list(klist,klist) - (i,o)
    product_flow(alpha,value,klist) - (i,o,i)
    product_flow_match(klist,value,alpha,klist) - (i,i,o,i)
    product_flow_set(klist,vlist,klist) - (i,o,i)
    reverse(klist,klist) - (i,o)
    selected_positive_component_flow_list(klist,klist,klist) - (i,i,o)
    separation_specification_table(klist,klist) - (i,i)
    set_above(klist,alpha,klist) - (i,i,o)
    set_below(klist,alpha,klist) - (i,i,o)
    split(klist,klist,klist,klist,klist,klist) - (i,i,o,o,o,o)
    split_above(klist,alpha,klist,klist) - (i,i,o,o)
    split_below(klist,alpha,klist,klist) - (i,i,o,o)
    sub_list(klist,klist) - (i,i)
    sum_component_flow(klist,value,klist) - (i,o,i)
    sum_flist(vlist,value) - (i,o)
    two_highest_flows(vlist,value,value) - (i,o,o)

predicates                                                 % Required for Turbo Prolog only
    append(klist,klist,klist)
    build_list(klist,klist,klist,klist)
    product_flow_help(alpha,value,klist,value)

/*****************************************************************************************
    All computer code above this point is unique to Turbo Prolog and is required to get it to
    run. Below this point, we begin the actual Prolog relations in the UTILITY MODULE. The
    relations below will run on almost any Prolog system.
*****************************************************************************************/

clauses

    max(A,B,A):- A > B,!.
    max(A,B,B):- A <= B,!.

    first_prod([],[]).
    first_prod([X|_],[X]).
```

```
last_prod([],[]).
last_prod([Head|Tail],[Prod]):-
    Tail = [],!,
    Prod = Head.
last_prod([_|Tail],X):-
    last_prod(Tail,X).

pause(N):-
    shiftwindow(7),
    write("\nPause. Press enter to continue."),nl,
    readln(_),
    clearwindow,
    shiftwindow(N),!.

pause_escape:-
    nl,
    write("Pause. Press enter to procees, or ESC to"),nl,
    write("consider other alternatives"),nl,
    readln(_).

ffactor(Top,Bot,Ratio):-
    Top < Bot,!,
    Ratio = Top/Bot.
ffactor(Top,Bot,Ratio):-
    Top >= Bot,!,
    Ratio = Bot/Top.

split_above(List,Prod,SetAboveIncludingProd,SetBelowProd):-
    set_above(List,Prod,SetAboveProd),
    set_below(List,Prod,SetBelowProd),
    append(SetAboveProd,[Prod],SetAboveIncludingProd).

split_below(List,Prod,SetAboveProd,[Prod|SetBelowProd]):-
    set_above(List,Prod,SetAboveProd),
    set_below(List,Prod,SetBelowProd).

set_above(List,Prod,ListAbove):-
    append(ListAbove,[Prod|_],List),!.

set_below(List,Prod,ListBelow):-
    set_above(List,Prod,ListAbove),
     append(ListAbove,[Prod|ListBelow],List),!.

largest_flow(List,Lprod,CList):-
     product_flow_set(List,FlowList,CList),
     max_flist(FlowList,Lflow),
     product_flow_match(List,Lflow,Lprod,CList).

two_highest_flows(List,MFlow,SFlow):-
     max_flist(List,MFlow),
     delete_number(MFlow,List,NList),
     max_flist(NList,SFlow),!.

product_flow(Head,PFlow,CList):-
     flow_set(Head,CList,PFlowList),
     sum_flist(PFlowList,PFlow).

product_flow_help(_,PFlow,[],PFlow):-!.
product_flow_help(Prod,PFlow,[HeadC|TailC],RefFlow):-
     flow(Prod,HeadC,NFlow),!,
     NRefFlow = RefFlow + NFlow,
     product_flow_help(Prod,PFlow,TailC,NRefFlow).

product_flow_match([Prod|_],PFlow,Prod,CList):-
     product_flow(Prod,Flow,CList),
     Flow = PFlow,!.
product_flow_match([_|Tail],PFlow,Prod,CList):-
     product_flow_match(Tail,PFlow,Prod,CList).

product_flow_set([],[],_).
product_flow_set([Head|Tail],[HeadFlow|TailFlow],CList):-
     product_flow(Head,HeadFlow,CList),
     product_flow_set(Tail,TailFlow,CList).

flow_set(_,[],[]).
flow_set(Prod,[HeadProd|TailProd],[HeadFlow|TailFlow]):-
     flow(Prod,Headprod,HeadFlow),!,
     flow_set(Prod,TailProd,TailFlow).

append_v([],List,List).
append_v([Head|Tail],List1,[Head|List]):-
     append_v(Tail,List1,List).
```

```
max_flist([X],X).
max_flist([Int1,Int2|Tail],Max):-
     max_flist([Int2|Tail],MaxTail),
     max_flow(Int1,Maxtail,Max).

max_flow(F1,F2,F1):-
     F1 >= F2,!.
max_flow(F1,F2,F2):-
     F1 < F2,!.

sum_flist([],0).
sum_flist([Head|Tail],Sum):-
     sum_flist(Tail,Sumtail),
     Sum = Head + Sumtail.

append([],List,List).
append([Head|Tail],List1,[Head|List]):-
     append(Tail,List1,List).

reverse([],[]).
reverse([Head|Tail],List):-
     reverse(Tail,List1),
     append(List1,[Head],List).

length([],0).
length([_|Tail],Len):-
     length(Tail,Len1),
     Len = Len1 + 1.

equal(X,X).

sub_list(Sublist,List):-
     append(_,List1,List),
     append(Sublist,_,List1),!.

positive_component_flow_list(List,CList):-
     initial_components(Components),!,
     component_flow_set(List,Components,FlowList),
     component_flow_list(Components,FlowList,[],CList).

selected_positive_component_flow_list(List,CList,CompList):-
```

```
     initial_components(Components),!,
     component_flow_set(List,Components,FlowList),
     component_flow_list(Components,FlowList,[],RefCompList),
           build_list(RefCompList,CList,[],CompList),!.

build_list([],_,C,C):- !.
build_list([Head|Tail],CList,RefBuildList,CompList):-
     member(Head,CList),
     append(RefBuildList,[Head],NewRefBuildList),
     build_list(Tail,CList,NewRefBuildList,CompList),!.
build_list([Head|Tail],CList,RefBuildList,CompList):-
     not(member(Head,CList)),
     build_list(Tail,CList,RefBuildList,CompList),!.

component_flow_list([],[],CList,CList).
component_flow_list([HR|TR],[HF|TF],TempCList,CList):-
     HF > 0,
     append(TempCList,[HR],NewCList),!,
     component_flow_list(TR,TF,NewCList,CList).
component_flow_list([_|TR],[HF|TF],TempCList,CList):-
     HF <= 0,!,
     component_flow_list(TR,TF,TempCList,CList).

component_flow_set(_,[],[]).
component_flow_set(List,[Head|Tail],[HeadFlow|TailFlow]):-
     component_flow(List,Head,HeadFlow),
     component_flow_set(List,Tail,TailFlow).

get_split_flow([],TFlow,_,TFlow):-!.
get_split_flow([Head|Tail],TFlow,CList,RefFlow):-
     !,product_flow_help(Head,PFlow,CList,0),
     NRefFlow = RefFlow + PFlow,
     get_split_flow(Tail,TFlow,CList,NRefFlow).

component_flow(List,Head,CFlow):-
     flow_set_c(Head,List,CFlowList),
     sum_flist(CFlowList,CFlow).

flow_set_c(_,[],[]).
flow_set_c(Comp,[HeadProd|TailProd],[HeadFlow|TailFlow]):-
     flow(HeadProd,Comp,HeadFlow),!,
        flow_set_c(Comp,TailProd,TailFlow).

    delete_elements([],Y,Y).
    delete_elements([Head|Tail],X,Y):-
        delete_element(Head,X,Z),
        delete_elements(Tail,Z,Y).

    delete_element(X,[X|Tail],Tail):-!.
    delete_element(X,[Y|Tail],[Y|Tail1]):-
        delete_element(X,Tail,Tail1),!.

    delete_number(X,[X|Tail],Tail):-!.
    delete_number(X,[Y|Tail],[Y|Tail1]):-
        delete_number(X,Tail,Tail1),!.

    sum_component_flow([],SumCompFlow,_):-
        SumCompFlow = 0.
    sum_component_flow([Head|Tail],SumCompFlow,List):-
        component_flow(List,Head,HeadFlow),
        sum_component_flow(Tail,RunningSum,List),
        SumCompFlow = RunningSum + HeadFlow.

    list_member([],_):-!.
    list_member([Head|Tail],List):-
        member(Head,List),
        list_member(Tail,List).

    member(X,[X|_]):- !.
    member(X,[_|Tail]):-
        member(X,Tail).

