VPI Prolog Compiler
Project Report

by

John Deighan

Project Report submitted to the Faculty of

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

in

Computer Science

APPROVED:

John W. Roach

Dr. John Roach, Chairman

James D. Arthur

Dr. James D. Arthur

A. Bloss

Dr. Adrienne Bloss

November, 1991

Blacksburg, Virginia

Abstract

This project report documents a PROLOG compiler based on an abstract machine design published by David H. D. Warren. The implementation is actually a low-level intermediate language generator and interpreter. However, it is customary among PROLOG users to call this a compiler, since most PROLOG implementations are pure interpreters, i.e. they store PROLOG code as lists which mirror the source clauses. This compiler imcorporates many extensions to the abstract machine described by Dr. Warren.

Acknowledgements

The design of the VPI Prolog compiler was heavily influenced by the abstract machine design by David H. D. Warren, as set forth in the paper An Abstract Prolog Instruction Set (SRI International Technical Note 309, Oct. 1983). We have made many enhancements to the abstract instruction set given therein, but kept the basic core instructions.

In addition, the authors would like to acknowledge the help of a large number of M.S. students of the Department of Computer Science at Virginia Tech who contributed to the construction of a preliminary version of this compiler, including Chandan Chitale, Guru Comarapalyama, and Steve Haugh.

Conventions used in this report

The font called New York is used throughout this report for explanatory text (such as this). The font called Helvetica is used for text which can be entered at the VPI Prolog prompt (sometimes the prompt will also be shown, but usually only the text to be entered is shown). The Helvetica font can be easily recognized because it has no serifs. An example of the Helvetica font is:

This is the font called Helvetica

In presenting the syntax for a command, the following conventions will be used. An item in angle brackets such as <int> represents a class of possible values. In this case, <int> represents any integer. If an item is enclosed in tall square brackets, it is optional and may be omitted. For example, in the command

step [<int>]

the command step may be followed by an integer, or may appear by itself. If an item is enclosed in tall curly brackets, it may appear zero or more times. For example, in the description of the print predicate

((print { <term> }))

the word "print" may be followed by zero or more terms.

If one of several alternatives is allowed, they may be presented separated by a tall vertical bar, like this

<atom> | <number>

which means that either an atom or a number may appear. Often, the use of this tall vertical bar leads to ambiguity, which will then be resolved using tall parentheses for grouping. For example, in the description of the echo command

echo (<filename> | off)

the echo command may be followed by a filename or the word 'off'. Without the parentheses, it would not be clear whether the command name 'echo' was just part of the first optional part.

Table of Contents

Chapter 1

Introduction

This report documents a PROLOG compiler that has been developed at Virginia Tech. In this document, the word PROLOG (upper case) will be used to refer to the language itself, and our implementation will be referred to as VPI Prolog. The code is written in C and has proven to be highly portable. VPI Prolog runs on the following computer systems.

Computer	Operating System
Sun SPARC Station	UNIX
VAX	VMS
Apple Macintosh	Mac OS
Apple Macintosh	A/UX
IBM-PC	MS-DOS
Data General	AOS/VS
Commodore Amiga	UNIX

PROLOG is a member of a class of logic programming and rule-based languages that are being explored by artificial intelligence researchers as a tool for working with symbolic data. Experience has shown that there are a number of problems with these languages. Program execution is often slow, the user environment is primitive, input/output facilities are poor, and there is often no interface to other languages. Most of these problems are due to the relative youth and lack of

development of these languages. Please see Clocksin and Mellish [3] and Kowalski [8] for a full description of the PROLOG language.

A PROLOG program consists mainly of a set of rules. A rule (also called a clause) consists of a head and zero or more subgoals. Each head and subgoal consists of a predicate name and zero or more arguments, a structure which is called a relational expression.

Besides the logic interpretation of PROLOG, there is also a database interpretation of PROLOG in which each rule which contains no subgoals is viewed as a database record. Such a rule is called a fact. A collection of facts with the same predicate name then define a database in the relational sense. In this view, rules with subgoals define logical databases by giving rules for their construction, rather than listing their contents literally. Unfortunately, virtually all PROLOG implementations lack the compact storage and efficient access methods of database systems, although Warren's abstract machine did include some rudimentary indexing capability for rules.

1.1 Implementations of PROLOG

Most early implementations of PROLOG were interpreters. When the user entered a rule into the rule base, the rule was stored in a form very much like the original clause - usually a linked list of relational expressions. Execution begins when the user enters a goal, i.e. one or more relational expressions. The interpreter proceeds by attempting to

find a rule with a matching head. If found, the original relational expression is replaced by the subgoals in the rule.

This direct implementation was a natural one to use for experimenting with the PROLOG language, but they, like most interpreters, suffered from poor execution speed. In 1983, David H. D. Warren published a paper [16] describing an abstract machine (which he called the WAM) with low-level instructions, along with directions for translating PROLOG code into these instructions. This abstract machine incorporated unification and backtracking in an efficient manner. Prolog systems based on this abstract machine are generally called compilers, to distinguish them from the interpreters which preceded them.

At Virginia Tech, much of the AI research being conducted used a PROLOG interpreter written by a former graduate student, which was named HC. In order to obtain a state of the art PROLOG compiler, it was decided that a PROLOG compiler based on the WAM was needed. Warren's description of the WAM left many holes to be filled in, but the basics were there. Implementation of the following features was not described by Warren, and their design in the compiler described in this report was developed by myself:

In addition to implementing a WAM based PROLOG compiler, it was decided to incorporate some database technology, including a demand paging system using B+ trees for indexing of database records.

Although Dr. D. H. Warren described an abstract machine design suitable for use in constructing a PROLOG compiler, he did not include the implementation of many important PROLOG features, including the following. Their design in this compiler represents new work.

1. Cut
2. Builtin or
3. Builtin not
4. Assert
5. Retract
6. Variable predicates
7. Variable goals
8. Debugging facilities
9. Database facilities
10. Environment saving and restoring

In addition, some elements of Dr. Warren's design have been improved and/or extended, in particular, the indexing of normal (i.e. non-database) rules.

1.2 Contents of this Report

This report consists of five chapters, including this introduction. In the second chapter I will describe the abstract machine on which the VPI Prolog compiler is based. In the third chapter, I describe how the compiler can be executed, along with various aspects of the user interface.The fourth chapter describes in some detail the syntax which VPI Prolog accepts, along with some example programs, and how they

may be entered and executed. The fifth chapter describes how to access the database facilities of VPI Prolog.

Chapter 2

Design of the Abstract Prolog Machine (PLM)

2.1 Overview

In this document, we will trace the design of the abstract machine upon which the VPI Prolog compiler is based. This abstract machine is an extension of an abstract machine described by Warren (1983). This abstract machine can be thought of as if it were a real microprocessor with machine instructions, registers, and memory. This machine will be referred to as the PLM. We will start with a very simple machine description with few instructions, then add additional capabilities (along with the PLM instructions needed to implement them) one at a time, culminating in the complete PLM upon which our compiler is based.

2.2 Prolog Machine #1

PLM #1 syntax

The first version of the PLM presented will be called PLM #1, and will allow only a very restricted syntax. The Prolog syntax which the compiler for PLM #1 handles is very simple:

```
<term> ::= <atom> | <variable>
<head> ::= '(' <atom> { <term> } ')'
<goal> ::= '(' <atom> { <term> } ')'
<rule> ::= '(' <head> [ if <goal> { <goal> } ] ')'
<assertion> ::= '(' assert { <rule> } ')'
<query> ::= '(' { <goal> } ')'
<program> ::= <assertion> <query>
```

where non-terminals are enclosed in angle brackets, quoted symbols are terminals, taken literally, | means either/or, [and] surround optional constructs, and { and } means zero or more of the enclosed construct.

Let an identifier be any string of characters not containing a blank, parenthesis, newline, or other control character. A <variable>, then, is an identifier whose first character is a question mark. An <atom> is any identifier that does not start with a question mark.

There is one more restriction in PLM #1: at most one rule can be asserted for each predicate. Therefore, upon failure, the computation stops - no backtracking ever occurs.