    decide_difficult_or_easy(Delta,Ease):-
        Delta > 10.0,!,
        Ease = "easy".
    decide_difficult_or_easy(Delta,Ease):-
        Delta <= 10.0,!,
        Ease = "difficult".
```

APPENDIX C

GLOSSARY OF TERMS

Abort: stop further execution of the program.

Adaptive Control: a form of process control that attempts to adjust controller tuning parameters in response to process dynamics.

AI: artificial intelligence.

Algorithm: a formal procedure that specifies a step-by-step execution path such that a correct or optimal answer results at a predefined and fixed stopping points. Importantly, if the algorithm is followed exactly, the result is guaranteed.

And: a logical conjunction of goals; both goals must be satisfied for the entire expression to be true. For example, if we write in logic:

B *AND* C → A, then both B and C must be true for A to be true.

AND-OR graphs: a way of solving problems by systematic problem-decomposition into a set of procedures that: 1) all must be done together for success (an AND node), or 2) can be done completely independent of each other for success (an OR node). AND-OR graphs identify the points of mutual dependence and mutual exclusivity in a problem, and subsequently decompose the problem based on those points. See section 11.3A.

AND-OR Problem Decomposition: problem-solving using AND-OR graphs.

ANN: artificial neural network.

Anonymous Variable: denoted by the underscore character, _. It is used in place of an ordinary variable when both the name and instantiation of that variable appear only once in the clause, or are inconsequential. For example, the Prolog statement **member(X,[X|_])**. This says that **X** is a member of a list if **X** is the head of that list. The tail of the list is inconsequential, and accordingly, is represented by an anonymous variable. Anonymous variables improve the readability of the program, and two or more anonymous variables in the same clause are treated as *totally separate variables*.

Answer: output from Prolog after execution. If Prolog answers ***yes***, the goal is successful and Prolog will report variable instantiations (if any) that make the goal successful. If Prolog answers ***no*** the goal could not be successfully achieved.

Argument: a name for any Prolog data object that is present in a *structure*; arguments are typically separated by commas and appear between the parenthesis of the structure, e.g., **functor(** *argument_1*, *argument_2*, *argument_3*, ...). If the structure utilizes an operator, the arguments typically do not appear in parenthesis.

Arithmetic Operator: an operator that defines an arithmetic expression. Arithmetic operators generally take no action on their own. Instead, they define the arithmetic expression that will allow built-in Prolog *evaluation predicates* (e.g., is, **=:=**) to perform the calculation.

Arithmetic Procedure: a *built-in* Prolog procedure that performs numerical calculation and evaluation of its arguments (e.g., is, =:=).

Arity: The number of *arguments* possessed by a *functor*, e.g., in the statement **member(X,[X|_])**, the **member** functor has an arity of two.

Artificial Intelligence: the part of computer science concerned with designing intelligent computer systems, that is, systems that exhibit characteristics we associate with intelligence in human behavior. In practical terms, artificial intelligence deals with symbolic, non-algorithmic methods of problem-solving.

Artificial Neural Network: also known as ANN, a quantitative empirical modeling tool characterized by a network of highly interconnected nodes that pass numerical values to each other and calculate an output based on the sum of inputs from other nodes. ANNs are particularly useful for pattern-matching and filtering noisy or incomplete information.

Atom: a constant that cannot be broken down into other objects and is inseparable in the program. In Prolog, an atom refers more specifically to a *nonnumerical* constant. The empty list, [], is an atom.

Atomic: an *atomic* data type is either a numerical or nonnumerical Prolog constant; an atomic data type cannot be broken down into other objects and is inseparable in the program. In Prolog, the atomic data type is not to be confused with an atom. The atomic data type can be numerical or nonnumerical; an atom is nonnumerical *only*.

Automated Abstraction: the ability of an AI technique to automatically ascertain the details and interactions of relationships without the need of a domain expert and/or knowledge engineer encoding the knowledge. ANNs have the property of automated abstraction.

Backpropagation Learning: a type of supervised, error-correction learning in ANNs where an error on the output layer, $\underline{\epsilon}$, is calculated, and that error is propagated backwards through the network to determine how each individual connection weight contributed to the output error. Based on each connection weight's contribution to the error, that weight is adjusted to minimize the total output error.

Backtracking: an execution step taken by Prolog when the current goal fails. All variables that were instantiated within the current goal become free variables again. Prolog then "backs up" the last previously satisfied goal, and makes it the current goal, in an attempt to satisfy the relation in some other way.

Backward Chaining: a reasoning mechanism that treats the conclusion of a rule as a goal and attempts to satisfy the goal by proving that the premises (sub-goals) are true. Backward chaining is *goal-driven.* It relies on THEN-IF-type rules; that is, THEN a certain goal is true, IF certain sub-goals are proven true. Backward chaining is a "top-down" inference mechanism that tries to *instantiate* variables through matching with facts. Prolog's built-in reasoning mechanism uses backward chaining.

Best-first Search: a search strategy that maintains an exhaustive set of candidate pathways for the search to expand to, computes a *heuristic estimate* for each candidate, and expands to the best candidate according to the estimate.

Bias Function: internal threshold values that add a fixed amount to the nodal summation. Typically, a bias function adds 1 to the nodal summation. Bias functions are in contrast with more typical internal threshold, where we *subtract* the threshold value from the nodal summation.

Binding Strength: a property of operators that indicates which operator is the principal functor. Operators with the *lower* binding strengths are principal functors. Thus the statement **2 + 3 * 4** is interpreted in Prolog as **+ (2 , * (3 , 4))**. Multiplication has higher binding strength than addition because it is the operation performed first when the statement is numerically evaluated.

Blackboard: an intentional combination of two or more knowledge representations into one single, operating system. A Blackboard attempts to effectively integrate the strengths of multiple knowledge representations, and perform opportunistic reasoning with a specific knowledge representation when the appropriate opportunity arises.

Blind Search: a search strategy that arbitrarily or randomly chooses the next expansion node with no heuristic guidance or systematic pattern.

Body of a Clause: the set of sub-goals, or conditions, following the :- operator in a Prolog clause, that must be fulfilled for the entire clause to be true. A Prolog clause can be viewed as:

conclusion :- conditions.

head :- body.

The body of a clause is empty for Prolog facts.

Boolean Model: a model where variables can take on only one of two values: 0 or 1. Usually, zero represents a logical *no*, or *false*. The one represents a logical *yes*, or *true*.

Box Representation: a representation of a Prolog goal as a box with four ports: two input ports (CALL and REDO) and two exit ports (EXIT and FAIL). This representation is useful for tracing program execution.

Breadth-First Search: a search strategy that maintains an exhaustive set of candidate pathways for the search to expand to, and expands to all states closest to the current state before going any deeper into the search space. Instead of favoring rapid penetration in to the search space, a breadth-first search expands equally to all immediate successor nodes before going deeper.

Brittleness: the undesirable property of a system, characterized by being catastrophically unable to handle situations outside the narrow scope of the system's expertise. For example, if the system is given inconsistent, erroneous, or incomplete data, a brittle system written in Prolog may *fail* and answer abruptly with a ***no***. See also robustness.

Built-in Operator: an operator with a predefined meaning that is automatically recognized by the Prolog program; no declarations are required of the user.

Built-in Predicate: an atom with a predefined meaning that is automatically recognized by the Prolog program; no declarations are required of the user. The *built-in predicate* is the atom that calls a built-in procedure.

Built-in Procedure: a procedure with a predefined meaning that is automatically recognized by the Prolog program; no declarations are required of the user.

C: a very efficient, general purpose, low-level language computing language typically associated with the UNIX operating system. C has traditionally been used for system programming, but with growing use, is now been used for much more diverse tasks.

CAD: computer-aided design, i.e., the use of computer tools to aid engineering design. CAD is used for diverse tasks in engineering; electrical engineers use CAD to design integrated circuits; mechanical engineers use CAD to design mechanical parts; and chemical engineers use CAD to design chemical process flowsheets.

Call: a call for satisfying a goal. The clause called becomes the current goal. The "call" is also a built-in procedure, where **call(G)** attempts to satisfy goal G.

Causal Knowledge: a type of knowledge used in fault diagnosis and characterized by in-depth, information derived from models. These models are usually qualitative, but may be semi-quantitative and based on first-principles. Causal knowledge uses a *deep-knowledge* approach. See deep-knowledge, evidential knowledge, and model-based knowledge.

Certainty Factor: a number (usually between zero and one) that reflects the certainty or confidence that a rule is true. Certainty Factors (CFs) are used for inexact reasoning in expert systems.

Clause: a Prolog relation identified by a principal functor and terminated with a period. There are three types of clauses: *facts*, *rules*, and *questions*.

Code: a set Prolog statements that follow the rules of syntax, and form, either in whole or in part, the Prolog program.

Comment: text to the right of **%**, or between the symbols **/*** and ***/**, a comment explains the program and is ignored by Prolog.

Comparison Operator: an operator used to compare the values of two expressions, e.g., **X > Y**, where > is a comparison operator.

Compilation Error: an error that occurs when the program is being compiled. An example is a syntax error, such as placing two commas together: **W:- X,,Y.**

Compiler: a computer program that converts (translates) a program into executable machine code. The compiled program (machine code) is normally not human-readable.

Conclusion: the head, or left-hand, side of the Prolog clause. A Prolog clause can be viewed as *conclusion* :- *condition*. The statement to the left of the **:-** operator is the conclusion that we try to prove is true when the Prolog clause is called.

Condition: the body, or right-hand side of the Prolog clause. A Prolog clause can be viewed as *conclusion* :- *condition*. The condition section of a clause is to the right of the **:-** operator, and is the set of sub-goals that must be proven true for the conclusion to be true.

Conflict Resolution: the strategy used to resolve the problem of multiple matches in an expert system. Frequently, there may be more than one rule in the database that matches the current goal. The system uses conflict resolution to determine which of the matching rules will be used. Typically, a ***priority ordering*** is used, e.g., in Prolog, the program searches from the top of the database down, and the matching rule that is closest to the top of the database is chosen first.

Conjunction: statements connected by an *and*. For the *conjunction* of goals to be true, *all* statements linked by the conjunction must be true.

Connectionist Model: an AI model of intelligent behavior that uses artificial neural networks, i.e., a set of highly interconnected nodes that map input-output responses.

Constant: a Prolog data object that cannot change its value as the program executes. A constant can be an atom or a number.

Constraint satisfaction: a *pruning* or *filtering* technique that eliminates certain potential solutions within the state space because they do not satisfy a specific set of constraints.

Consult: to read Prolog relations from a file and add them to the database.

Control Predicate: a built-in Prolog tool used to adjust or improve the procedural nature of program execution. Examples include **!** (cut), **fail**, and **repeat**.

Conventional Language: a procedural language such as Fortran, Pascal, or C, where we tell the computer *how* to execute and write a specific set of ordered steps.

CSTR: continuous stirred-tank reactor.

Current Input Stream: the input port (source) from which an executing program currently reads data. When a program begins to execute, by default the *current input stream* is a terminal. It can be changed during execution to, for instance, a disk file.

Current Output Stream: the output port (destination) to which an executing program currently writes data. When a program begins to execute, by default the *current output stream* is a terminal. It can be changed during execution to, for instance, a disk file.

Cut: a control predicate, denoted by the exclamation mark, !, that prevents backtracking between itself and the parent goal.

Data Object: a syntactic construct used to represent information. There are two types of Prolog data objects: *simple objects* and *structured objects*. Simple objects are *constants* or *variables*. Structured objects consist of a functor and its arguments.

Database: the set of Prolog relations (facts and rules) that is currently part of the Prolog program and is used by Prolog to answer questions and satisfy goals.

Debug: to locate and correct errors (bugs) in a program.

Declarative Nature: Prolog program characteristics concerned with the *relations* between objects. The is contrasted with the *procedural nature*, which focuses on the ordered *sequence* of goals that must be proven true for the relation to be true.

Deep Knowledge: knowledge that includes the fundamental physical principles supporting and justifying macroscopic rules and heuristics. Expert systems using deep knowledge usually contain *shallow knowledge* (e.g., rules and heuristics), and when the problem is outside of the domain of shallow knowledge, the deep knowledge (e.g., models) is then used. Deep knowledge is an attempt to make systems more robust, since catastrophic failure is averted by accessing models found in deep knowledge.

Default-Reasoning: a reasoning-principle used in frame-based systems, where, if a slot is empty in a lower-level frame, then using inheritance, a value is automatically assigned to the slot by default from a higher-level frame.

Depth-first Search: a systematic strategy for searching graphs where the path is determined by first trying a node's *successor* instead of expanding to adjacent nodes of equal depth.

Deterministic: a relation is *deterministic* if it can generate only one solution. Therefore, the relation will *not* generate alternate solutions if backtracked into. Consequently, we know at the outset of the goal which solution will ultimately succeed.

Disjunction: statements connected by an *or*. For the *disjunction* of goals to be true, only *one* of the statements linked by the disjunction must be true.

Domain: the problem-area of interest, apart from the computational tools use to solve problems in that area.

Domain-dependent search: a search strategy that has been custom-developed for the problem the program is attempting to solve. A domain-dependent search is usually superior to a domain-independent search, but is problem-specific, must be developed for every new problem, and consequently, requires a higher up-front time investment in program development.

Domain expert: a person who is very knowledgeable and proficient at problem-solving in a specific problem area. Typically persons building expert systems interact with the domain expert to develop the knowledge-base for the expert system.

Domain-independent search: a search strategy that employ tools that can be applied universally to all AI problems regardless of the specific task or domain at hand. Examples include the *depth-first search*, *breadth-first search*, and *best-first search*.

Domain knowledge: knowledge about a specific problem area, e.g., separation process synthesis.

Edinburgh Prolog: a common Prolog syntax, characterized by lists with square brackets (i.e., [and]), and the principal functor followed by its arguments, where the arguments are enclosed in parenthesis and the principal functor is outside of the parenthesis (i.e., **principal_functor(** a_1, a_2, ... ,a_n)).

Efficiency: a rating of how *quickly* a program runs and how long it takes to solve a problem.

Empty List: a list with no elements, i.e., [ ]. The empty, or nil list cannot be separated into a head and tail; because it is inseparable, it is a Prolog atom.

Equality Operator: a *comparison operator* that attempts to match its arguments. There are a numerous equality operators in Prolog (i.e/, =, ==, =:=, and **is**). Some operators evaluate numerical expression; other do not. Some instantiate variables; other do not. Refer to each specific equality operator for details.

Error-Correction Learning: a form of supervised learning in ANNs where connection weights are adjusted as a direct response to the output error, with the ultimate goal of minimizing or eliminating that output error. In error-correction learning, we are typically concerned with the *total* output error based on the vector of output error, $\underline{\epsilon}$.

Evidential Knowledge: a type of knowledge used in fault diagnosis and characterized by evident, experiential information derived from observables. Evidential knowledge typically uses a *shallow-knowledge* approach. See shallow-knowledge; causal knowledge.

Execution: the sequence of actions Prolog takes using the facts and rules of the database to answer a question or satisfy a goal.

Exhaustive search: a problem-solving strategy that systematically tries *all* potential problem solutions until the final solution is actually found. An exhaustive search is a "brute force," or "scorched-earth," technique, where heuristic guidance is not used. An example of an exhaustive search is the breadth-first search.

Expert System: also called a knowledge-based system, an expert system is a computer program that uses high-quality, in-depth, *knowledge* to solve complex and advanced problems typically requiring experts.

Expression: a Prolog statement connected by at least one operator. If the operator is logical, we have a *logical expression*; if the operator is numerical, we have a *numerical expression*.

EXSEP: the <u>EX</u>pert system for <u>SEP</u>aration Synthesis (see chapters 12-15).

Fact: a Prolog statement that is always, unconditionally, true. A fact is a Prolog clause with a head only and no body.

Fail: (1) a built-in predicate that always fails and forces immediate backtracking. (2) the condition that exists when Prolog is unable to satisfy a goal. At this point, the program backtracks to the previously satisfied goal in search of a solution.

Failure: the completion of an unsuccessful attempt to satisfy a goal.

Fault Diagnosis: an application of AI concerned with troubleshooting, i.e., determining the origins of problems based on observable information, and then recommending solutions or corrections.

Feedback Connections: interlayer connections in an ANN where signals are sent in the backward direction, i.e., from a node deeper into the network (closer to the output layer) backward to nodes more shallow in the network (closer to the input layer).

Feedforward Connections: interlayer connections in an ANN where signals are sent in the forward direction, i.e., from nodes more shallow in the network (closer to the input layer) to nodes node deeper into the network (closer to the output layer).

Forward Chaining: a reasoning mechanism that establishes the premises first, and then concludes that the rule is true. Forward chaining is *data-driven*. It relies on IF-THEN-type rules; that is, IF the data are in a certain configuration, THEN certain conclusions can be drawn. Forward chaining is a "bottom-up" inference mechanism that tries to *reduce* a number of premises into a single conclusion.

Frame: a knowledge representation where a grouping of related properties are organized hierarchically. Each property fills a *slot*, and slots *inherit* properties from the frame and higher level frames. Frames can also perform *default reasoning*, where, if a slot is empty, a property can be assigned by default. In addition, frames can have attached procedures that execute if a certain set of conditions exist.

Frame-Based System: a knowledge representation that utilizes the *frame* as its primary data structure.

Free Variable: an uninstantiated or unbound variable that may take on any value through Prolog's unification (matching) mechanism.

Functor: the *name* of a structured object, which must be an atom. Syntactically, a *predicate* is a the same as a *functor*, although predicate is normally used in the context of describing relations, while functor is used in the context of describing structured objects.

Fuzzy Logic: a method for reasoning under uncertainty in rule-based systems. Fuzzy logic is used to express the absence of a sharp boundary between qualitative descriptions. To attain this goal, we use fuzzy logic to *quantify* a qualitative relation. For example, if the outside temperature is -10°C, we might say it is *cold*/0.9, or *very cold*/0.7. Thus the temperature -10°C belongs to: the *cold* qualitative description with a membership of 0.9, and the *very cold* qualitative description with a membership of 0.7.

Fuzzy Set: a data structure used for reasoning with fuzzy logic. Typically, a fuzzy set represents a *qualitative description* or *relation*. If the temperature is -10°C, we may say its *cold*/0.9 or *very cold*/0.7. If the temperature is -20°C, we may say its *cold*/0.99 or *very cold*/0.8. The qualitative descriptions of *cold* and *very cold* are represented by fuzzy sets:

cold { -20°C/0.99, -10°C/0.9}

very cold { -20°C/0.8, -10°C/0.7}

Generalization Phase: the final phase of ANN development where we feed novel input data to the network. It is hoped that, with the training the network has undergone, the output response will be proper.

Generalized Delta-Rule: also called GDR, an iterative gradient-descent backpropagation training algorithm for ANNs. The GDR possesses *momentum* and a *bias function*.

Generate-and-Test: a two-stage problem solving approach where a set of potential solutions are first generated, and then placed under further scrutiny and tested. The "generator" develops potential solutions rapidly, and the "tester" places these potential solutions under more rigorous analysis. The problem is successfully solved when a solution passes through both the generation and the testing phases. If no solutions pass the tester, we backtrack to the generator in search of alternate solutions.

Global Variable: a variable whose semantic meaning encompasses the entire program. Thus, if a global variable changes value in one part of the program, that change propagates to all other parts of the program where the variable is found. Prolog contains no global variables.

Goal: the statement (i.e., the functor and its arguments) that Prolog is attempting to satisfy at a given point in program execution. When a question is asked at Prolog's question-mark prompt (?-), that question becomes the initial goal. Alternate goals (sub-goals) are created when Prolog calls a clause and attempts to satisfy the sub-goals in its body.

Goal Node: the specific situation or configuration desired to correctly solve the problem.

Granularity: The level of detail of a data structure, e.g., a rule, frame, or object. A very-detailed data structure is more granular than one with less detail. Thus, in an object-oriented program, objects at the bottom of the hierarchy have higher granularity than objects lower in the hierarchy.

Hardwired ANNs: ANNs with all connection weights predetermined and fixed (i.e., hardwired).

Head of a Clause: the left-hand side, or conclusion section of the clause; when a new goal is formed, Prolog first attempts to match the goal with the head of a clause. After matching the head, Prolog tries to prove the goal is true by satisfying all sub-goals in the tail of the clause.

Head of a List: the first element of a list.

Hebbian Learning: (named after Donald Hebb), a type of learning in ANNs that adjusts the connection weight between two nodes based on a correlation between the output values of those two nodes.

Heuristic: a rule-of-thumb used to guide the problem-solving pathway in a knowledge-based system. Heuristics typically limit the search space in problem domains that are not very well understood. By limiting the search space, heuristics improve expert-system efficiency and make the search more controllable.

Hidden Layer: a layer of nodes in an ANN that receives input from the input layer, performs calculations and signal processing, and passes the information to other nodes within the network.

Hill-Climbing: a variation of the generate-and-test technique; in a pure generate-and-test, the tester replies *yes* or *no*. When the tester rejects a potential solution, the program simply backtracks to the generator *without* feeding any information back to the generator. In *hill-climbing*, the tester does more than just answer yes or no. It also provides *guidance* as a form of feedback to the generator. The guidance is typically heuristic in nature.

Horn Clause: a clause that can have, at most, only one positive conclusion. If the clause fails, nothing can be concluded. If the clause succeeds, then only one (rather than multiple) positive conclusions can be drawn.

If-added procedure: a set of commands or actions attached to a frame that executes when new information is placed into a specific slot.

If-needed procedure: a set of commands or actions attached to a frame that executes when information is needed from a specific slot, and that slot is currently empty.

If-removed procedure: a set of commands or actions attached to a frame that executes when information is deleted from a specific slot.

Implementation: a specific version or type of Prolog, usually named by the company that manufactures the software (e.g., Edinburgh Prolog, Turbo Prolog, IBM Prolog, and Arity Prolog are all different Prolog *implementations*). Each implementation normally has slightly different features; different implementations are normally not compatible with each other.

Inference Chain: the *sequence* of steps, rule applications, and/or conclusions generated and used by an expert system to analyze and solve problems. To generate an inference chain, rule-based systems generally use either *forward chaining* or *backward chaining.*

Inference Engine: The portion of the knowledge-based system that contains problem-solving methodologies and/or general problem-solving knowledge. The inference engine is usually separate from, and acts upon the knowledge-base. For example, Prolog has a built-in inference engine that begins at the top of the database and moves downward in search of a match. This mechanism is the same regardless of the nature of the knowledge base.

Inference Method: the procedure used by the inference engine to systematically analyze the knowledge base and draw conclusions. Usually, the term "inference method" is applied to rule-based systems, e.g., forward-chaining or backward-chaining.

Infix Operator: an operator that lies *between* two arguments.