Consider the following program for this PLM:

```
(assert
    ((lucky ?x) if (loves Helen ?x))                    ; (1)
    ((unlucky ?x) if (bully ?y)(hates ?y ?x))           ; (2)
    ((loves Helen ?x))                                  ; (3)
    ((bully  Buster))                                   ; (4)
    ((hates Buster Bob))                                ; (5)
    )                                                   ; (6)
((unlucky ?x)(lucky ?x))                                ; (7)
```

The program has the following logical meaning. Line 1 states that a person (?x) is lucky if Helen loves that person. Line 2 states that a person (?x) is unlucky if some other person (?y) is a bully, and that bully hates the original person (?x). Line 3 states that Helen loves everyone. Line 4 states that Buster is a bully, and line 5 states that Buster hates Bob. The query (line 7) attempts to discover a person (returned in ?x) for which both (unlucky ?x) and (lucky ?x) are true.

The following Prolog constructs are treated much like the procedure-oriented constructs in the following table:

Prolog construct	Procedure-oriented construct
query	main program block
rule	procedure definition
goal	procedure call
goal arguments	actual parameters
variable	variable local to a procedure

Thus, the above query can be viewed as a main program with 2 procedure calls. The arguments in a goal are treated like actual parameters, i.e. they are "passed" to the procedure being called. Each rule (as well as the query itself) has its own set of variables, which initially hold no value (i.e., are unbound). Each time a rule is "called", a new set of empty variables for the called rule is allocated on the stack. During execution, these variables may become bound, i.e. take on values.

Registers and memory areas

PLM #1 will contain 2 areas of memory, the code area and the stack, along with the following registers: 1) the PC (or program counter) contains the location in the code area of the next instruction to be executed, 2) the RR (or return register) contains the location in the code area of the next instruction to be executed after the next return instruction is executed, 3) the E register (or environment pointer) points to the current environment frame on the stack, and 4) the TOS (or top of stack) points to the first unused location on the stack. Last of all, the array of argument registers A0, A1, etc. contain terms representing arguments passed to called Prolog rules.

The stack

The stack holds environment frames. Each rule currently being executed has a corresponding environment frame on the stack. An environment frame contains a reference to the previous environment

frame, a copy of the RR register when the frame was created, and the variables for the corresponding rule. For example, the rule for **unlucky** contains 2 distinct variables, ?x and ?y, and therefore its environment frame contains 2 variables. Note that while a procedure (predicate) is being executed, only that procedure's local variables are accessible, i.e., there is no concept corresponding to the 'global variables' available in other languages. Variables within a rule are numbered, starting from 0; this number can be used to access a variable as an offset from where the E register points. The E register initially points to the bottom of the stack, and variables in the query are referenced via offsets from this location. In other words, corresponding to the query is an initial environment frame which consists only of variables, without a reference to a previous environment frame or copy of the RR register.

Representing terms in the PLM

Terms in the PLM are represented by an ordered pair, a tag indicating the kind of term we are dealing with, and some data which varies depending on the tag. PLM #1 has only 2 kinds of terms, the variable and the atom. A variable will contain the tag "undef", with no other data, and the atom will contain the tag "atom" followed by the atom's name. For example, the atom "a" and a variable are represented as:

<atom, a>
<undef>

Actually, the atom term's data is an atom number, where each unique atom seen by Prolog is assigned a number. This allows an atom term to be of a fixed size.

Internally, PLM #1 has one other kind of term, called a reference term. A reference term consists of the tag "ref" followed by a reference to another term on the stack. Logically, therefore, the reference term represents the term it refers to and does not represent a new kind of Prolog term. Reference terms are needed to implement variable-variable binding.

Different atoms are distinguished by having different data components, i.e. if the term <atom, a> exists in two different memory locations, they still represent the same atom. Variables, on the other hand, are distinguished by their location, i.e. two <undef> terms in separate locations represent different variables.

Variable binding

A variable in Prolog represents an undetermined (or unbound) value. During execution, a variable may be assigned (bound to) a value. This occurs, for example, if a variable is unified with an atom (unification is discussed below). A variable may be bound either of two ways. The new value may directly overwrite the <undef> term, or the <undef> term may be replaced by a reference term, containing a reference to the other term. If both terms are variables, then, since variables are distinguished

by location, we must bind the terms by replacing one <undef> term with a reference term containing a reference to the other variable.

In our implementation, references are implemented using a memory address, but other ways to refer to terms are possible. Since in this PLM all terms exist on the stack, in our discussions we will use S5, for example, to refer to element #5 of the stack.

Unification and failure

When a rule is "called", the arguments of the calling goal are matched (unified) with the arguments in the head of the rule being called. This matching is somewhat analogous to the matching between the actual and formal parameters which occurs in a procedure-oriented language. In PLM #1, unification of two terms is simple: 1) if both terms are atoms, then unification succeeds if and only if both atoms are identical, 2) if one term is an atom, and the second is a variable, the variable is bound to the atom, and 3) if both terms are variables, the variable appearing higher in the stack is bound to (changed to a reference term referencing) the other variable. If either of the terms is a reference term, the term it refers to is used. This process of following a chain of reference terms to find a non-reference term to use is called dereferencing the term.

Note that, in PLM #1, all <undef> (i.e. variable) terms exist on the stack. Also, note that since each clause has its own set of variables,

variables appearing in different clauses are different variables even though they may have the same name in the source text.

Consider the following rule:

```
((P a ?x) if (Q ?x))
```

The head of this rule, (P a ?x), has 2 arguments, the atom **a** and the variable ?x. If the first argument in the calling goal is the atom **a**, then matching succeeds, and Prolog moves on to the second argument; if the first argument in the calling goal is a variable, that variable is bound to the atom **a** and matching also succeeds; otherwise, the head matching, and therefore the call to the rule, fails. In PLM #1, failure of a single unification causes failure of the entire computation. (In Prolog, however, failure does not mean an error occurred, it is simply one possible outcome of a computation). The second argument to a call to **P**, however, can be anything. If an atom, then the variable ?x is bound to that atom (which is later passed to the **Q** rule). If a variable, then the variable ?x is bound to the incoming variable, (which is later passed to the **Q** rule).

If the goal (P a b) is being executed, then head matching in the rule above would succeed, binding the variable ?x to the atom **b**. After head matching succeeds, a call is made to **Q**, passing the atom **b** as **Q**'s only argument.

PLM Instructions

The following PLM instructions are needed to implement PLM #1 (instructions consist of a mnemonic followed by some arguments):

instruction	effect
allocEnv <n>	Allocate a new environment frame containing <n> variables.
getVar V<n> from A<m>	Unify V<n> (the term at E + <n>) with the contents of argument register A<m>. Note that A<m> is either an atom term, or a reference term containing a reference to an <undef> term on the stack.
getConst <trm> from A<m>	Dereference A<m>. If the result is an <undef> term, replace it with the term <trm>; else if the result isn't the term <trm>, fail.
putVar V<n> into A<m>	Put the term <ref, E + <n>> into A<m>.
putConst <trm> into A<m>	put the term <trm> into A<m>
deallocEnv	deallocate the current environment frame, restoring the previous RR and E registers
call <pred>	call the code for the predicate <pred> by setting RR to the instruction following the call, and setting PC to the start of the code for <pred>
return	return from the current call, i.e. set PC to the current RR
halt	halt execution

The V<n> represent variables, which are numbered starting from 0, and A<m> represent argument registers, which are also numbered from 0, i.e. the first argument is stored in A0, the second in A1, etc. Note that the registers never contain the term <undef>. Since variables are

distinguished by location, when we wish to pass a variable to another procedure, we cannot copy the variable into the argument register, but must put a reference to the variable into the argument register. In effect, variables are passed "by reference".

A Prolog rule generates a block of code which conforms to the following pattern:

```
allocEnv <n>
<get instructions>
-----------------
<put instructions>
call <pred>
-----------------
deallocEnv
return
```

where the section between the dashed lines is repeated once for each goal in the rule, and <n> is the number of variables in the rule being translated. If there are no variables in the rule, the allocEnv and deallocEnv instructions are omitted.

A Prolog query generates a block of code which conforms to the following pattern:

```
-----------------
<put instructions>
call <pred>
-----------------
halt
```

where the section between dotted lines is repeated once for each goal in the query.