Inheritance: a principle used in frame-based systems, where a lower level frame inherits the properties of higher-level frame because the lower-level frame belongs to the specific classification designated by the higher-level frame.

Input: essential information or data given to the program.

Input Argument: an argument in a clause that is already instantiated when that clause is called. For example, if the goal is to find the reverse of the list **L**, where **L = [a,b,c]**, we write **reverse(L,RL)**. Here, L is instantiated to **[a,b,c]** when the clause is called, and hence **L** is the *input argument*.

Input Layer: the first layer of an ANN; this layer receives information from an external source, and passes the information into the network for processing.

Input Stream: the source port (e.g., terminal, disk, etc.) from which the program reads and inputs data.

Instantiate: to bind a free variable to a specific object for the purpose of matching. If a free variable is bound to another free variable, then the variables are instantiated to each other and are said to *share* the same data object.

Interlayer Connections: connections between nodes in an ANN where outputs from nodes in one layer feed into nodes in a completely different layer.

Internal Model Control: an advanced control strategy that uses a process model on-line and built right into the control loop.

Internal Threshold: a numerical value that controls the activation of the ANN. In most ANN algorithms, we subtract the internal threshold from the total nodal input. If this difference is below a certain level, the node is deactivated and has zero output.

Interpret: to read a line of a program and immediately execute it without compiling or translating the original code.

Interpreter: a program that reads and executes line-by-line on an "as you go" basis rather than translating or compiling the entire code into a single executable file.

Intralayer Connections: connections between nodes in an ANN where outputs from nodes in one layer feed into nodes in that same layer.

I/O: input/output, i.e., communication between the user and the computer program.

Knowledge: the accumulation of facts, rules, and heuristics programmed into the computer in a knowledge-based system.

Knowledge Acquisition: the process of taking expert knowledge, possibly from multiple sources, and translating it into a viable knowledge representation and computer code for a knowledge-based expert system.

Knowledge-Based System: a computer program that uses high-quality, in-depth, *knowledge* to solve complex and advanced problems typically requiring experts. Knowledge-based systems typically require some form of *symbolic computing*, since a large portion of knowledge is inherently qualitative. Synonymous with expert system.

Knowledge Engineer: the person who designs, builds, and debugs a knowledge-based system. Usually, the knowledge engineer is someone who is very familiar with AI techniques.

Knowledge Engineering: the process of building a knowledge-based system.

Knowledge Indexing: the ability to store large amounts of information (knowledge) and access it quickly and efficiently.

Knowledge Representation: the way of characterizing and organizing the knowledge required by an expert system to solve a complex problem. For example, the knowledge can be *rule-based* or *frame-based*.

Length of a list: the number of elements in a list, e.g., **[a, b, c]** has a length of three (atoms), and **[[a, b, c], [d, e, f]]** has a length of two (lists).

LIPS: an acronym for "Logical Inferences Per Second"; it is used to compare the speed of different implementations or versions of Prolog.

LISP: a symbolic computing language used extensively in artificial intelligence; LISP stands for LISt Processing.

List: ordered sequence of objects of any length, denoted in Prolog by the square brackets [and]. Each element of a list is separated by a comma, e.g., list **[a, [b], c]** is a list with three elements, where the second element is itself a list with only one element. All lists, except the empty list, have a head and a tail.

Listing: displaying the Prolog database, usually on the screen, and done using the **listing** built-in predicate.

Local Receptive Field Network: an ANN with unique architecture targeted to improve performance and speed-up the training session; the local receptive field network has one layer and a radially symmetric function (such as a gaussian density function). Nodes are pruned as a means of speeding up the training session.

Local Variable: a variable whose semantic meaning is limited to a specific part of the program. Thus, if a local variable changes value in one part of the program, that change *does not* propagate to any other part of the program where the same variable name is found. In Prolog, variables are local to each *clause*.

Logical Operator: an operator that identifies logical operation between Prolog goals; examples include: *and* (denoted in Prolog by a comma), *or* (denoted by a semicolon), and *not* (denoted by the built-in predicate **not**).

Logic Programming: using a computer to simulate mathematical logic. Prolog is a representation of *first-order predicate logic*.

Logical Value: a value, either 0 (i.e., *no*, or *false*) or 1 (i.e., *yes*, or *true*), given to a variable.

LRFN: Local Receptive Field Network.

Matching: the built-in unification procedure that Prolog uses to make two terms *identical*. When free variables exist, Prolog *instantiates* them to the appropriate objects to create a match. If the match fails, Prolog backtracks.

Meta-Knowledge: knowledge about knowledge, i.e., meta-knowledge is information that a knowledge-based system uses to control the program, reasoning strategy, and/or adjust the knowledge base.

Meta-Rule: a rule that describes how facts or rules in the database should be modified.

Meta-variable: a variable that later becomes the *goal* in the body of clause to which the variable was passed.

Meta-logical Predicate: a *built-in predicate* that uses meta-variables. An example is the built-in predicate **=..** (called "univ"), that creates a structure from a list.

Microfeature Concept: the unique property of ANNs, where each node affects the overall input-output pattern from the network only slightly. Each node, therefore, operates independently of other nodes, and incorporates a microfeature of the total input-output response. As a result, the network does not depend heavily on the performance of a single node, and thus, can act as a filter or process noisy data.

Model-Based Control: the use of a numerical process model to aid controller design or operation. A numerical process model can be used to: design a traditional control system, investigate control strategies off-line, or be placed in directly in a control loop and used on-line for advanced control strategies.

Model-Based Knowledge: knowledge that incorporates in-depth, quantitative information based on first-principles. Model-based knowledge is a deep-knowledge approach that is typically more quantitative than causal knowledge.

Model-Predictive Control: an advanced control strategy that uses a process model to predict response over a "long" period of time, i.e., at least as long as the open loop response of the system.

Modular Programming: a way of designing programs where the entire package is separated into sub-programs, frequently called *modules*. Modular programs are usually easier to build, maintain, and expand.

Module: a sub-program that usually achieves one major objective or theme.

Momentum: extra weight added to the change in weight factor to speed up training in an ANN.

Natural Language: also called natural language processing, the branch of AI that attempts to emulate the standard methods of communication between people, e.g., written English rather than a computer language.

Negation: the statement that declares a certain relation is *not true*.

Negation as Failure: a way to implement negation in Prolog. The negation of a goal *succeeds* if the goal itself *fails*. Likewise, the negation of a goal *fails* if the goal itself *succeeds*. For example, the statement **not(X)** succeeds if and only if goal **X** fails.

Neurode: a node in an ANN.

No: the statement declared by the Prolog program when the question is unable to be satisfied, and the goal ultimately fails.

Node: a single, specific configuration or situation in the state space. For example, in flowsheet development, a specific flowsheet design represents one *node* out of millions.

Nondeterministic: a relation is *nondeterministic* if it can generate multiple solutions to the same question, and therefore, it is unclear at the outset of the goal which solution will ultimately succeed.

Nonmonotonic Reasoning: a reasoning strategy that supports multiple lines of reasoning to the same conclusion. Nonmonotonic reasoning usually allows the acceptance of conclusions which may (temporarily) violate or invalidate certain constraints. It is useful for processing complex, unreliable information.

Not: the built-in predicate Prolog uses for negation. The statement **not(X)** implements *negation as failure*.

Numerical Modeling: problem-solving based on a strictly quantitative, algorithmic approach. Accurate values of quantitative parameters and variables is absolutely essential to the success of quantitative models.

Object-Oriented Programming: a type of computer programming that combines data and computer code together into a single, inseparable "object." The object contains not only a list of properties, but also the *required procedures to manipulate them.* Object-oriented programs closely resemble frame-based systems. However, frame-based systems define the properties of object classes, and cannot communicate between different classes. Object-oriented programming overcomes this limitation by easily allowing us to directly relate to different classes. Hence, any object may communicate with any other object.

Operator: a functor used to improve the readability of Prolog programs. Operators generally take no action on their own, and can be used in *prefix*, *infix*, or *postfix* form, depending on their location in the statement. An operator is fully specified when we know its name, precedence, and type.

Or: a logical disjunction of goals. Only one goal need be satisfied for the statement to be true.

Order-of-Magnitude Reasoning: a form of semi-quantitative reasoning that is concerned with numerical parameter estimation within one order of magnitude. Order-of-magnitude reasoning is one step less quantitative than numerical modeling, but is more quantitative than qualitative modeling.

Output: information sent from the program to the user, usually via the screen or a disk file.

Output Argument: an argument in a clause that is free when that clause is called, and becomes instantiated if the clause succeeds. For example, if the goal is to find the reverse of the list **L**, where **L = [a , b , c]**, we write **reverse(L,RL).** Here, RL is free when the clause is called, and becomes instantiated to **[c, b, a]** when the clause succeeds. Hence, RL is the *output argument*.

Output Layer: the layer of nodes in an ANN that receives input from other nodes in the network, calculates and output, and sends this output to an external receptor.

Output Stream: the destination port (e.g., terminal, disk, etc.) to which the program writes and sends data.

Parent Goal: the previously called goal that matches the *head* of the current clause that Prolog is in and attempting to satisfy.

Perceptron: an ANN with feedforward interlayer connections only, and no intralayer or recurrent connections.

Plan-Generate-Test: a three-stage problem solving approach that is a modification of the two-stage *generate-and-test*. The plan-generate-test first executes a *planning* stage before moving into the generate-and-test procedure. The *planner* restricts the state space and streamlines the generation stage. The purpose of the plan-generate-test is to improve the efficiency of the program over the straight generate-and-test.

Postfix Operator: an unary operator that is written *after* its argument. A example is the *factorial* operator, !, which can be written in a statement as **X !**.

Precedence: a characteristic property of an operator that indicates its *binding strength*. Precedence is required in expressions that have more than one operator. Precedence tells Prolog which operator is the principal functor. In Prolog, we define precedence numerically, and assign a number from 0 to 1200. The lower the number, the lower the precedence, and the more tightly binding the operator. Consequently, the operator with the highest precedence is the principal functor in the expression.

Predefined Operator: an operator with a predefined meaning that is automatically recognized by the Prolog program; no declarations are required of the user. Synonymous with *built-in operator*.

Predefined Predicate: an atom with a predefined meaning that is automatically recognized by the Prolog program; no declarations are required of the user. The *predefined predicate* is the atom that calls a predefined procedure. Synonymous with *built-in predicate*.

Predefined Procedure: a procedure with a predefined meaning that is automatically recognized by the Prolog program; no declarations are required of the user. Synonymous with *built-in procedure*.

Predicate: a representation of a Prolog relation. A predicate is specified by its ***name*** (an atom) and its *arity* (the number of associated arguments). A predicate defines the relation between the principal functor and its arguments.

Predicate Calculus: same as *predicate logic*.

Predicate Logic: a type of systematic logic that makes relationship declarations and then attempts to formally analyze the consequences of those declarations. Prolog's logic structure is based, for the most part, on *predicate logic*. As in Prolog, *predicate logic* uses constants, variables and functors to build terms. It also uses logical operators (such as *and*, *or*, and *not*) to qualify relationships. Finally, predicate logic uses *implication statements* (such as -->, which means "implies", or <-->, which means "is equivalent to") to analyze relations.

Prefix Operator: an unary operator that is written *before* its argument. An example is the numerical negative operator, -, which can be written as - X.

Principal Functor: the functor that gives a structure its *name*. It is the functor with the highest precedence in an expression, and is typically the atom outside of the parenthesis of a Prolog structure.

Procedural Nature: the characteristics of a Prolog program concerned with *how* the program executes and in what *sequences* sub-goals are called and satisfied. In contrast, the declarative nature focuses on the *relations* only and is not concerned with *how* the program executes.

Procedural Language: A computer language where we program strictly by defining *how* to solve the problem and what ordered *sequence* of actions to take. Common procedural programs include C, PASCAL, and FORTRAN.

Process Forecasting: the attempt to predict the value of measured process variables in the future based on a history of noisy or seemingly chaotic data.

Process Synthesis: chemical process flowsheet development, i.e., determining appropriate unit operations and configuring these operations into a coordinated process that achieves the desired objectives.

Processing Element: a node in an ANN.

Production Rule: a type of rule used in a forward-chaining rule-based reasoning strategy. A production rule is an "IF-THEN" type of rule where, if multiple conditions are met, the rule "fires" and takes action.

Production System: a forward-chaining rule-based system. A production system contains "IF-THEN" type of rules, known as *production rules.*

Program: a set of clauses that can be syntactically understood and interpreted by a Prolog *compiler* or *interpreter.*

Prolog: a computer language that attempts to emulate mathematical logic. Prolog stands for PROgramming in LOGic. Prolog has a procedural nature that is required for the language to operate efficiently on a computer. Besides this essential procedural nature, Prolog relies on its declarative nature that emulates *predicate logic.* Prolog is the leading language in the field of *logic programming.*

Prototype System: an operational expert system that is incomplete, in the early stages of development, and still in need of adjustment and refinement. Prototype systems are developed quickly so that the knowledge engineer can get rapid feedback on areas of improvement.

Pruning: reducing the alternatives and reasoning pathways open to the expert system as a means of simplifying the problem and improving the system's efficiency. Pruning reduces the *state space*, and there are a number of approaches available to reduce the size of the state space. One of the most common is constraint satisfaction, where states within the state space are pruned if they violate specific constraints.

Pure Prolog: programs written such that they do not rely on any *built-in* predicates.

Qualitative Modeling: a form of simulation based on symbolic (i.e., qualitative) relationships between concepts. Qualitative models are used to express directionality, e.g., as the speed of the pump motor goes up, the volumetric flow rate goes up. However, qualitative models do not represent the knowledge numerically.

Question: a Prolog clause with a *body*, but no *head*. A *question* is used to initiate program execution. It is a goal or sequence of goals that Prolog attempts to satisfy. If the question is successfully satisfied, Prolog will also report the instantiation values of all free variables in the original question. A *question* is synonymous with a *query*.

Real-Time Expert Systems: expert systems that take in data and respond fast enough to be used on-line in a real-time environment, e.g., in on-line process control or fault diagnosis. Most expert systems are too slow to be used in real-time.

Recall Phase: the phase of ANN development where we assess the networks performance entering a specific input into the network, calculating the output, and assessing the error from the desired input-output response.

Reconsult: to read Prolog relations from a file and add them to the database, while simultaneously *retracting* already existing clauses with the same name in the database. Thus, to *reconsult* is to read the new relations off of a file and *replace* the old ones. This action is in contrast to a simple *consult* which reads new clauses and *adds* to the database, while simultaneously ***maintaining*** old clauses with the same name.

Recurrent Connection: a connection in an ANN where the output from a node feeds into itself as input.

Recursive Relation: a relation that is defined in terms of itself. An example is the **member** relation, which relies on the recursive rule:

member(X,[_ | T]):- member(X , T).

Reinforcement Learning: a type of supervised learning in ANNs characterized by adjustment of weight factors based on a single *scaler* error value. This learning is contrasted with error-correction learning, where an out put error *vector*, $\underline{\epsilon}$ is used to adjust the weight factors. Thus, reinforcement learning is "selectively supervised" and is faster and easier to use than error-correction learning. On the downside, reinforcement learning uses less precise information to adjust the weights.

Resatisfy: to satisfy a goal in an alternate way. Normally required when Prolog backtracks into a previously satisfied goal.

Robustness: the property of a system to be able to (non-catastrophically) handle situations outside the narrow scope of the system's expertise. A robust system gradually degrades in performance when it is given inconsistent, erroneous, or incomplete data. See also brittleness.

Rule: a Prolog clause that is *conditionally* true, i.e., the head of the clause is true if the body (condition section) can be proven true. A rule must have both head and a body. If the clause has no body, it is a fact; if it has no head, it is a question.

Rule-Based System: a knowledge-representation that uses IF-THEN conditional statements (rules) to build an *inference chain* to ultimately solve problems.

Runtime Error: an error that occurs during program execution. An example is an attempt to divide by zero.

Satisfy: to prove that a goal is true. A goal is satisfied (true) if it can be matched with a clause in the database, and all conditional sub-goals (if any) associated with the matched clause are also proven true.

Search: in a knowledge-based system, *search* is the systematic procedure used to analyze and reason through the knowledge-base and solve the problem. Typically, the pathway between the starting node and the goal node is unknown ahead of time, and the search determines the pathway.

Search Space: the state space open to the system to analyze in an attempt to find a solution to the question or problem.

Self-Tuning Controller: a controller that can change its tuning parameters on-line based on changes in process conditions or dynamics.

Semantic Network: a knowledge representation characterized by a network of nodes, each of which represents a concept, item, or object, connected by arcs describing the relation between the two nodes that the arc connects. For example, the node **centrifugal_pump** may be connected to the node **pump** by the **isa** arc.

Shallow Knowledge: knowledge based on macroscopic rules and heuristics, with no understand of the fundamental physical principles supporting and justifying this knowledge. Shallow knowledge systems are brittle, i.e., their performance falls off very rapidly (sometimes catastrophically) outside of their domain of expertise. See also deep knowledge.

Share: two variables are said to *share* the same data object if they are both instantiated to that same object.

Shared Variable: a variable that occurs more than once in a clause.

Simple Object: a Prolog *constant* or *variable*. Constants can be *atoms* or *numbers*, and numbers can be *integers* or *real numbers* (floating point).

Sigmoid Function: a continuous, monotonically increasing, S-shaped numerical function. Sigmoid functions typically have limiting values of [0,+1] or [-1,+1], e.g., as X → ∞, the output → 1.

Sigmoid Threshold Function: see *threshold function*.

Sign-Directed Graph: a graphical representation of causality, where nodes, representing state variables, are connected to each other in a cause-and-effect network. Each connection has a positive (+) or negative (-) sign to indicate the directionality of influence between variables.

Slot: a property location in a *frame*. For example, we may have the frame *pump* that contains nine slots: 1) unit number, 2) type, 3) material of construction, 4) capacity, 5) motor, 6) head, 7) temperature of fluid, 8) inlet stream, and 9) outlet stream.

Solution: the set of variable instantiations that Prolog has performed to successfully satisfy the goal.

Speech Recognition: the branch of AI that attempts to process, understand, and interpret audio communication and human speech.

Spy Point: the point in the program where Prolog's built-in tracing mechanism is turned on for the purpose of debugging the program.

Starting Node: the beginning situation or configuration from which we initiate the *search* through the *state space* to get to the *goal node*.

Stack Overflow: a runtime error that occurs when the number of execution steps remembered by Prolog exceeds the available computer memory. Each execution step taken by Prolog in remembered and placed onto the computer stack memory to allow for backtracking. When the stack requirements exceed that available by the computer, a *stack overflow* error occurs.

State Space: the collection of all possible situations or configurations in a problem. For example, in flowsheet development, there may be 10^{20} different flowsheet configurations that could successfully solve the problem.

Stochastic Learning: a type of learning in ANNs that uses statistics, probability, and/or random processes to adjust the connection weights. Error-correction learning has a problem of getting trapped at a local minimum of error; some stochastic methods have the ability to avoid local minima and move to the global minimum in error.

Stream: the source (input stream) or destination (output stream) device (terminal, disk file, etc.) that Prolog reads and writes to.

String: a sequence of characters enclosed in double quotation marks (").

Structure: a structured object.

Structured Object: a Prolog data object consisting of a functor and its arguments. In Edinburgh Prolog, the functor must be an atom and is written first. Following the functor, the arguments are written, and are enclosed in parenthesis and separated by commas: *functor*(arg_1, arg_2, ... , arg_n).

Subsymbolic Processing: the processing used by artificial neural networks, characterized by microscopic interactions that eventually manifest themselves as macroscopic, symbolic, intelligent behavior.

Success: fulfillment of the goal(s) posed to the Prolog program.

Supervised Learning: a type of learning in ANNs where an external teacher controls the learning and incorporates global information and training data.

Symbol: a string of letters that represents a single concept, object, or item.

Symbolic Computing: a branch of computer science that deals with the processing of non-numerical symbols and names. Symbolic computing is contrasted with the more classical numerical computing, which deals with the processing and calculation using numbers.

Syntax Error: an error that occurs when the program as written is incorrect according to the rules of syntax of Prolog. An example is two consecutive commas: **W:- X ,, Y.**

Tail of a List: the remainder of a list after removal of the first element. The tail is *always* another list. If the list has only one element, the tail is the empty list, **[ ]**. The empty list is atomic and cannot be separated into a head and tail.

Task-Oriented Approach: an approach to problem-solving in expert systems by dividing the problem up into sub-problems such that a single task, when executed, satisfies the sub-problem. Usually, the complete problem-solution is the combination of all the tasks used to solve every sub-problem.

Term: any Prolog data object.

Threshold Function: the functional form used in ANNs to calculate nodal output. At very low input values, the threshold-function output is zero. At very high input values, the threshold-function output is one. Since threshold functions are typically sigmoid functions, they *gradually* deactivate the node as the input magnitude decreases.

Trace: a display of the step-by-step execution Prolog takes to satisfy a goal.

Tracing: following the step-by-step execution Prolog takes to satisfy a goal.

Trace Mode: a step-wise execution mode that displays the program trace. In the *trace mode*, Prolog executes only one step at a time and displays the result. After executing a single step, Prolog usually waits for the user to hit the ENTER key on the keyboard to execute the next step.

Training Phase: the initial phase of ANN development, characterized by repeatedly presenting sets of input-output data to the network and adjusting the weights of the interconnections to minimize error.

Tree Structure: a way of organizing knowledge (e.g., the state space) as a directed graph where nodes are connected to each other, such that we may move to different nodes (states) in the problem, usually deeper into the structure.

Unary: having only one argument.

Unbound Variable: a free, uninstantiated variable. An unbound variable can take on any value as the program executes to facilitate a match.

Unification: a *matching* process where two terms are made identical by *instantiating free variables.*

Uninstantiate: to convert an instantiated variable to a free, uninstantiated variable. A variable must be *uninstantiated* if Prolog backtracks into the clause where the variable was originally instantiated. Backtracking requires previously instantiated variable to become free in search of an alternate solution.

Uninstantiated Variable: a free, unbound variable. An unbound variable can take on any value as the program executes to facilitate a match.

Unit Clause: a Prolog fact.

Unsupervised Learning: a type of learning used in ANNs where no external teachers is used and the ANN relies upon internal control and local information. Frequently, the ANN develops its own models automatically without additional information input.

Value of a Variable: the data object to which a variable is instantiated. Importantly, this *value* is not necessarily numeric. A variable can be instantiated to any simple or structured object. The value does *not* have to be "fully instantiated," i.e., a variable can be instantiated to another free variable, or to a structure containing free variables. In these cases the *value of the variable* is not entirely specified.

Variable: a Prolog data object the can take on any value as the program executes to facilitate a match. Variables in Prolog begin with a capital letter (e.g., **W, Result, Feasibility**), or an underscore character (e.g., **_x**, **_12**, **_A1**).

Weight Factor: an adjustable parameter in ANNs. The weight factor is frequently denoted as w_{ij}, and is the numerical value that we multiply the output from node i by to give the input to node j.

Yes: the statement declared by the Prolog program when the question is satisfied, and the goal ultimately succeeds.

APPENDIX D

SOLVENT-BASED SEPARATIONS

APPENDIX D

SOLVENT-BASED SEPARATIONS

This appendix describes a knowledge-based approach to solvent-based separations. We have discussed ordinary distillation in chapters 12-15. Distillation uses energy to separate materials, and hence is an energy-separating-agent (ESA) process. Solvent-based separations use a solvent to achieve the separation by modifying the equilibrium behavior of the mixture. Thus, solvent-based separations are mass-separating-agent (MSA) processes.

In this appendix, we discuss both the chemical engineering and AI aspects of knowledge-based design of MSA processes. We discuss aspects of shortcut feasibility analysis, absorption heuristics, knowledge representation, and program control structure. We demonstrate this approach through the use of gas absorption.

D.1 KNOWLEDGE-BASED DESIGN OF SOLVENT-BASED SEPARATIONS

A. Introduction

Knowledge-based approaches to separation process synthesis use facts, rules, and heuristics to guide the flowsheet development. A number of researchers have attempted to use knowledge-based techniques for computer-aided process synthesis. However, the majority of this work has been strictly conceptual. Researchers have proposed control structures and reasoning pathways for these systems, but few of these proposals have actually been implemented into operational prototype expert systems. Conceptual development of knowledge-based systems is valuable, but the "acid test" is how well the concept operates when implemented. Because few, if any, of these proposals have been implemented, it is difficult at times to judge their utility.

A knowledge-based system must be able to develop flowsheets *accurately* and *efficiently*. An important challenge arises when addressing the accuracy of the knowledge-based system. Short of a rigorous, multistage and multicomponent thermodynamic equilibrium analysis, how do we determine if the flowsheet proposed by the system is thermodynamically feasible? The goal of process synthesis is to develop a feasible flowsheet concept, which can then be further evaluated by rigorous simulation and optimization using computer-aided design (CAD).

Proper process synthesis must ensure that the proposed flowsheet is indeed feasible and practical. However, this assurance must be developed prior to rigorous flowsheet optimization.

To synthesize initial flowsheets, engineers rely on both qualitative and quantitative information. Understandably, researchers have attempted to use knowledge-based systems in process synthesis. Most knowledge-based systems have had one of the following drawbacks: 1) systems using shallow-knowledge tend to be brittle, and in the presence of novel situations, may be unreliable; 2) systems using deep knowledge require numerical models, which are cumbersome and run too slowly in an expert-system environment to be practical. We propose the use of heuristics coupled with a shortcut design technique for process synthesis using MSA separations. We demonstrate the excellent balance of accuracy and efficiency that shortcut design techniques bring to a knowledge-based system. We illustrate the use of these shortcut design techniques using absorption.

B. Literature Review

There has been very little prior work on knowledge-based approaches to MSA processes. Virtually all applications of separation process synthesis have been for ordinary distillation performing sharp splits only.

Barnicki and Fair (1990), have proposed a conceptual framework for MSA process synthesis. This work, to date, is strictly conceptual -- no prototype system has been developed. Barnicki (1989), however, *has* developed an operational prototype system that performs process synthesis using ordinary distillation with sharp splits only.

Let us discuss Barnicki and Fair's conceptual framework for MSA process synthesis. They propose a *general* separation expert system that is able to handle solids, liquids, and gases. They recommend a rule-based approach that would perform:

(1) process synthesis- configuration of the actual flowsheet,

(2) method selection- selecting the best separation method for each separator, e.g., ordinary distillation, extractive distillation, etc., and

(3) rigorous design- design and size specific equipment, and then perform a cost analysis.

As mentioned, Barnicki (1989) has developed a prototype system for sharp splits only. His proposed system is broad and ambitious, but the currently implemented system is narrow. The scope of the proposed system is so wide that implementing the entire system will take a tremendous amount of effort. Nevertheless, the *process synthesis* portion of the proposal could be implemented and holds promise. To date, he has not addressed the thermodynamic feasibility of splits that the system is considering -- all splits are assumed to be feasible. Indeed, in one example from Barnicki (1989), he attempts to split o-xylene, m-xylene, and p-xylene using sharp splits with ordinary distillation. These separation are practically infeasible; the relative volatilities are too close to use distillation. Consequently, fractional crystallization is used commercially rather than distillation. Unfortunately, Barnicki's system does not recognize this fact and recommends an infeasible separation.

Barnicki and Fair are in the early stages of development (their first paper came out in April 1990; EXSEP has been in operation since June 1989). They may address the feasibility issue in the future when they move from the general, structural considerations to more specific engineering considerations.

For a detailed review of currently available and developing knowledge-based systems in chemical engineering, interested readers may refer to Chapter 16. This review summarizes the applications of knowledge-based systems to process fault diagnosis, process control, process design, process planning and operations, process modeling and simulation, and product design, selection, and development.

D.2 EXSEP AND GAS ABSORPTION

In this section, we discuss MSA processes, and in particular, absorption. We then introduce EXSEP's knowledge representation for absorption, including absorption heuristics, shortcut feasibility analysis, and program control structure. We close with an example of the ABSORB module, and compare these results with those from a rigorous computer simulation.

A. Characteristics of Absorption

Ordinary distillation is the most frequently used separation method in the chemical process industry. However, ordinary distillation is not always the best separation method to use. For example, if low-boiling constituents are in the vapor phase and require separation, gas absorption is favored over some type of cryogenic distillation. In gas absorption, components in a vapor phase are removed from that phase by contacting with a liquid. Thus, gas absorption is a *solvent-based separation*, i.e., a liquid (mass-separating agent, MSA) is chosen that has particular affinity for the vapor-based component(s) that we desire to remove. A typical absorption column, a lean-oil absorber for the removal of hydrocarbons from a hydrogen-rich stream, is shown in Figure D.1.

B. Heuristics for Absorption

When do we use absorption, and how do we synthesize processes using absorption? We have compiled a comprehensive list of absorption heuristics that answer these questions. As with distillation, we divide these heuristics into method heuristics, species heuristics, composition heuristics, and design heuristics.

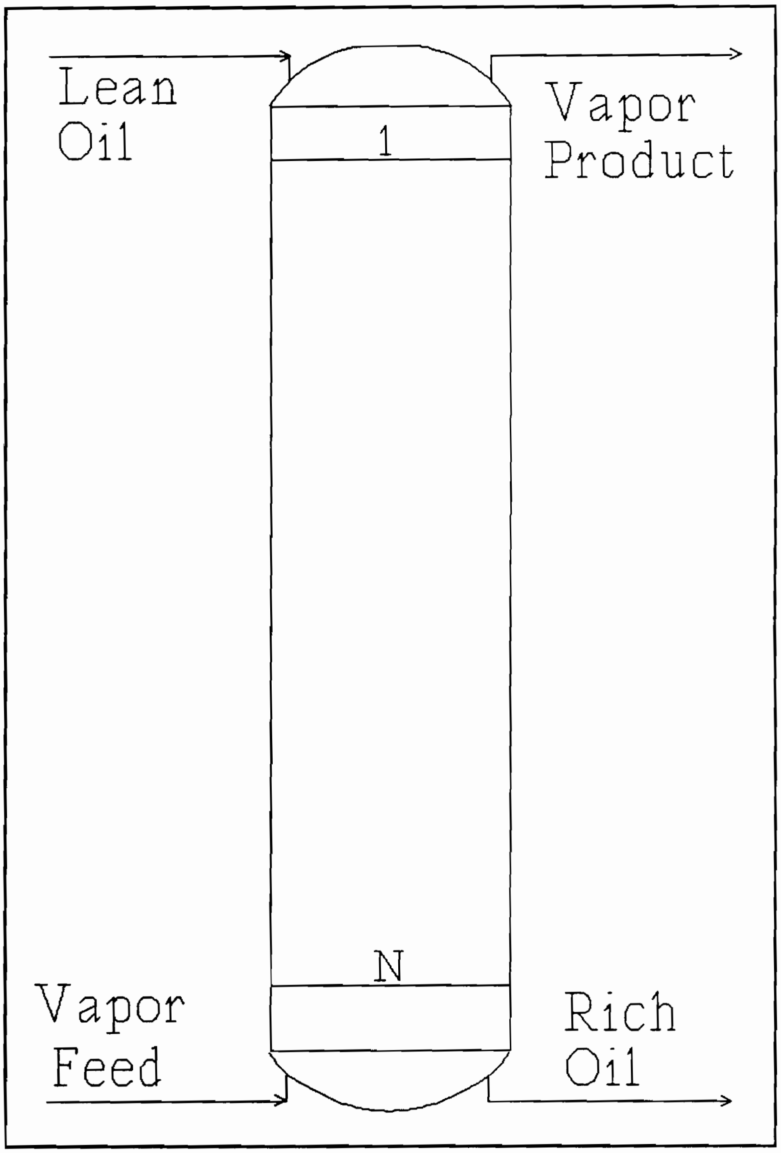


Figure D.1. The absorption process.

METHOD HEURISTICS

Absorption is favored when:

- Trace (2-3 mol%) amounts of a high-boiling feed component are to be removed from a vapor stream. (Keller, 1982)

- The feed stream has to be cooled, compressed, or both before the bubble point is reached. (Keller, 1982)

- The relative volatility (α) is too close for normal distillation, and the solvent chosen significantly affects α. (Keller, 1982)

- A trace amount of material can be removed through irreversible reaction with solvent. Good only for systems where: 1) the material to be removed is in low concentration (1000 ppm to 10,000 ppm) 2) solvent cost is low, and 3) spent solvent is easily salvaged or disposed. (Keller, 1982)

SPECIES HEURISTICS

(none)

COMPOSITION HEURISTICS

- For trace removal, favor a solvent that has strong affinity for the solute. This combination will usually require both temperature elevation and pressure reduction to recover solvent. (Keller, 1982)

- For bulk removal, favor a solvent with lower affinity for the solute, and utilize pressure reduction only for solvent recovery. (Keller, 1982)

DESIGN HEURISTICS

We divide design heuristics into MSA, stage, and energy-conservation heuristics.

MSA HEURISTICS

- Favor a selectivity $\geq$ 2.0; if this is not attainable, utilize reboiled absorption. (Keller, 1982)

- Favor solvents with: 1) high capacity for absorbed material, low volatility, low viscosity, low toxicity, and low corrosivity. (Keller, 1982)

- If a reactive solvent is used, favor a reaction that is reversible. (Treybal, 1980)

- Favor solvents that yield reversible chemical absorption rather than those over irreversible chemical or physical absorption. (Douglas, 1988)

STAGE HEURISTICS

- It is almost always advantageous to have N $\geq$ 5 (theoretical number of stages). The only exceptions are low desired recoveries of solute (< 80-90%) or extremely soluble and selective solute and solvent combination. (Keller, 1982)

- If an absorber is not performing up to specifications, attempt to improve stage efficiency. [Note: stage efficiency in absorption ranges from 10-50%, while distillation can approach 80%] (Keller, 1982)

- Favor bubble-cap tray design only when there is: 1) low liquid flow rate, 2) high turndown ratio service, or 3) multiple feed and draw-off points. Investigate valve trays if bubble-caps are appropriate. (Kohl, 1987)

- Favor sieve tray design when there is: 1) medium liquid flow rate, or 2) difficult separation requiring many stages. (Kohl, 1987)

- Favor packed columns when there is: 1) high liquid rate, 2) difficult separations requiring many stages, 3) foaming or corrosive liquids, or 4) design requiring maximum flexibility and versatility. (Kohl, 1987)

- Favor spray contacting when there is: 1) high liquid rate, 2) easy separation requiring only one stage, 3) corrosive fluids, 4) solids or viscous fluids, or 5) low ΔP required across column.

- Minimize solvent circulation rates by increasing the number of theoretical stages (especially if N < 5). (Keller, 1982)

ENERGY CONSERVATION AND HEAT INTEGRATION

- Favor staged-blowdown pressure reduction for solvent recovery. (Keller, 1982)

- Place inter-coolers in column to minimize adiabatic temperature rise. (Keller, 1982)

- Use unabsorbed "product gas" where possible as the stripping gas. (Keller, 1982)

- Favor the pressure-reduction mode of solvent recovery (gas desorption or stripping) if the feed gas is originally supplied at high pressure. (Keller, 1982)

- If the vapor feed to the absorption column is low pressure, it normally *does not* pay to increase column pressure. Compressor capital costs will usually outweigh solvent-recovery savings resulting from reduced liquid flow rate. Instead, with low pressure vapor feed, favor a low pressure column and temperature elevation as the means of desorption. (Douglas, 1988)

- Favor low solvent temperature inlet, and minimize the temperature rise in the column through the use of an inter-cooler. (Douglas, 1988)

- For isothermal absorbers targeting high recoveries (> 99%), favor a L/kG between 1 and 2, with L/kG ˜ 1.4 as optimal. [Higher values of L (increased solvent flow rate) raise stripper costs. Lower values of L (reduced solvent flow rate) require more equilibrium stages and increase absorber costs.] (Douglas, 1988)

C. Knowledge Representation: Overview

1. Fundamental Approach

For the absorption module, we maintain the basic approach that EXSEP uses in distillation sequencing, specifically, the plan-generate-test approach with:

(1) The component assignment matrix and bypass analysis;
(2) Shortcut feasibility analysis;
(3) Heuristic analysis to guide the search; and
(4) Evolutionary improvement of the initial flowsheet.

Let us briefly review the plan-generate-test approach used in EXSEP. The strategy consists of three major portions, as shown in Figure D.2. These three portion are:

- *Planner*- problem representation and bypass analysis. The *Planner* sets up the component assignment matrix, performs bypass analysis and methods selection, and implicitly passes a set of splits that obey heuristic D1 to the generator.
- *Generator*- feasibility analysis. The Generator performs a shortcut feasibility analysis, eliminates splits that are thermodynamically impractical or infeasible, generates a set of feasible splits, and passes this set on to the tester.
- *Tester*- heuristic synthesis and evolutionary improvement. The tester uses a heuristic analysis to recommend the best split passed from the generator. The tester will never fail on its own -- it will always recommend at least one split. The user can reject this recommendation and force backtracking in search of alternate splits. This is especially useful for generating competitive alternatives.

The knowledge representation tools, from both a chemical engineering and AI perspective, are shown in Table D.1.

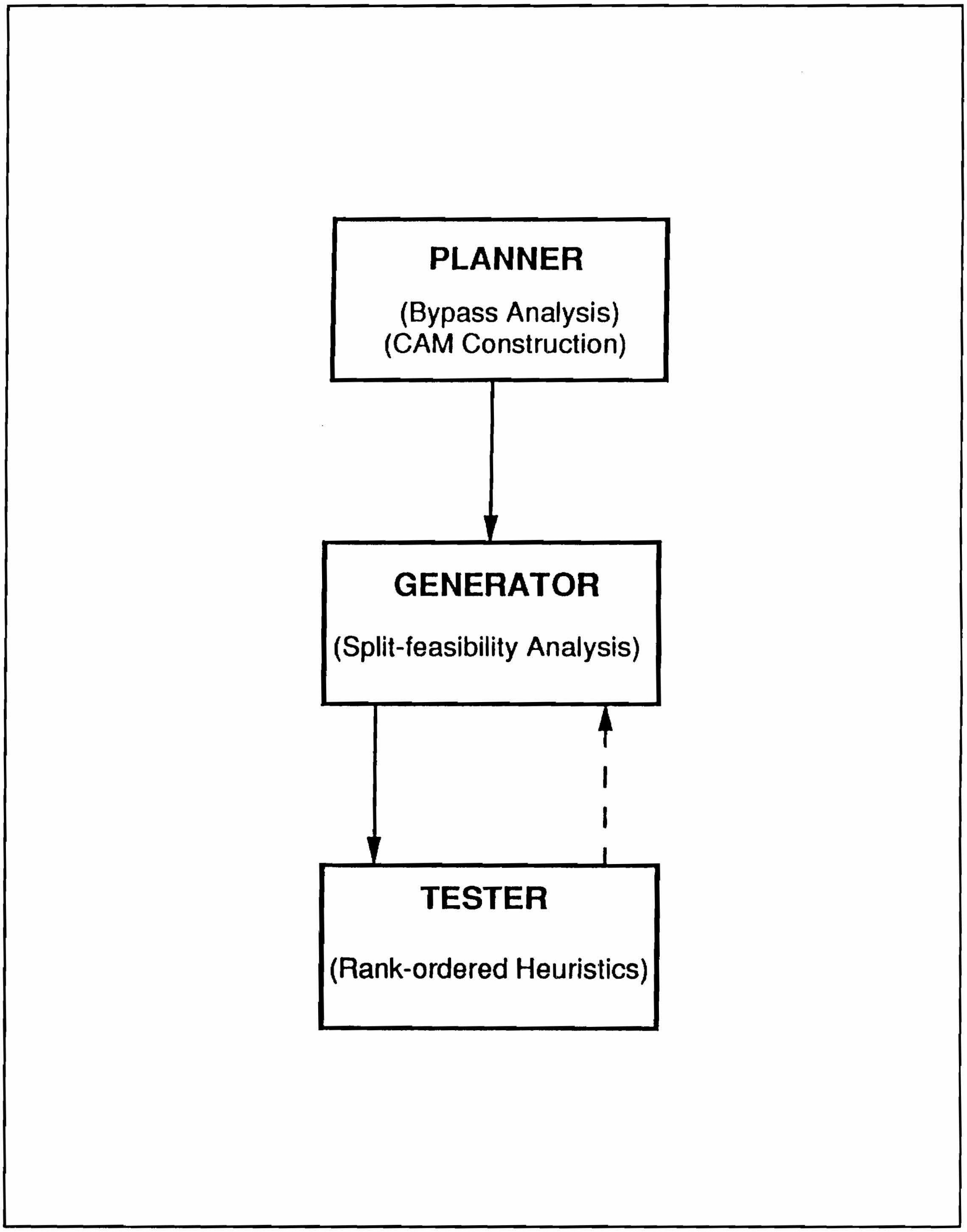


Figure D.2. EXSEP's Plan-Generate-Test Strategy.

Table D.1. Chemical Engineering an Artificial Intelligence Aspects of EXSEP's Knowledge Representation.

CHEMICAL ENGINEERING PERSPECTIVE	ARTIFICIAL INTELLIGENCE PERSPECTIVE
COMPONENT ASSIGNMENT MATRIX	
• Identifies Opportunistic Separations • Able to Handle ESA and MSA Processes • Facilitates Bypass Analysis	• Facilitates Problem-Decomposition Approach • Implements Material-Balance Constraints Globally • Provides for Conflict-Resolution on Material Constraints • Facilitates Either List Processing or Logic Programming • Part of the *Planner*
BYPASS ANALYSIS	
• Reduces Downstream Mass Load • Reduces Cost • Can Make Previously Infeasible or Impractical Separations Feasible	• Simplifies the Problem • Part of the *Planner*
SHORTCUT FEASIBILITY ANALYSIS	
• Essential for Multicomponent Separations • Ensure the Technical Feasibility and Practicality of Results	• Form of Deep Knowledge • Use of Shortcut Analysis is Efficient • Generates Feasible Separations • Part of the *Generator*
HEURISTIC SEPARATION SEQUENCING	
• Use Proven Set for Accuracy • Rank-Ordered for Easy Conflict Resolution	• Facilitates Rule-Based Reasoning Strategy • Part of the *Tester*
EVOLUTIONARY SYNTHESIS	
• User-Directed by Rejecting EXSEP's Recommendation	• Implemented by Backtracking from *Tester* to *Generator*

2. Objectives

The ABSORB portion of EXSEP has the following objectives:

Objective 1: To develop the essential chemical engineering tools to systematically synthesize feasible and economical multicomponent separation sequences using MSAs. These tools include:

(1) *Constraint-handling mechanism*- to ensure that the systems abides by all design constraints, and in particular, the material balance constraint.

(2) *Shortcut feasibility analyses*- to ensure that all separations recommended are thermodynamically feasible.

(3) *Heuristic flowsheet synthesis*- to guide the overall design development, leading to energy-efficient, cost-effective flowsheets.

(4) *Evolutionary flowsheet improvement*- to opportunistically modify a currently existing flowsheet for the purpose of generating a new, more cost-effective design.

Objective 2: To convert these tools into a knowledge representation suitable for an additional module in EXSEP.

Objective 3: To write and actually develop the ABSORB MODULE, that performs separation process synthesis using MSAs accurately and efficiently.

Objective 4: To develop a system that readily supports both evolutionary synthesis and retrofit design.

Our focus is to use sound engineering principles and an excellent knowledge representation to develop a module that is:

- Accurate- flowsheets developed must be technically feasible.
- Efficient- the module must be practical and useful; it must solve problems quickly and efficiently to be successfully implemented on a *personal computer*.
- Flexible- the module must be capable of generating process alternatives.

We now turn our discussion to the knowledge representation that achieves these objectives.

D. Component Assignment Matrix

To achieve the objectives stated in the previous section, a number of additions and modifications are required to EXSEP's knowledge representation. One key area of adjustment is the component assignment matrix (CAM).

The CAM is expanded to include the addition of mass separating agents (MSA) into the material balance. MSAs facilitate mass transfer by selectively partitioning one or more components. However, MSAs must be removed from the system too. A CAM processing strategy has been developed to both add and remove MSAs from process streams.

The CAM is a matrix where rows and columns are ordered according to volatility. Let us consider the removal of H_2S from a vapor stream of light hydrocarbons using monoethanolamine (MEA) absorption. EXSEP's original CAM is:

$$\begin{array}{c|cccc|}
 & \text{H2S} & \text{C3} & \text{C4} & \text{C5} \\
\text{p3} & 0 & 9 & 15 & 25 \\
\text{p2} & 0 & 20 & 12 & 8 \\
\text{p1} & 4 & 0 & 0 & 0
\end{array}$$

The volatility of each component is given by its thermodynamic equilibrium ratio, also know as the K-value. In this CAM:

$$K_{H2S} > K_{C3} > K_{C4} > K_{C5}$$

Thus, columns in the CAM are ranked according to volatility; columns to left are more volatile than those to the right.

Note also, however, that rows are ranked according to a *molar average K-value*. For a multicomponent product, the molar average K-value is defined by:

$$K_{avg} = \sum_{i} y_i \, K_i \tag{D.1}$$

Thus, in the previous CAM, $K_{p1} > K_{p2}$.

As mentioned, we wish to first remove H_2S from the stream via absorption. We add MEA solution to the mix, and EXSEP gives:

$$\begin{array}{l|ccccccc|}
 & \text{C3} & \text{C4} & \text{C5} & & \text{H2S} & & \text{MEA} \\
\text{solvent} & 0 & 0 & 0 & & 0 & & \text{L} \\
\text{p1} & 0 & 0 & 0 & & 4 & & 0 \\
\text{p3} & 9 & 15 & 25 & & 0 & & 0 \\
\text{p2} & 20 & 12 & 8 & & 0 & & 0 \\
 & & & & \overset{\uparrow}{\text{V3}} & & \overset{\uparrow}{\text{V4}} &
\end{array}$$

EXSEP arrives at these results by considering the actual K-value (Henry's law constant) for each component, and the average molar K-value for each product. H_2S reacts reversibly with MEA. Thus, in the presence of MEA, H_2S jumps from the most volatile to the least volatile, next to MEA itself. Likewise, product p1 jumps from being the most volatile to the least volatile, next to the solvent itself.

Now that EXSEP has rearranged the CAM, we may execute separations (provided, of course, that they are feasible). Separation V3 is an absorber, with C3, C4, and C5 in the overhead, and H_2S and MEA in the bottom. Likewise, V4 is a stripper, where MEA solvent is recovered.

E. Shortcut Feasibility Analysis: the Kremser Equation

The Kremser equation is used for shortcut feasibility analysis for absorption in the *generator* portion of EXSEP.

The Kremser equation can be used not only for dilute gas absorption, but also for liquid-liquid extraction and stripping (Wankat, 1988, pp.485-493, 529-530, 550-553). In addition, it can be expanded to include other separations. The following assumptions are associated with its development:

- Constant molal overflow (L/V is constant)
- Isothermal system
- Isobaric system
- Negligible heats of absorption
- Straight equilibrium line (i.e., Henry's law applies)

The following sections outline the development and use of the Kremser equation in EXSEP. We demonstrate its applicability to absorption, and also discuss other applications.

1. Development of the Kremser Equation

Consider the equilibrium cascade performing gas-liquid contacting in Figure D.3. We assume that for this absorber, Henry's law applies to all vapor components that distribute into liquid L, the solvent. Thus, we assume that

$$y_i = k_i x_i + b \qquad \text{(D.2)}$$

is an acceptable equilibrium expression for all vapor components *i* that distribute into liquid L, where y_i is the mole fraction of component i in the vapor, k_i is the Henry's law constant for component i, x_i is the mole fraction of component i in the liquid, and b is an arbitrary constant.

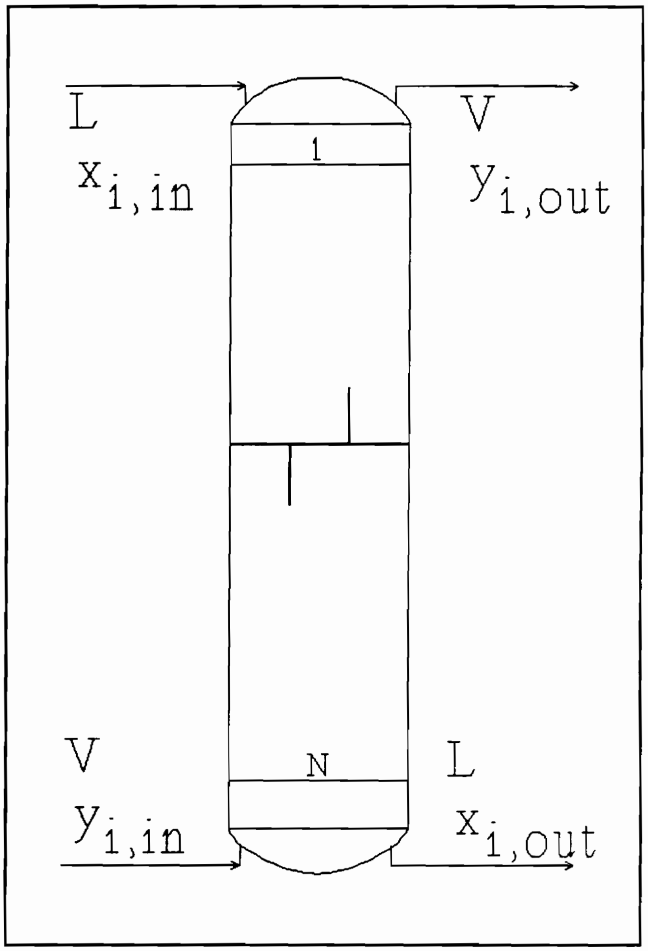


Figure D.3. An equilibrium-cascade performing gas-liquid contacting.

A material balance around the entire absorber for component i gives the following equation:

$$\frac{y_{i,\,in} - k_i x_{i,\,in}}{y_{i,\,out} - k_i x_{i,\,in}} = \frac{A^{N+1} - 1}{A - 1} \qquad \text{(D.3)}$$

where:

$$A = \frac{L}{k_i V} \tag{D.4}$$

and:

- $y_{i,\,in}$ = mole fraction of component i in the vapor feed.
- $y_{i,\,out}$ = mole fraction of component i in the vapor product.
- $x_{i,\,in}$ = mole fraction of component i in make-up solvent.
- k_i = Henry's Law constant for component i in the solvent.
- L and V = solvent (liquid) and vapor flow rates, respectively.
- N = number of theoretical stages.

The parameter A is called the *absorption factor*.

This relation is known as the *Kremser equation*, and is valid for *any* equilibrium-stage contacting unit, as shown in Figure D.3, within the assumptions upon which the equation is based. Additional applications of the Kremser equation (Wankat, 1988, pp. 552-53) are shown in Table D.2. In some cases, the Kremser equation may not appear to be applicable, since the equilibrium relation is not linear (in this case, a McCabe-Thiele analysis appears more appropriate). In that situation, we always have the option of applying the Kremser equation *repeatedly* over the equilibrium relation, using the linear assumption on *small concentration intervals*.

Table D.2. Applicability of the Kremser Equation.

OPERATION	FEED	CONSTANT FLOW	UNITS
Dilute Absorption	Vapor	Total flow rates	y, x mole fractions
Stripping	Liquid	L (kg solvent/hr) G (kg gas/hr)	Y, X mass fractions
Dilute Stripping	Liquid	Total flow rates	y, x mole fractions
Dilute Extraction	Raffinate	D (kg diluent/hr) S (kg/solvent/hr)	$Y = \frac{kg\ solute}{kg\ solvent}$ $X = \frac{kg\ solute}{kg\ diluent}$
Very Dilute Extraction	Raffinate	E (kg extract/hr) R (kg raffinate/hr)	y, x mass fractions (extract and raffinate)
Washing	Solids + underflow liquid	U (underflow liquid kg/hr) O (overflow liquid kg/hr)	y, x mass fractions (overflow and underflow)
Leaching	Solids and solutes	L (kg solvent/hr) S (kg insoluble solids/hr)	$Y = \frac{kg\ solute}{kg\ solvent}$ $X = \frac{kg\ solute}{kg\ solid}$
Adsorption and Ion exchange	Vapor or Liquid	S (kg adsorbent/hr) F (kg fluid/hr, liquid or vapor)	$Y = \frac{kg\ solute}{kg\ solvent}$ $q = \frac{kg\ solute}{kg\ adsorbent}$
Three-phase systems	One or two phases	L1 (kg liquid-1/hr) L2 (kg liquid-2/hr) V (kg vapor/hr)	y, x mole fractions (both phases)

2. Absorption-Column Operating Conditions

If we apply the Kremser equation to absorption, variables V, $y_{i,\ in}$, and $y_{i,\ out}$ are externally specified. For initial design purposes, we can also assume we have pure solvent ($x_{i,\ in} = 0$), since subsequent stripping operations on the solvent are usually very effective. (This assumption can be relaxed later, when the stripping operation is investigated in detail.) Therefore, N, the number of stages, and L, the solvent flow rate, are free design variables.

Determining the values of N and L is an economic decision. A high value of N (more equilibrium stages) leads to a lower solvent flow rate L. This lower solvent flow rate reduces the operating cost, and keeps the subsequent solvent-recovery capital cost at a minimum. However, the absorber itself will need more stages to offset the lower solvent flow rate. Conversely, a low N leads to a high L. This smaller column reduces absorber capital cost, but increases recovery costs. A key goal in absorber design is to identify the trade-off of L vs. N. We use the following heuristics (Douglas, 1988, pp. 85-87):

$$\mathit{RULE}\ 1: \quad 1.3 \leq \frac{L}{k_i\, V} \leq 1.6 \tag{D.5}$$

and (Keller, 1982)

$$\mathit{RULE}\ 2: \quad N > 5 \tag{D.6}$$

If desired recoveries cannot be achieved within these bounds, perhaps a more efficient solvent should be considered.

3. Feasibility Analysis: Distributions of Other Components

Once we determine appropriate values for the solvent flow rate, L, and the number of theoretical stages, N, we can assess the distributions of other vapor components, again using the Kremser equation, which for component j, is:

$$\frac{y_{j,\ in} - k_j x_{j,\ in}}{y_{j,\ out} - k_j x_{j,\ in}} = \frac{A^{N+1} - 1}{A - 1} \tag{D.7}$$

Here, we again assume $x_{j,\ in}$ is equal to zero. Since N and L are now known, A is also known, and we can estimate the recovery of component j by:

$$y_{j\ out} = \frac{y_{j\ in}\,(A - 1)}{A^{N+1} - 1} \tag{D.8}$$

Knowing the recoveries for each component, we can calculate the feasibility of using gas absorption.

E. EXSEP Program Development and Control

We now understand how EXSEP uses tools such as the component assignment matrix, shortcut feasibility analysis, and heuristic operational bounds to synthesize processes using absorption. Let us now investigate how the ABSORB module is developed and controlled.

1. Acknowledgements

I would first like to acknowledge Mr. David Schenk (now at Shell Oil) and Mr. Jean Christian Brunet for their substantial contributions to the Prolog programming of this module. Their contributions are sincerely appreciated.

2. Program Structure and Control

The ABSORB module is called through one clause, **absorb**. The syntax for the **absorb** clause is:

absorb(List, CList, TopList, TopCList, BotList, BotCList)

The information flow in the absorb clause is:

absorb(*input*, *input*, *output*, *output*, *output*, *output***).**

The following describes what each variable represents:

- **List** - the list of products in the CAM,
  e.g., **[p1, p2, p3, solvent]**.
- **CList** - the list of components in the CAM,
  e.g., **[c3, c4, c5, h2s, mea]**.
- **TopList** - the list of products in the overhead CAM,

e.g., **[p1, p2]**

- **TopCList** - the list of components in the overhead CAM,

  e.g., **[c3, c4, c5]**

- **BotList** - the list of products in the bottom CAM,

  e.g., **[p3, solvent]**

- **BotCList** - the list of components in the bottom CAM,

  e.g., **[c4, c5, h2s, mea]**

The **absorb** clause is an "umbrella" clause for the entire module. It performs necessary preparatory steps and coordinates the entire module. The steps undertaken by the absorb clause are shown in Figure D.4.

The **absorb** clause inputs necessary solvent and Henry's law constants. It then calculates a preliminary solvent flow rate based on the lower-bound heuristic, $L/k_iV = 1.3$, where i is the key component (fixed component) in the process. Also, the **absorb** clause appends **List** and **CList** with the solvent to give the new CAM, identified by **LList** and **CCList**. For example, in our initial H_2S absorption problem, **List** and **CList** are:

```
List =  [ p1, p2, p3].
CList = [h2s, c3, c4, c5].
```

After we add solvent (MEA solution), **List** and **CList** become **LList** and **CCList**, respectively:

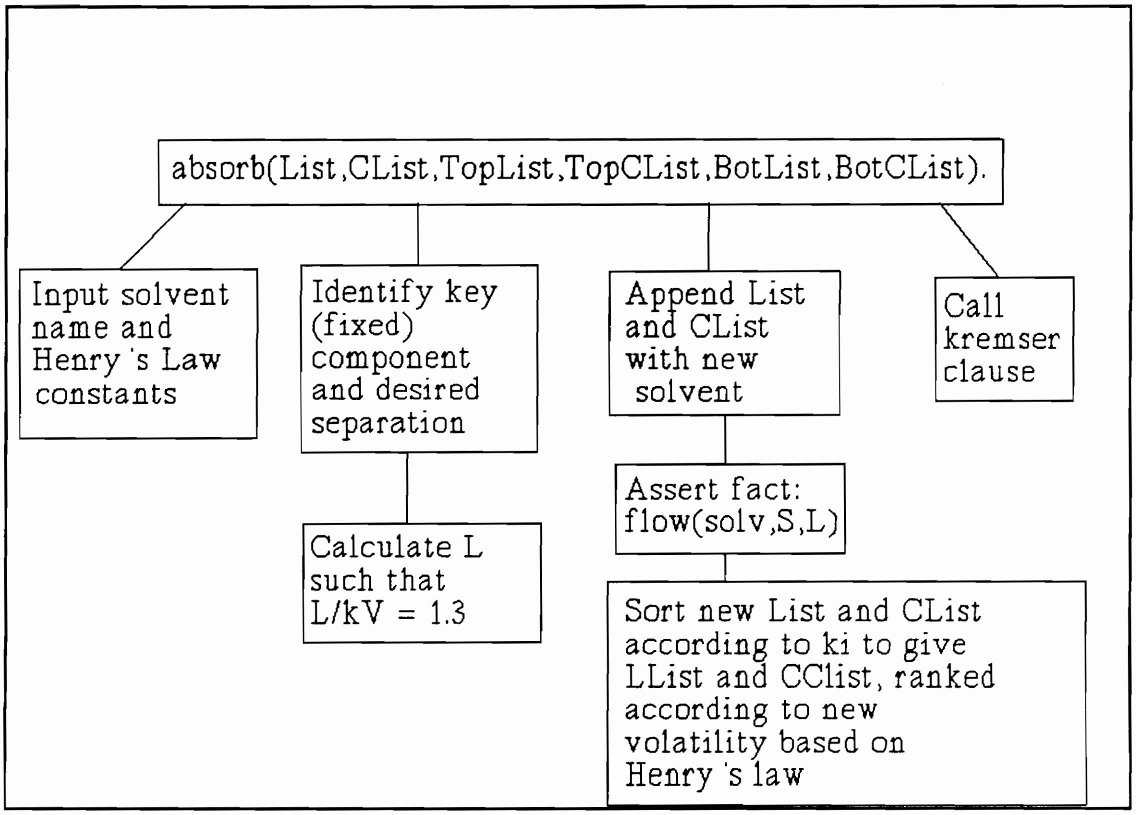

absorb(List,CList,TopList,TopCList,BotList,BotCList).
Input solvent name and Henry 's Law constants
Identify key (fixed) component and desired separation
Calculate L such that L/kV = 1.3
Append List and CList with new solvent
Assert fact: flow(solv,S,L)
Sort new List and CList according to ki to give LList and CClist, ranked according to new volatility based on Henry 's law
Call kremser clause

Figure D.4. Actions of the absorb clause.