The PLM #1 code generated by the example Prolog program given earlier is (with some labels added at the left for reference) :

```
lucky:          allocEnv 1
                getVar V0 from A0
                putConst <atom, Helen> into A0
                putVar V0 into A1
                call loves
                deallocEnv
                return

unlucky:        allocEnv 2
                getVar V0 from A0
                putVar V1 into A0
                call bully
                putVar V1 into A0
                putVar V0 into A1
                call hates
                deallocEnv
                return

loves:          allocEnv 1
                getConst <atom, Helen> from A0
                getVar V0 from A1
                deallocEnv
                return

bully:          getConst <atom, Buster> from A0
                return

hates:          getConst <atom, Buster> from A0
                getConst <atom, Bob> from A1
                return

query: (PC)->   putVar V0 into A0
                call unlucky
                putVar V0 into A0
                call lucky
                halt
```

Handling of variables

In PLM #1, variables are handled as follows:

1. When an "allocEnv n" instruction is encountered, after pushing E and RR onto the stack, n <undef> terms are pushed onto the stack, i.e. the variables are initialized to <undef>.

2. When a getVar Vn from Am instruction is encountered, we must check the value of variable Vn since it may no longer be set to <undef>. For example, consider the rule:

```
((P ?x ?x) if (Q ?x))
```

which generates the code:

```
P:  allocEnv 1
    getVar V0 from R0
    getVar V0 from R1
    putVar V0 into R0
    call Q
    deallocEnv
    return
```

and the goal

```
(P a b)
```

which generates the following code:

```
    putConst <atom, a> into R0
    putConst <atom, b> into R1
    call P
```

The variable ?x will correspond to variable V0 in the clause. V0 will initially be <undef>; then the first getVar instruction will set V0 to <atom, a>. When the second getVar instruction is executed, V0 is no longer <undef>. Execution of the getVar instruction, therefore, requires a full, general unification.

3. When a putVar instruction is encountered, <ref, E+<n>> is placed in Am. Once again, the variable Vn may be bound or unbound.

Consider the first occurrence of the variable ?x in 2. above. As noted, execution of a getVar instruction in general requires a check to see if the corresponding stack variable is still <undef>. However, since this instruction corresponds to the first occurrence of this variable in the clause, the corresponding variable must be <undef>. We therefore introduce the getNewVar instruction which can be used in place of getVar in this situation. If the variable ?x occurs first in a goal, there is a corresponding putVar instruction. Once again, execution of the putVar instruction cannot, in general, assume that the variable is <undef>. In this instance, however, it must be, so we introduce the putNewVar instruction, which simply places in the argument register a reference to the variable.

Example execution

Execution of the above program on PLM #1 will now be illustrated using Figure 1. In Figure 1, the stack is represented as a sequence of

cells labeled S0, S1, etc. Each cell holds a term in the <tag, data> form explained above. The argument registers are represented as a sequence of cells labeled A0, A1, etc. The code area is represented as the sequence of PLM instructions above. The registers PC and RR are represented by the words "PC" and "RR" placed near the PLM instruction they point to along with an arrow pointing to the instruction. The E register is represented by "E" along with an arrow pointing to the stack cell it points to. TOS implicitly points to the first unoccupied cell on the stack. When execution begins:

1. E points to S0, the first cell on the stack
2. Let n be the number of variables in the query. Each of S0 through S(n-1) are set to <undef>, and TOS, therefore, implicitly points to Sn.
3. PC points to the first instruction in the query.
4. RR contains no value, and therefore does not appear in the code above.

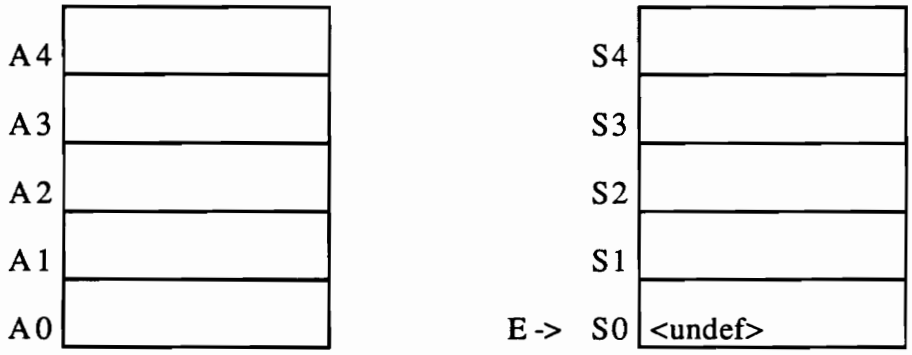
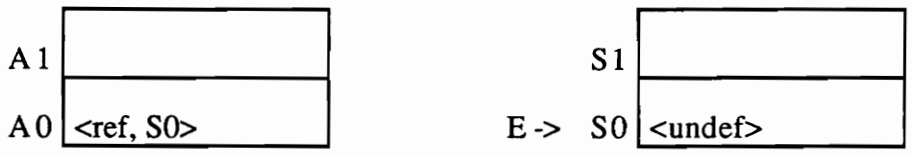