```
LList =  [ pl, p2, p3, solvent].
CCList = [ h2S, c3, c4, c5, mea].
```

Now the **absorb** module sorts the lists based on the Henry's law constants to make **LList** and **CCList** ranked according to volatility. The **quicksort** procedure is used, and the result is:

```
LList =  [ p2, p3, pl, solvent].
CCList = [c3, c4, c5, h2s, mea].
```

We now have a new CAM, based on the new volatilities created by the presence of MEA.

After forming the new CAM and developing the new ranking for the products and components, the **absorb** module's job is finished. It passes responsibility to the **kremser** clause, the second major clause in the ABSORB module. The job of the **kremser** clause is to find a heuristically acceptable operating condition where the separation specifications can be attained.

The syntax for the **kremser** clause is:

kremser(LList, CCList, [S], L, V, FConst, TopList, BotList)

The information flow in the absorb clause is:

kremser(*input*, *input*, *input*, *input*, *input*, *input*, *input*, *input*).

The following describes what each variable represents:

- **LList** - the list of products in the CAM,

  e.g., **[p2, p3, pl, solvent]**.
- **CCList** - the list of components in the CAM,

  e.g., **[c3, c4, c5, h2s, mea]**.
- **[S]** - the solvent in list form, e.g., **[mea]**
- **L** - the liquid flow rate to the column

- **V** - the vapor flow rate to the column
- **FConst** - the Henry's law constant, k_i, for the fixed (key) component
- **TopList** - the list of desired overhead products

  e.g., **[p2, p3]**
- **BotList** - the list of desired bottoms products

  e.g., **[p1, solvent]**

As mentioned, the **kremser** clause attempts to find operating conditions that achieve the separation specifications and are heuristically acceptable. The clause accomplishes this goal by continually incrementing L upwards from the lower-bound value of $L/k_iV = 1.3$. After each increment, **kremser** calculates: 1) the value of N that achieves the desired key-component recovery, and 2) all nonkey component distributions. These nonkey component distributions are compared with specifications. If $(d/b)_{calculated}$ is within ± 10% of $(d/b)_{desired}$, the separation is feasible.

The logic for the **kremser** clause is shown in Figure D.5. If the clause is unable to find an acceptable operating condition that achieves the product objectives, the clause fails. This fail induces backtracking (ultimately) to the **absorb** clause, which will also fail if **kremser** fails. Thus, if no acceptable operating conditions exist, the entire **absorb** module will fail, and no acceptable splits are generated. EXSEP backtracks from the absorb module and looks for alternate separation methods.

Note, however, that before the **kremser** equation fails, the results

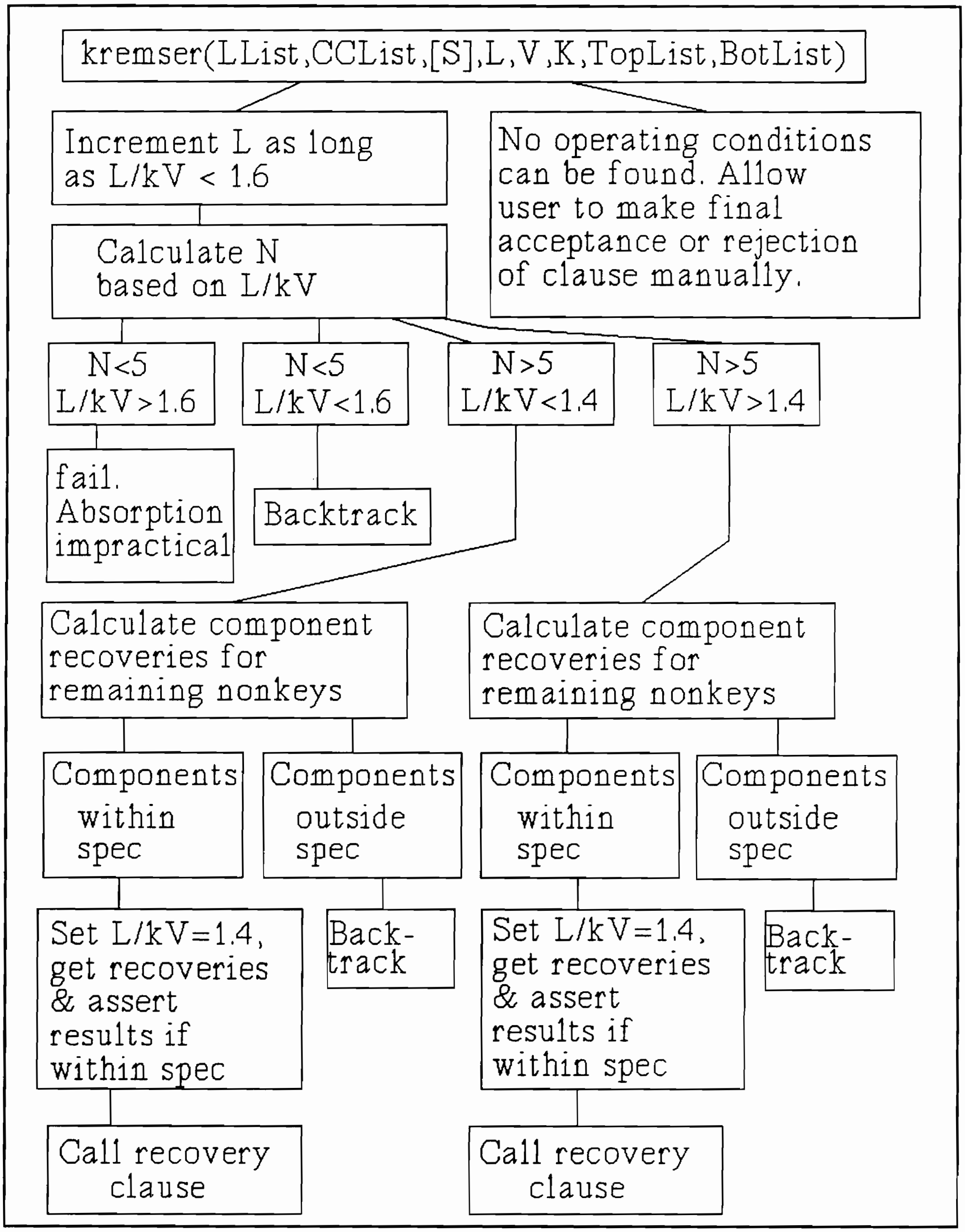

kremser(LList,CCList,[S],L,V,K,TopList,BotList)
Increment L as long as L/kV < 1.6
No operating conditions can be found. Allow user to make final acceptance or rejection of clause manually.
Calculate N based on L/kV
N<5 L/kV>1.6
N<5 L/kV<1.6
N>5 L/kV<1.4
N>5 L/kV>1.4
fail. Absorption impractical
Backtrack
Calculate component recoveries for remaining nonkeys
Calculate component recoveries for remaining nonkeys
Components within spec
Components outside spec
Components within spec
Components outside spec
Set L/kV=1.4, get recoveries & assert results if within spec
Back-track
Set L/kV=1.4, get recoveries & assert results if within spec
Back-track
Call recovery clause
Call recovery clause

Figure 5. The actions of the kremser clause.

are posed to the user, who is then asked to accept or reject the separation(s) that the module has generated. If the user accepts the separation, the cut is made and the program continues to execute. If the user rejects the separation, the module fails and the program backtracks in search of alternate separation methods.

The last key clause in the **absorb** module is called by **kremser**, and is the **recovery** clause. The **recovery** clause processes the absorber bottoms to recover solvent. Thus, the recovery clause separates the solvent from the absorbed components using a stripper. The recovery clause assumes that stripping is feasible, and that a sharp split between the solvent and additional components is attainable. This assumption is well-founded, since most stripping operations are very efficient.

D.4. EXAMPLE OF THE EXSEP ABSORB MODULE

A. The Problem: Lean-Oil Absorption

Let us consider a hydrogen-rich feed stream containing light hydrocarbons for use as gasoline (Nelson, 1969). The stream has the following composition:

Component	Mole Percent
H_2	85.59
C_3H_8	7.15
i-C_4H_{10}	1.39
n-C_4H_{10}	2.55
i-C_5H_{12}	1.34
n-C_5H_{12}	1.98

The total flow rate of the stream is 100 mol/hr. We wish to achieve the following flow rates:

Component	Vapor Overhead (mol/hr)	Liquid Bottoms (mol/hr)
H_2	85.59	0
C_3H_8	5.72	1.43
i-C_4H_{10}	0.751	0.639
n-C_4H_{10}	0.994	1.556
i-C_5H_{12}	0.013	1.327
n-C_5H_{12}	0	1.98

The key component is $i\text{-}C_5H_{12}$. We wish to recover 95-99% of this component in the bottoms. The feed stream is at 90 °F and 50 psia. With lean-oil as the solvent (MW = 160, specific gravity = 0.83), the Henry's law constants for the components are:

Component	Henry's law constant
H_2	50
C_3H_8	2.8
iC_4H_{10}	1.2
nC_4H_{10}	0.9
iC_5H_{12}	0.37
nC_5H_{12}	0.24

The CAM in the absence of solvent is:

$$\begin{array}{c|cccccc|}
 & H2 & C3 & iC4 & nC4 & iC5 & nC5 \\
p1 & 0 & 1.43 & 0.639 & 1.556 & 1.327 & 1.98 \\
p2 & 85.59 & 5.72 & 0.751 & 0.994 & 0.013 & 0
\end{array}$$

We add lean oil to the system, and the new CAM becomes:

$$\begin{array}{c|ccccccc|}
 & H2 & C3 & iC4 & nC4 & iC5 & nC5 & Oil \\
solvent & 0 & 0 & 0 & 0 & 0 & 0 & L \\
p1 & 0 & 1.43 & 0.639 & 1.556 & 1.327 & 1.98 & 0 \\
p2 & 85.59 & 5.72 & 0.751 & 0.994 & 0.013 & 0 & 0
\end{array}$$

A number of questions now arise. Is the pl/p2 split feasible using absorption? What is the optimal liquid flow rate? How many stages should the absorber have? EXSEP will answer all of these preliminary design equations. EXSEP's reasoning mechanism goes to work and finds operating conditions that achieve the material balance objectives within a d/b of ± 10%. These operating conditions are contrasted with Nelson's results, as shown in Table D.3.

Table D.3. Comparison of EXSEP's Shortcut Results with Nelson (1969).

Parameter	Nelson	EXSEP
A	1.5	1.45
L	55.5	53.65
N	8	9.2

The absorption factor, A = L/kV = 1.45, falls within the heuristic bounds:

$$\textit{RULE}\ 1: \quad 1.3 \le \frac{L}{k_i\ V} \le 1.6 \qquad (12)$$

Likewise, N, the number of theoretical stages also fulfills the minimum:

$$\textit{RULE}\ 2: \quad N > 5 \qquad (13)$$

The desired recoveries as calculated by EXSEP are all within ± 10% of the desired d/b. Note that EXSEP achieves its results with a *lower* solvent flow rate than that of Nelson.

B. Testing of Shortcut Methodology with Rigorous Simulation

We may test EXSEP's results with a rigorous absorption model to determine the accuracy of our shortcut method. Let us compare the shortcut methodology with the rigorous simulation. We use DESIGN II by ChemShare Corp., Houston, TX, for our rigorous absorption simulation. The input and output for shortcut and rigorous simulations is shown in Table D.4.

Table D.4. Input and Output for Rigorous and Shortcut Calculations.

<table>
<tr><th colspan="2">SHORTCUT ABSORPTION: KREMSER EQUATION</th></tr>
<tr><th>INPUT</th><th>OUTPUT</th></tr>
<tr><td>Feed Composition, $y_{i,\ in}$</td><td>Solvent flow rate, L</td></tr>
<tr><td>Overhead recovery of key</td><td>Number of stages, N</td></tr>
<tr><td>Bottoms recovery of key</td><td>Nonkey component recoveries</td></tr>
<tr><th colspan="2">RIGOROUS SIMULATION</th></tr>
<tr><th>INPUT</th><th>OUTPUT</th></tr>
<tr><td>Feed composition, $y_{i,\ in}$</td><td>Distillate composition, $y_{i,\ out}$</td></tr>
<tr><td>Number of stages, N</td><td>Bottom composition, $x_{i,\ out}$</td></tr>
<tr><td>Solvent flow rate, L</td><td>Overhead and bottoms flow rates</td></tr>
</table>

From this table, we see that the rigorous simulation tests the accuracy of nonkey component distributions as estimated by the Kremser equation. The comparison of shortcut versus rigorous results is shown in Table D.5.

Table D.5. Component Recoveries for Rigorous and Shortcut Calculations.

		SHORTCUT		RIGOROUS	
Component	Vapor Feed (mol/hr)	Vapor Product (mol/hr)	Percent Absorbed	Vapor Product (mol/hr)	Percent Absorbed
H2	85.59	85.59	0 %	85.47	0.13 %
C3	7.15	5.72	20 %	6.06	15.3 %
i-C4	1.39	0.75	46 %	0.90	35.1 %
n-C4	2.55	0.99	61 %	1.24	51.3 %
i-C5	1.34	1.3×10^{-2}	99 %	3.4×10^{-2}	97.5 %
n-C5	1.98	0	100 %	4.6×10^{-3}	99.8 %
Oil	0	0	---	0.115	---

From these results, we see that the Kremser equation *is a suitable shortcut feasibility tool for absorption*. Nevertheless, some care must be exercised when using the Kremser equation. When we use equilibrium ratios based on feed-conditions, the Kremser equation tends to overpredict the amount of material that will be absorbed. According to the shortcut analysis, we achieve 99% recovery of i-C5 in the bottoms at L = 53.65 and N = 9.2. When we perform a rigorous simulation under these same conditions, we achieve only a 99.7 % recovery of i-C5. While this difference is virtually inconsequential for the current design, it may play a more important role in other problems.

The fact that the Kremser equation overpredicts the amount of material absorbed may have consequences. These consequences depend on whether we are focusing on the key component or the nonkey component(s).

(1) *Potential Consequence #1: Key Component Concerns.* The Kremser equation overpredicts the amount of key component absorbed into the solvent. Therefore, the shortcut model predicts better column performance than what we would see rigorously. Thus, what is feasible as predicted with the Kremser equation may not necessarily be feasible rigorously. The Kremser equation gave 99% recovery of i-C5 with 9.2 theoretical stages. Rigorously, the recovery is 97.5%. To achieve 99% recovery of i-C5, we need 14 theoretical stages.

If we use the Kremser equation for shortcut calculations, we should be conservative and place in a safety factor on the number of stages required for desired recovery.

(2) *Potential Consequence #2: Nonkey Component Concerns.* The Kremser equation also overpredicts the amount of nonkey components absorbed into the solvent. In most absorption problems, we prefer to have the nonkey components remain in the vapor with *minimal* partitioning to the solvent. Therefore, with respect to nonkey components, the Kremser equation is a conservative model. We desire minimal partitioning of vapor-phase components into the solvent; the Kremser equation overpredicts this partitioning. Thus, if the partitioning is acceptable with the Kremser equation, it will be acceptable under

real conditions where there is less partitioning.

C. Understanding the Kremser Equation

We wish to determine *why* the Kremser Equation overpredicts component-recoveries in the solvent and where we must be cautious in using this shortcut tool. Let us review again the assumptions underlying the Kremser equation:

- Constant molal overflow (L/V is constant)
- Isothermal system
- Isobaric system
- Negligible heats of absorption
- Straight equilibrium line (i.e., Henry's law applies)

Now let us determine how well these assumptions stood up under the rigorous simulation of the lean-oil absorption problem. There are two primary concerns with respect to the assumptions: the validity of constant L/V, and the operation of the column isothermally.

Let us compare the L/V of the Kremser Equation versus the rigorous simulation. A graph of L/V in a stage-by-stage calculation is shown in Figure D.6. From this graph, we see that L/V is indeed constant throughout the column, and therefore, this assumption of the Kremser equation is valid. The L/V concern does not appear to be the reason for the Kremser equation's behavior.

Now, let us assess the temperature profile through the column. The Kremser equation assumes isothermal behavior. The actual temperature profile, as obtained from rigorous simulation using DESIGN II, is shown in Figure D.7. From this graph, we see that the column *does not* behave isothermally. The temperature at the top of the column is lower than at the bottom. This hydrocarbon-oil system is as near to ideal as we can get, yet the temperature still rises 15 °F above the feed-temperature. The consequence of this temperature rise is that *less vapor is partitioned into the solvent*, and hence, real-life recoveries of the key component in the solvent are *lower* than what is predicted by the isothermal Kremser equation.

Intuitively, we realize that as the temperature of the solvent rises, thermodynamic equilibrium favors the migration of absorbed gaseous components into the vapor. Diab and Maddox (1982), gives the values for k_i for methane, ethane, propane, and n-butane in benzene as a function of temperature. These results are calculated from the Redlich-Kwong-Soave equation of state, and are shown in Figure D.8.

From these curves, we see that nonisothermality can be a great source of concern. A 20.4% change in absolute temperature (temperature change from 0 °F to 100 °F, or $\Delta T = 55.6$ K) induces a 48.6% change in K_{C1} and a 220% change in K_{C4}.

We can conclude that temperature is a great source of difference between the Kremser equation and real-conditions. Some researchers (Diab and Maddox, 1982) have recommended taking the nonisothermality into account by choosing and average K-value across the absorber. This approach has two practical difficulties associated with it:

(1) In preliminary design, we do not know what the adiabatic temperature-rise in the column will be; that analysis occurs in the rigorous design stage. Thus it may be premature to provide an average K-value based on the adiabatic temperature rise.

(2) By incorporating an average K-value based on the adiabatic temperature rise, we are imposing an iterative solution on the shortcut design equations. If we must resort to an iterative solution anyway, we may as well forget the shortcut calculation and perform a rigorous simulation from the beginning.

Thus, we may conclude that if the solution thermodynamics are highly non-ideal (i.e., $\Delta H_{mixing} << 0$), we must be cautious with the Kremser equation, since isothermal operation is assumed. In addition, we must make sure we place in a safety factor by increasing the number of stages in the column.

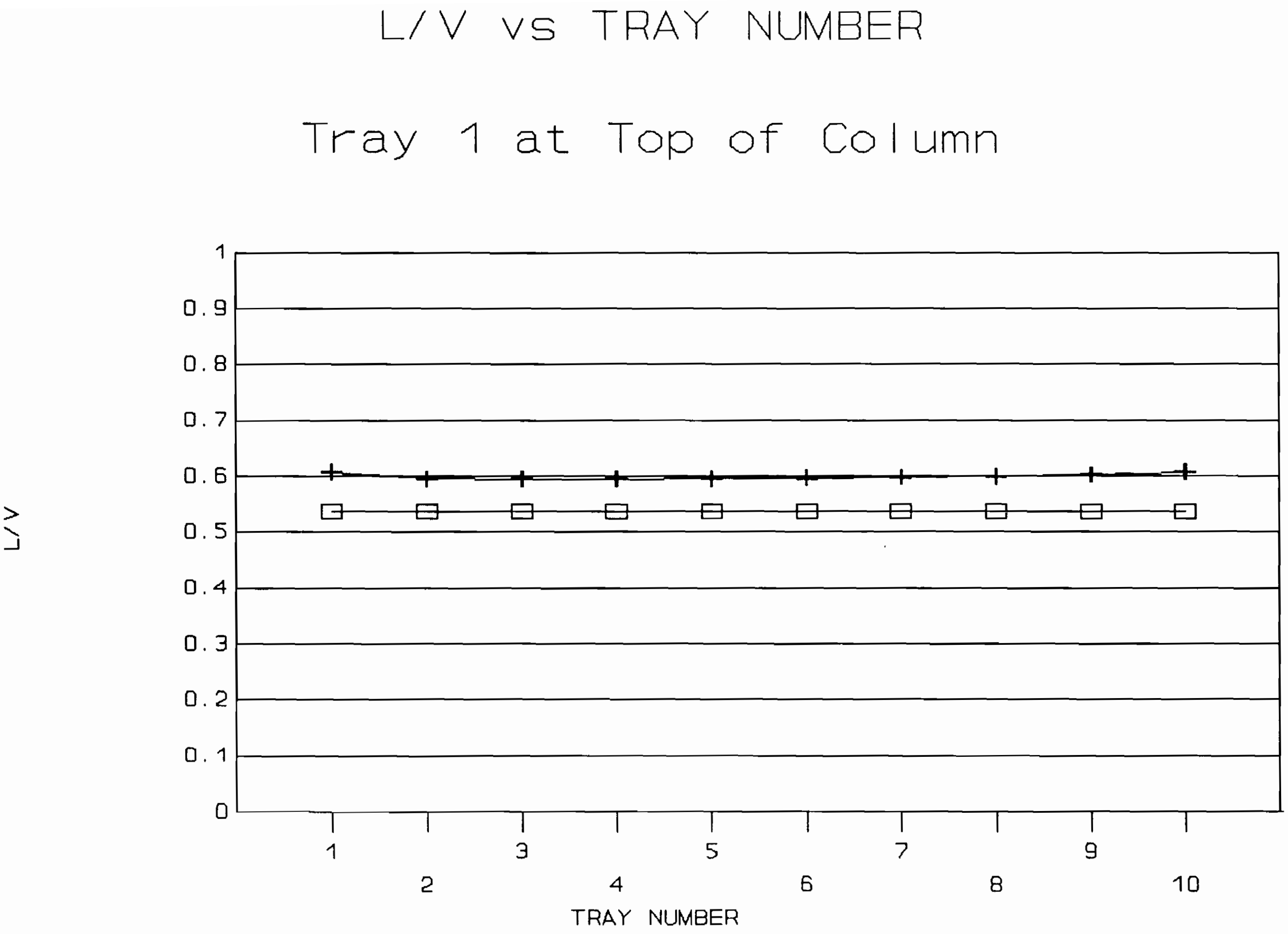


Figure D.6. L/V vs. stage number. The +-+-+ is rigorous, and □-□-□ is Kremser.

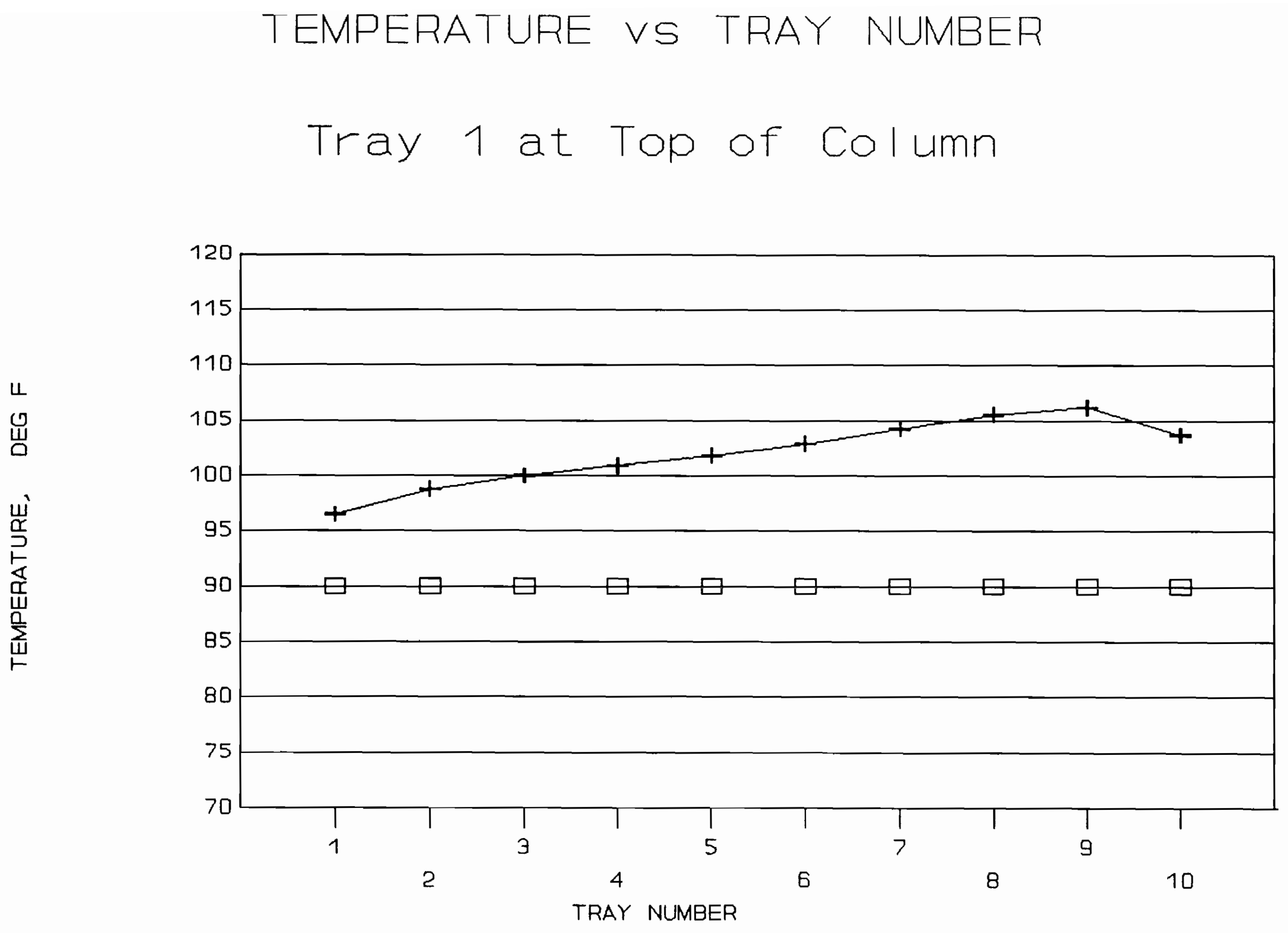


Figure D.7. Temperature (°F) vs. stage number. The +-+-+ is rigorous, and □-□-□ is Kremser.

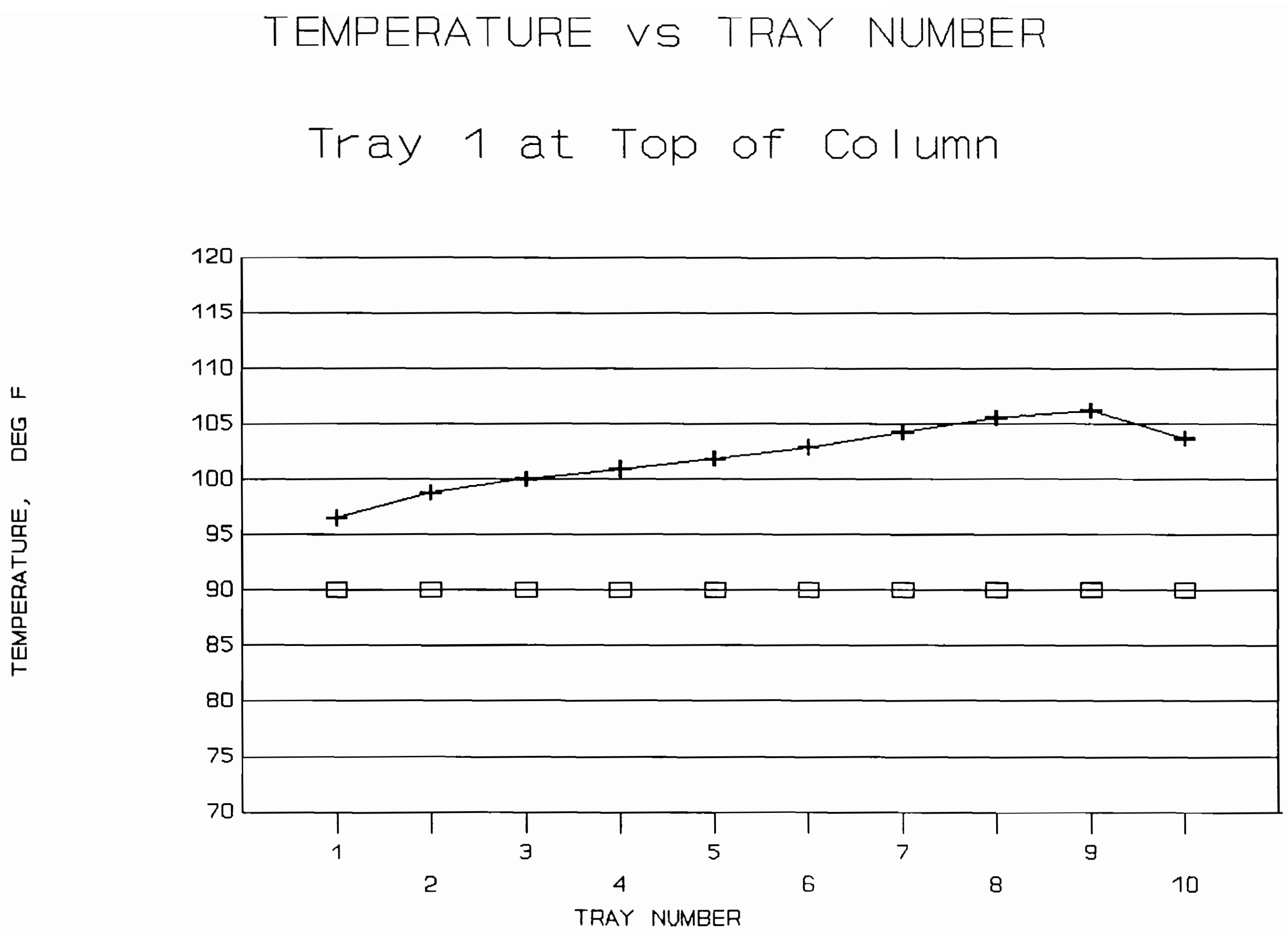


Figure D.8. K-values vs. temperature (°F).
Legend: the +-+-+ is methane, □-□-□ is ethane, X-X-X is propane, and *-*-* is n-butane.

D.4 CONCLUSIONS AND FUTURE DIRECTIONS

EXSEP is the first expert system ever successfully implemented that can perform multicomponent, computer-aided separation process synthesis. Other researchers have attempted knowledge-based separation process synthesis, but have not been particularly successful. For the most part, their representation of the problem was not sufficient for successful implementation. For one, no other researcher has used any feasibility analysis test, let alone a shortcut calculation that EXSEP uses. EXSEP possesses a knowledge-representation that solves problems accurately and efficiently. Indeed, that is our emphasis in EXSEP: rather than focus on computer-science tools, *we have expanded the usefulness of shortcut analysis in multicomponent separation synthesis*. We feel that this emphasis will greatly advance the industrial practice of computer-aided process synthesis.

The development of shortcut design techniques is a key need in chemical engineering design. The Amundson report, i.e., *Frontiers in Chemical Engineering* (1988), specifically identifies this need. The incorporation of shortcut design into knowledge-based systems is an excellent combination -- it enables engineers to generate feasible process alternatives quickly and efficiently, before moving to the laborious rigorous design stage. We hope that EXSEP's combination of heuristic analysis plus shortcut design will fulfill this need.

EXSEP has been developed modularly, and because it is written in

Prolog, it can be adjusted relatively easily. We feel that using EXSEP's approach to process synthesis, additional separation beyond absorption and ordinary distillation can be continually added to the system.

With both the ordinary distillation and absorption modules operational, there are a number of available routes open to EXSEP. Since the absorption module uses the Kremser equation, and this equation is applicable to a wide range of separation operations (Table D.2), we may opportunistically expand EXSEP into other MSA processes, such as liquid-liquid extraction.

Another potential area of expansion for EXSEP is the "umbrella" reasoning procedure that performs separation method selection. Barnicki and Fair (1990) have done an excellent job in knowledge representation for this task, and perhaps their cooperation would be useful.

EXSEP's main strength, however, is its unique plan-generate-test strategy that couples the component assignment matrix, shortcut feasibility, and heuristic process synthesis. We feel that the most dividends for future work will result from emphasizing this strength and expanding EXSEP to other separation methods using shortcut tools.

EXSEP is already a practical evolutionary synthesis tool, but it can be further developed to include additional aspects of evolutionary synthesis as well as retrofit design. In an era of ever-increasing energy costs, systematic retrofit design is key need in the chemical engineering community.

D.5 ABSORPTION MODULE PROGRAM LISTING

The absorb module program listing is shown below.

ABSORB MODULE

```
project "exsep"

global domains
        alpha = symbol
        klist = alpha*
        number = integer
        value = real
        vlist = value*
        str = string

/*                                              *
 *    Global domains for Toolbox Predicates     *
 *                                              */
  ROW, COL, LEN, ATTR   = INTEGER
  STRINGLIST = STRING*
  INTEGERLIST = INTEGER*
  KEY    = cr; esc; break; tab; btab; del; bdel; ctrlbdel; ins;
           end ; home ; fkey(INTEGER) ; up ; down ; left ; right ;
           ctrlleft; ctrlright; ctrlend; ctrlhome; pgup; pgdn;
           ctrlpgup; ctrlpgdn; char(CHAR) ; otherspec


global database - bypass
        dbypass_amount(klist,alpha,value)
        dbypass_result(klist,alpha)
        dypass_status(alpha)

global database - materials
        flow(alpha,alpha,value)
        k_value(alpha,value)
        boiling_temp(alpha,value)
        initial_components(klist)
        initial_set(klist)
        corrosive(alpha)
        pseudoproduct(klist,alpha)

global database - table
        dsst(klist,klist,alpha,klist,klist,alpha,alpha,value,vlist,vlist,alpha,alpha,value)
        ddsst(klist,klist,alpha,klist,klist,alpha,alpha,value,vlist,vlist,alpha,alpha,value)
        dddsst(klist,klist,alpha,klist,klist,alpha,alpha,value,vlist,vlist,alpha,alpha,value)
        ddddsst(klist,klist,alpha,klist,klist,alpha,alpha,value,vlist,vlist,alpha,alpha,value)
        dddddsst(klist,klist,alpha,klist,klist,alpha,alpha,value,vlist,vlist,alpha,alpha,value)
        process_keys(alpha,alpha)
        process_product(alpha)
        sloppy_keys(alpha,alpha)


global database - sequence
        dseparator(number,klist,vlist,klist,vlist)
        dstreamin(number,klist,vlist)
        dstreambypass(number,alpha,klist,vlist)

global predicates
        append_v(vlist,vlist,vlist) - (i,i,o)
        bypass(klist,klist) - (i,i)
        component_flow(klist,alpha,value) - (i,i,o)
        component_flow_list(klist,vlist,klist,klist) - (i,i,i,o)
        component_flow_set(klist,klist,vlist) - (i,i,o)
        decide_difficult_or_easy(value,alpha) - (i,o)
        delete_element(alpha,klist,klist) - (i,i,o)
        delete_elements(klist,klist,klist) - (i,i,o)
        delete_number(value,vlist,vlist) - (i,i,o)
        equal(klist,klist) - (i,i)
        ffactor(value,value,value) - (i,i,o)
        first_prod(klist,klist) - (i,o)
        flow_set(alpha,klist,vlist) - (i,i,o)
        flow_set_c(alpha,klist,vlist) - (i,i,o)
        get_split_flow(klist,value,klist,value) - (i,o,i,i)
        largest_flow(klist,alpha,klist) - (i,o,i)
        last_prod(klist,klist) - (i,o)
        length(klist,value) - (i,o)
        list_member(klist,klist) - (i,i)
        max(value,value,value) - (i,i,o)
        max(number,number,number) - (i,i,o)
        max_flist(vlist,value) - (i,o)
        max_flow(value,value,value) - (i,i,o)
        member(alpha,klist) - (i,i)
        pause(number) - (i)
        pause_escape
        positive_component_flow_list(klist,klist) - (i,o)
        product_flow(alpha,value,klist) - (i,o,i)
        product_flow_match(klist,value,alpha,klist) - (i,i,o,i)
        product_flow_set(klist,vlist,klist) - (i,o,i)
        reverse(klist,klist) - (i,o)
        reverse_v(vlist,vlist) - (i,o)
        selected_positive_component_flow_list(klist,klist,klist) - (i,i,o)
        separation_specification_table(klist,klist) - (i,i)
        set_above(klist,alpha,klist) - (i,i,o)
        set_below(klist,alpha,klist) - (i,i,o)
        split(klist,klist,klist,klist,klist,klist) - (i,i,o,o,o,o)
        split_above(klist,alpha,klist,klist) - (i,i,o,o)
        split_below(klist,alpha,klist,klist) - (i,i,o,o)
        sub_list(klist,klist) - (i,i)
        sum_component_flow(klist,value,klist) - (i,o,i)
        sum_flist(vlist,value) - (i,o)
        two_highest_flows(vlist,value,value) - (i,o,o)

database
        insmode
        lineinpstate(string,integer)
        lineinpflag
        henry(alpha,value)
        x_in_solvent(alpha,value)

include "tpreds.pro"
include "menu.pro"
include "lineinp.pro"
include "filename.pro"

predicates
        absorb(klist,klist,klist,klist,klist,klist)
        adjust_hl(value,value,value)
        append(klist,klist,klist)
        assert_comp_flow_in_solvent(klist,klist,value)
        assert_no_solvent_in_product(klist,alpha)
        build_list(klist,klist,klist,klist)
        calc_Component_d_over_b(klist,klist,value,value,value,klist,klist,vlist,vlist)
        calc_Yi_out(value,value,value,value)
        check_every(klist,klist,value,value,value,value,value,klist,klist,vlist,vlist)
        check_FC_in_solvent(alpha)
        check_in_spec(klist,klist,value,value,value,value,value,value,klist,klist,vlist,vlist)
        check_member(alpha,klist)
        check_N(value)
        check_one_point_four(value,value,value,value)
        components_within_spec(klist,klist,value,value,value,klist,klist,vlist,vlist)
        conc_v(vlist,vlist,vlist)
        contained(integer,alpha)
        d_over_b(value,value,value)
        equal_type(stringlist,klist)
        essai
        flow_setP(alpha,klist,vlist)
        henry_prod_flow(alpha,value,klist)
        identify_desired_separation(klist,klist,klist,klist)
        input_henrys_law(klist)
        kremser(klist,klist,klist,value,value,value,klist,klist)
        make_henlist_from_list(klist,klist,vlist)
        make_hlist_from_clist(klist,vlist)
        print_list(klist)
        print_vlist(vlist)
        product_flow_help(alpha,value,klist,value)
        quicksort_dec(vlist,vlist)
        recovery(klist,klist,klist)
        repeat_and_increment(value,value,value,value)
        resolve_contain(integer,klist)
        resolve_list_choice(value,klist,integer,alpha)
        resolve_user_choice(integer)
        split_v_dec(value,vlist,vlist,vlist)
        set_L_over_KV_one_point_four(value,value,value)
        sort_a_list_by_hhlist(klist,vlist,klist)

clauses

absorb(List,CList,TopList,TopCList,BotList,BotCList):-
        makewindow(1,30,30,"Knowledge-Based Separation Sequencing",1,1,24,79),
        makewindow(2,113,113,"Component Assignment Matrix",2,1,23,79),
        makewindow(7,14,14,"",21,45,4,35),
        makewindow(10,241,64,"",15,10,3,60),
        shiftwindow(1),
        consult("raf.dat",materials),
        consult("hraf.dat"),
        asserta(dseparator(0,[],[],[],[]),sequence),
        clearwindow,
        trace(off),
        initial_set(List),
        initial_components(CList),
        get_split_flow(List,V,CList,0),
        clearwindow,
        write("Absorption appears possible for this CAM."),nl,
        write("Please enter name of solvent:"),
        readln(S),nl,
        write("/Does the solvent contain any of the problem components/?"),nl,
        menu(25,23,14,14,["Yes","No"],"Answer",2,Choice_contain),
        trace(off),
        resolve_contain(Choice_contain,CList),
        assert_no_solvent_in_product(List,S),
        write("Please enter Henry's Law constants for components."),nl,
        input_henrys_law(CList),*/
```