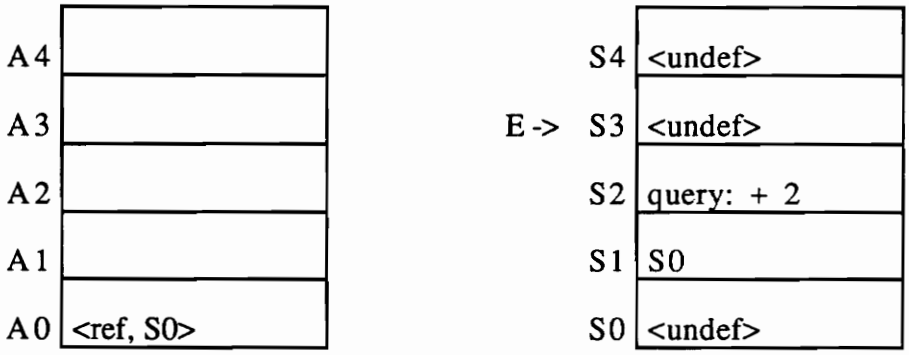
Figure 1

There are actually many more argument registers and stack cells than are depicted here. In future diagrams, simply assume that any undisplayed argument registers and stack cells are empty. After execution of the first instruction (putVar V0 into R0), the PLM looks like this:

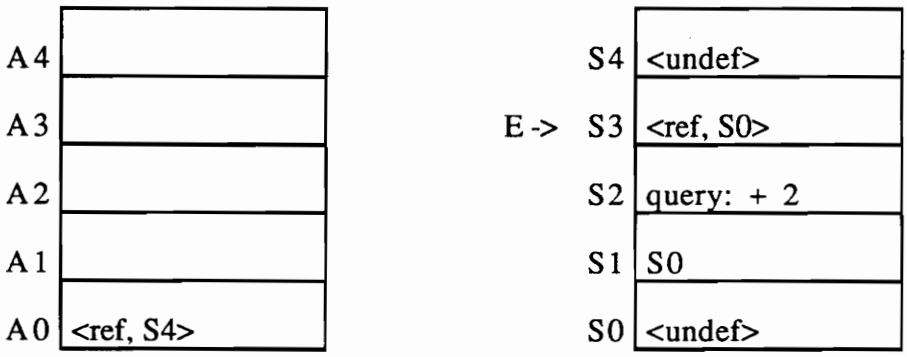


Next, the call instruction is executed. The argument registers and stack do not change; however, RR is set to point to the instruction following the call, and the PC is set to point to the first instruction in the code for the 'unlucky' predicate.

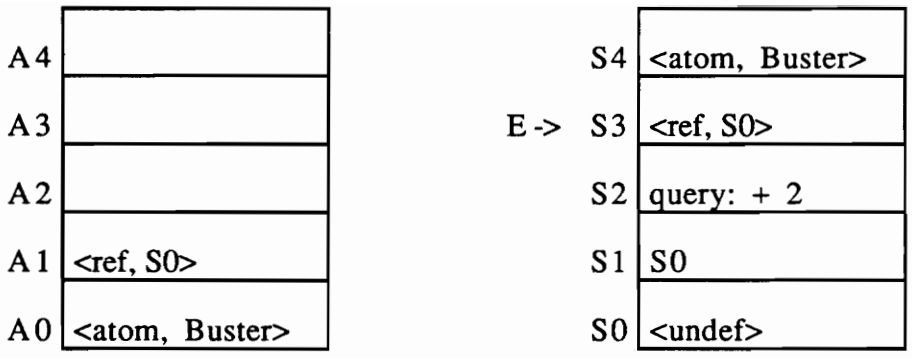
Next, the 'allocEnv 2' instruction is executed. This results in E and RR being pushed onto the stack, along with 2 new variables. E is then set to point to the first of these variables. The PLM now looks like this:



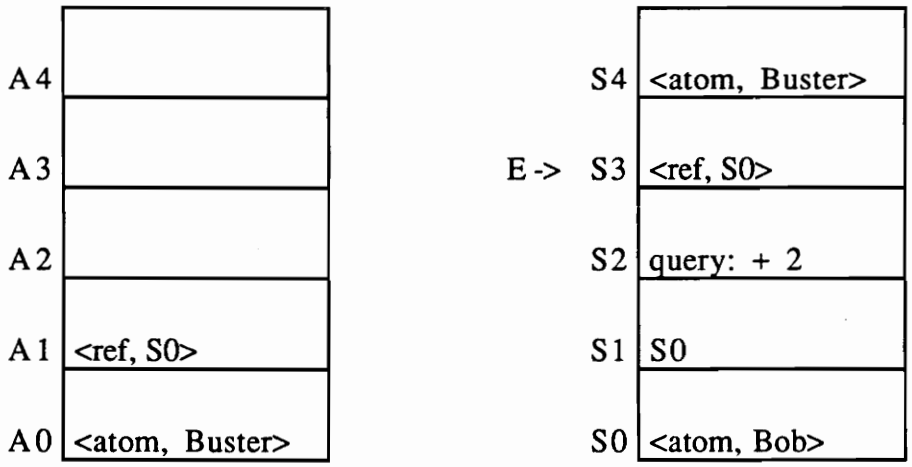
We will use the labels in the PLM code to denote addresses within the code. For example, the term 'query: + 2' above refers to the third instruction in the query. After executing the getVar, putVar, and call instructions at the beginning of the 'unlucky:' code, the PLM looks like this (with RR pointing to unlucky: + 4, and PC pointing to bully:):



The code for 'bully' now binds the variable at S4 with the atom Buster, and the return instruction sets PC to RR, thus PC is back at unlucky: + 4. By the time execution gets to the 'hates' predicate, the PLM looks like this:

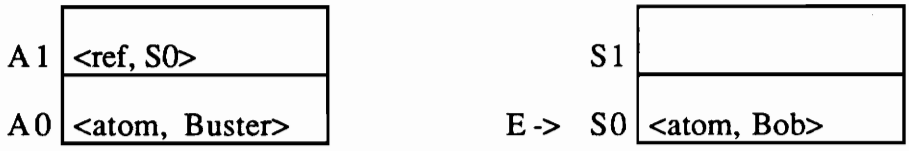


The code for 'hates' then checks to see that the atom Buster is the first argument, and sets the second argument (i.e., the variable it refers to) to Bob, then returns. The PC now points to the deallocEnv instruction in the code for 'unlucky', and the PLM looks like this:

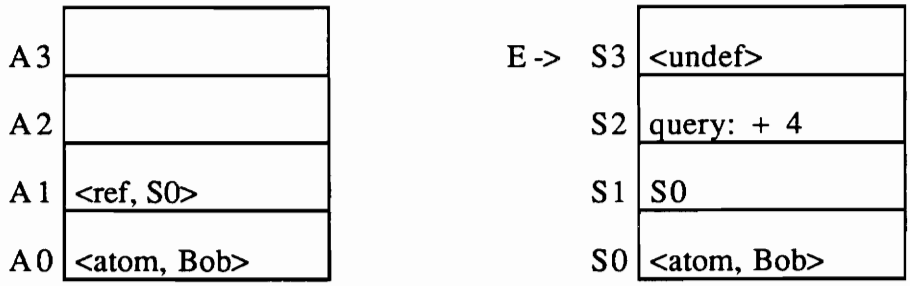


The deallocEnv instruction is now executed, which removes the variables in the current environment from the stack, and restores E and RR from the current environment frame. The return instruction then