```
            trace(off),
            make_hlist_from_clist(CList,Hlist),
            quicksort_dec(HList,HHCList),
            sort_a_list_by_hhlist(CList,HHCList,CLList),
            make_henlist_from_list(List,CList,HL),
            quicksort_dec(HL,HHL),
            sort_a_list_by_hhlist(List,HHL,LList),
            append(LList,["solv"],NList),
            append(CLList,[S],NCList),
            print_cam_driver(LList,CLList,_),
            trace(off),
            shiftwindow(2),
            identify_desired_separation(NList,NCList,TopList,BotList),/*BotList contains solvent*/
            shiftwindow(1),
            nl,nl,
            write("You may choose a fixed component (FC):"),nl,
            length(CList,LL),
            equal_type(CLst,CList),
            menu(7,50,3,7,CLst,"Choose FC",LL,FC_Choice),
            resolve_list_choice(FC_Choice,CList,LL,FC),
            shiftwindow(1),
    /*      write("Input fixed component from the following list:\n",CList),
            readln(FC),nl,*/
            check_member(FC,CLList),
            trace(off),
            check_FC_in_solvent(FC),
            trace(off),
            henry(FC,FConst),
            Old_L = 1.3 * FConst * V,
            repeat_and_increment(V,FConst,Old_L,L),
            trace(off),
            kremser(LList,CLList,[S],L,V,FConst,TopList,BotList),
            dseparator(Num,_,_,_,_), /*BotList contains solvent          */
            NNum = Num + 1,          /*TopList doesn't                    */
            trace(off),              /*LList & CLList don't  contain S    */
            component_flow_set(TopList,NCList,TopFList),
            component_flow_set(BotList,NCList,BotFList),
            trace(off),
            asserta(dseparator(NNum,NCList,TopFList,NCList,BotFList),sequence),
            retractall(henry(_,_)),
            shiftwindow(10),
            clearwindow,
            write("The results for the ABSORPTION  COL. #",NNum," are now available"),
            pause(N),
            trace(on),
            print_cam_driver(NList,NCList,_).
```

```
resolve_contain(2,[]).
resolve_contain(2,[HC|TC]):-
        assertz(x_in_solvent(HC,0)),
        resolve_contain(2,TC).

resolve_contain(1,[]).
resolve_contain(1,[HC|TC]):-
        write("Does it contain ",HC," ?:"),nl,
        trace(off),
        menu(28,23,14,14,["Yes","No"],"Answer",2,Choice2),
        trace(off),
        contained(Choice2,HC),
        resolve_contain(1,TC).

contained(1,HC):-
        write("Enter the fraction (molar or mass) of ",HC," in the solvent:"),
        readreal(XC),
        assertz(x_in_solvent(HC,XC)).

contained(2,HC):-
        assertz(x_in_solvent(HC,0)).

assert_no_solvent_in_product([],S).
assert_no_solvent_in_product([HL|TL],S):-
        assertz(flow(HL,S,0),materials),
        assert_no_solvent_in_product(TL,S).

input_henrys_law([]).
input_henrys_law([Head|Tail]):-
        write("Input Henry's Law constant for ",Head,":\n"),
        readreal(HVal),
        assertz(henry(Head,HVal)),
        input_henrys_law(Tail).

make_hlist_from_clist([],[]).
make_hlist_from_clist([HC|TC],[HH|TH]):-
            henry(HC,HH),
            make_hlist_from_clist(TC,TH).

sort_a_list_by_hhlist(_,[],[]).
sort_a_list_by_hhlist(List,[HH|TH],[HL|TL]):-
        henry(HL,HH),
        member(HL,List),
        sort_a_list_by_hhlist(List,TH,TL).

make_henlist_from_list([],_,[]).
make_henlist_from_list([HL|TL],CList,[HH|TH]):-
        product_flow(HL,PFlow,CList),
        henry_prod_flow(HL,HPFlow,CList),
        adjust_hl(HPflow,PFlow,HH),
        assertz(henry(HL,HH)),
        make_henlist_from_list(TL,CList,TH).

henry_prod_flow(Head,HPflow,CList):-
        flow_setP(Head,CList,HFlowList),
        sum_flist(HFlowList,HPFlow).

flow_setP(_,[],[]).
flow_setP(Head,[HC|TC],[HHF|THF]):-
        flow(Head,HC,Flow),
        henry(HC,Hval),
        HHF = Hval * Flow,
        flow_setP(Head,TC,THF).

adjust_hl(0,_,0).
adjust_hl(Pi2List,PiList,HlAdj):-
        HlAdj = Pi2List / PiList.

quicksort_dec([],[]).
quicksort_dec([X|Tail],Sorted):-
        split_v_dec(X,Tail,Small,Big),
        quicksort_dec(Small,SortedSmall),
        quicksort_dec(Big,SortedBig),
        conc_v(SortedSmall,[X|SortedBig],Sorted).

split_v_dec(X,[],[],[]).
split_v_dec(X,[Y|Tail],[Y|Small],Big):-
        X < Y,!,
        split_v_dec(X,Tail,Small,Big).
split_v_dec(X,[Y|Tail],Small,[Y|Big]):-
        split_v_dec(X,Tail,Small,Big).

conc_v([],L,L).
conc_v([X|L1],L2,[X|L3]):-
        conc_v(L1,L2,L3).

identify_desired_separation(NList,NCList,TopList,BotList):-
        write("Enter the product above which you want the split:"),
        readln(Prod),nl,
        split_above(NList,Prod,TopList,BotList).

equal_type([],[]).
equal_type([H¦T],[HC¦TC]):-
        H=HC,
        equal_type(T,TC).

resolve_list_choice(Choice,[HC¦TC],LL,FC):-
        length([HC¦TC],K),K+Choice=LL+1,FC=HC.
resolve_list_choice(Choice,[HC¦TC],LL,FC):-
        resolve_list_choice(Choice,TC,LL,FC),!.

repeat_and_increment(_,_,X,X).
repeat_and_increment(V,FConst,X,NX):-
        trace(off),
        retractall(flow("solv",_,_),materials),
        NNX = X + 0.025 * FConst * V,
        repeat_and_increment(V,FConst,NNX,NX).

kremser(List,CList,[S],L,V,FConst,TopList,BotList):-
        trace(off),
/*      write("V=",V),nl,write("L=",L),nl,*/
        henry(FC,FConst),
        L <= 1.6 * FConst * V,
        A = L/(FConst * V),
/*      write("A=",A),nl,*/
        trace(off),
        component_flow(List,FC,FCFlow),/*don't need flow in solvent for Y_in*/
/*      write("FCFlow=",FCFlow),nl,*/
        Yi_in = FCFlow / V,
/*      write("Yi_in=",Yi_in),nl,*/
        component_flow(TopList,FC,FCTFlow),
/*      write("FCTFlow=",FCTFlow),nl,*/
        sum_component_flow(CList,TTFlow,TopList),
/*      write("TTFlow=",TTFlow),nl,*/
        trace(off),
        calc_Yi_out(FCFlow,FCTFlow,TTFlow,Yi_out),
/*      write("Yi_out=",Yi_out),nl,*/
        x_in_solvent(FC,XFC_in),
        R=(Yi_in-Yi_out)/(Yi_in-FConst*XFC_in),
        N = log((R-A)/(R-1))/log(A) - 1.0,
/*      write("N=",N),nl,*/
        !,
        check_every(List,CList,L,RL,V,FConst,N,TopList,BotList,DBact,DBspec),
        trace(off),
        write("The number of theoretical plate is:"),
        writef("  %-2.1",N),nl,
        write("The absorption factor is:",A),nl,
        pause(Z),
        assertz(flow("solv",S,RL),materials),
        recovery([S],BotList,CList).

kremser(List,CList,[S],Last_L,V,FConst,TopList,BotList):-
        trace(off),
        Last_L/V/FConst >1.6,
        trace(off),
        set_L_over_KV_one_point_four(V,FConst,L),
/*      write("V=",V),nl,write("Last_L=",Last_L),nl,write("L=",L),nl,*/
        A = L/(FConst * V),
        write("A=",A),nl,
        trace(off),
        henry(FC,FConst),
        component_flow(List,FC,FCFlow),
/*      write("FCFlow=",FCFlow),nl,*/
        Yi_in = FCFlow / V,
/*      write("Yi_in=",Yi_in),nl,*/
        component_flow(TopList,FC,FCTFlow),
        sum_component_flow(CList,TTFlow,TopList),
        trace(off),
        Calc_Yi_out(FCFlow,FCTFlow,TTFlow,Yi_out),
        x_in_solvent(FC,XFC_in),
        R=(Yi_in-Yi_out)/(Yi_in-FConst*XFC_in),
        N = log((R-A)/(R-1))/log(A) - 1.0,
        check_N(N),
        check_one_point_four(L,V,FConst,NewL),
        calc_Component_d_over_b(List,Clist,L,V,N,TopList,BotList,DBact,DBspec),
        clearwindow,
        Q = Last_L / FConst/V, QR = L/FConst/V,
        write("The number of theoritical stage is:"),
        writef("  %-2.1",N),nl,
        write("The absorption factor is:",QR),nl,
        write("/*Do you accept the flowsheet with this conditions ?*/"),nl,
        menu(28,23,14,14,["Yes","No"],"Answer",2,Choice),
        !,
        /* verif backtracking if choice is no*/
        resolve_user_choice(Choice),
        trace(off),
        assertz(flow("solv",S,L),materials),
        recovery([S],BotList,BotCList).



calc_Yi_out(FCFlow,FCTFlow,TTFlow,Yi_out):-
        FCTFlow = 0,
        write("Yi_out being 0 implies N=infinite, we set 2% of FC in the top"),
        nl,
        Yi_out = 0.02 * FCFlow / TTFlow.

calc_Yi_out(FCFlow,FCTFlow,TTFlow,Yi_out):-
        Yi_out = FCTFlow / TTFlow.


check_every(List,CList,L,RL,V,FConst,N,TopList,BotList,DBact,DBspec):-
        check_N(N),
        check_one_point_four(L,V,FConst,NewL),
        !,
        check_in_spec(List,CList,L,NewL,RL,V,FConst,N,TopList,BotList,DBact,DBspec).

check_N(N):-
        N>5.

check_one_point_four(L,V,FConst,NewL):-
        L/V/FConst<=1.4,
        NewL=1.4*V*FConst.

check_one_point_four(L,V,FConst,NewL):-
        L/V/FConst>1.4,
        NewL=L.

check_in_spec(List,CList,L,NewL,RL,V,FConst,N,TopList,BotList,DBact,DBspec):-
        components_within_spec(List,CList,NewL,V,N,TopList,BotList,DBact,DBspec),
        RL=NewL.

check_in_spec(List,CList,L,NewL,RL,V,FConst,N,TopList,BotList,DBact,DBspec):-
        NewL/V/FConst = 1.4,
        components_within_spec(List,CList,L,V,N,TopList,BotList,DBact,DBspec),
        RL=L.

components_within_spec(_,[],_,_,_,_,_,[],[]).
components_within_spec(List,[HC|TC],L,V,N,TopList,BotList,[HDa|TDa],[HDs|TDs]):-
        trace(off),
        assert_comp_flow_in_solvent(["solv"],[HC],L),
        component_flow(TopList,HC,Dspec),
        component_flow(BotList,HC,Bspec),
        trace(off),
        d_over_b(Dspec,Bspec,DBspec),
        trace(off),
        component_flow(List,HC,FFlow),
        trace(off),
        Yi_in_actual = FFlow / V,
        henry(HC,FConst),
        A = L / (FConst * V),
        x_in_solvent(HC,Xi_in),
        Yi_out_actual = Yi_in_actual-(Yi_in_actual-FConst*Xi_in)*(exp((N+1)*ln(A))-A)/ (exp((N + 1)
* ln(A)) - 1),
        Dact = Yi_out_actual * V,
        Bact = FFlow - Dact,
        d_over_b(Dact,Bact,DBact),
        !,
        DBact < 1.10 * DBspec,
        DBact > 0.90 * DBspec,
        HDa = DBact,
        HDS = DBspec,
        trace(off),
        components_within_spec(List,TC,L,V,N,TopList,BotList,TDa,TDs).


assert_comp_flow_in_solvent(["solv"],[],_).
assert_comp_flow_in_solvent(["solv"],[HC|TC],L):-
        x_in_solvent(HC,X),
        LX=L*X,
        assertz(flow("solv",HC,LX),materials),
        assert_comp_flow_in_solvent(["solv"],TC,L).


calc_Component_d_over_b(_,[],_,_,_,_,_,[],[]).
calc_Component_d_over_b(List,[HC|TC],L,V,N,TopList,BotList,[HDa|TDa],[HDs|TDs]):-
        trace(off),
        assert_comp_flow_in_solvent(["solv"],[HC],L),
        component_flow(TopList,HC,Dspec),
        component_flow(BotList,HC,Bspec),
        trace(off),
        d_over_b(Dspec,Bspec,DBspec),
        trace(off),
        component_flow(List,HC,FFlow),
        trace(off),
        Yi_in_actual = FFlow / V,
        henry(HC,FConst),
        A = L / (FConst * V),
        x_in_solvent(HC,Xi_in),
        Yi_out_actual = Yi_in_actual-(Yi_in_actual-FConst*Xi_in)*(exp((N+1)*ln(A))-A)/ (exp((N + 1)
* ln(A)) - 1),
        Dact = Yi_out_actual * V,
        trace(off),
        Bact = FFlow - Dact,
        d_over_b(Dact,Bact,DBact),
        HDa = DBact,
        HDS = DBspec,
        calc_Component_d_over_b(List,TC,L,V,N,TopList,BotList,TDa,TDs).

d_over_b(0,_,0.02040816327).
d_over_b(_,0,49.0).
d_over_b(D,B,DB):-
        D/B>49.0,
        DB=49.0.
d_over_b(D,B,DB):-
        D/B<0.02040816327,
        DB=0.02040816327.
d_over_b(D,B,DB):-
        DB=D/B.

check_member(X,List):-member(X,List).
check_member(X,List):-write("/your chosen fixed component is not in the list/"),
                       fail.
check_FC_in_solvent(X):-x_in_solvent(X,0).

set_L_over_KV_one_point_four(V,FConst,L):-
        L = 1.4*V*FConst.

resolve_user_choice(1).
resolve_user_choice(2):-fail.

print_list([]).
print_list([Head|Tail]):-
        !,write(Head,' '),
        print_list(Tail).
print_vlist([]).
print_vlist([Head|Tail]):-
        !,write(Head,' '),
        print_vlist(Tail).

recovery([S],BotList,CList):-
        dseparator(Rum,_,_,_,_),
        RRum = Rum + 1,
        trace(off),
        delete_element("solv",BotList,Bot_no_S_List),
        print_list(Bot_no_S_list),
        append(CList,[S],CRList),
        trace(off),
        component_flow_set(Bot_no_S_List,CRList,TopFRList),
/*      print_Vlist(TopFRList),*/
        component_flow_set(["solv"],CRList,BotFRList),
        trace(off),
        asserta(dseparator(RRum,CRList,TopFRList,CRList,BotFRList),sequence),
        dseparator(X,Y,Z,T,U),
/*      write("Num before reco=",X),nl,
        write("CRList:"),print_List(Y),nl,
        write("TopFRList:"),print_vList(Z),nl,
        write("CRList:"),print_List(T),nl,
        write("BotFRList:"),print_vList(U),nl,*/
        shiftwindow(10),
        clearwindow,
        write("The results for the RECOVERY  COLUMN #",RRum," are now available"),
        pause(N),
        clearwindow,
        trace(on),
        print_cam_driver(BotList,CRList,_),
        pause(N).
```

REFERENCES

Douglas, J.M., *Conceptual Design of Chemical Processes*, McGraw-Hill, New York, NY (1988).

Barnicki, S.D., "Distill: A KEE-Based Distillation Column Sequencing System," *Separations Research Program*, **F-89-8**, University of Texas at Austin, Austin, TX, (1989).

Barnicki, S.D. and J.R. Fair, "Separation Synthesis: A Knowledge-Based Approach. 1. Liquid Mixture Separations," *Ind. Eng. Chem. Res.*, **29**, 421 (1990).

Diab, S,. and R.H. Maddox, "Absorption," *Chemical Engineering*, **89** (No. 26) 38, December (1982).

Keller, G.E., ***Adsorption, Gas Absorption, and Liquid-Liquid Extraction: Selecting a Process and Conserving Energy***, MIT Press, Cambridge, MA (1982).

Kohl, A.L., "Absorption and Stripping," in *Handbook of Separation Process Technology*, pp. 340-404, R.W. Rousseau, Editor, Wiley, New York, NY (1987).

Liu, Y. A., T. E. Quantrille, and S. H. Cheng, "Studies in Chemical Process Design and Synthesis. 9. A Unifying method for the Synthesis of Multicomponent Separation Sequences with Sloppy Product Streams," *Ind. Eng. Chem. Res.*, **29**, (1990).

Nelson, W.L., *Petroleum Refinery Engineering*, pp. 850-858, McGraw-Hill, New York, NY (1969).

Treybal, *Mass Transfer Operations*, Third Edition, pp.275-341, McGraw-Hill, New York, NY (1980)

Wankat, *Equilibrium-Staged Separation Operations*, Elsevier, New York, NY (1988).

VITA

The author was born in Bethesda, Maryland, on September 5, 1961. He is the son of Clinton A. and Janet P. Quantrille. The author is married to Sharon S. Quantrille. At the time of this writing, both the author and his wife are expecting their first child.

Following his high school education at Oakton High School in Vienna, Virginia, the author attended Virginia Polytechnic Institute and State University from 1979-1984 where he received his B.S. in Chemical Engineering.

After completion of his undergraduate studies, the author was employed industrially from 1984-1988 as a research and development engineer. The author worked for a major consumer-products company as a process development engineer, and was involved in polymer and material science in the packaging area. The author took another research position with a major pulp and paper company, and again worked in the polymer science area, but this time with emphasis on nonwoven fibers and fabrics.

After completing over four years in industrial research, the author returned full-time to Virginia Polytechnic Institute and State University in 1988 to pursue doctoral studies in chemical engineering. The author is a member of AIChE, Kappa Theta Epsilon, TAPPI, INDA, and AAAI.

Upon graduation, the author plans to work as a research scientist in polymer science and engineering, with emphasis on nonwoven fibers.