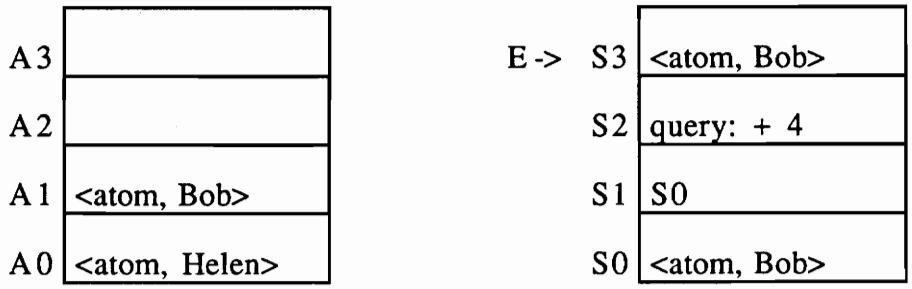
restores the PC from RR, which has just been set to query: + 2. The PLM now looks like this:



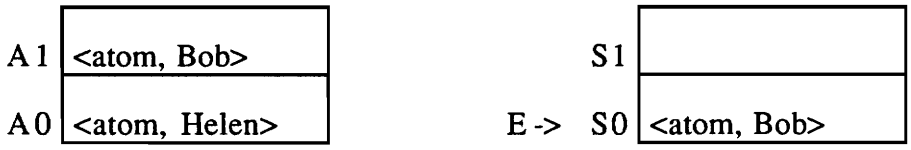
The putVar instruction in the query following the 'call unlucky' instruction is now executed, putting <atom, Bob> into A0, then the code for 'lucky' is executed. After execution of the initial allocEnv instruction, the PLM looks like this:



By the time the 'loves' predicate is called, the PLM looks like this:



The code for the 'loves' predicate simply checks that the first argument it is passed is the atom Helen, which it is, then it returns. Back in the code for 'lucky', we simply deallocate the current environment frame, and return to the query code, executing the 'halt' instruction. The query succeeded, binding the variable to the atom Bob. The final state of the PLM looks like this:



The original query was:

((unlucky ?x)(lucky ?x))

but when execution terminates, since ?x was bound to the atom Bob (as determined by looking at the stack), Prolog provides the answer:

((unlucky Bob)(lucky Bob))

2.3 Prolog Machine #2 - Multiple Rules

PLM #2 will build on PLM #1 by allowing multiple rules to be defined for a predicate. The execution of a Prolog program is now considerably more involved. When a call to a predicate with more than one rule is encountered, Prolog must now choose one (Prolog always chooses the first one), but still remember the other possible choices in case the first fails. What is more, if the first fails, the next possible

choice must be tried in the same state (i.e. with the same arguments) as the original call. In other words, Prolog must remember the original arguments in the call, and must undo any variable bindings created in attempting to satisfy the call via the first clause for that predicate.

Two new data structures must be introduced. First, whenever a call may result in either of several clauses being activated, a choice point must be created. The current argument registers and the current value of the RR and E registers need to be stored in the choice point so that the state currently in effect can later be restored. Also, since several choice points may exist at the same time, each choice point contains a reference to the previous choice point. A new register, called the B register, points to the current (or last created) choice point. Choice points are placed on the stack along with environment frames. Second, a new memory area called the trail stores the addresses of variables which are bound during execution of the first clause, and will need to be unbound before attempting an alternative clause. A new register, called TR, points to the next unused entry on the trail. The trail is managed like a stack, with new entries going on the top, and entries being popped off one at a time as variables need to be unbound. A choice point, in addition to the entries listed above, contains the value of the TR register in effect when it was created. Lastly, the address of the next clause to be tried in the event of failure of the current clause needs to be stored in the choice point.

Four new opcodes handle the creation and destruction of choice points, in addition to the restoration of previous states using information stored in a choice point. These opcodes and their effect are:

instruction	effect
onlyClause <n>	Does nothing, used as a place holder.
firstClause <address>	Create a new choice point and put <address> into FR.
interClause <address>	Restore a previous state from the choice point pointed to by B, and put <address> into FR.
lastClause	Restore a previous state from the choice point pointed to by B and remove the choice point from the stack (restoring B from the choice point).

Every clause begins with one of these opcodes, and these opcodes never occur anywhere but as the first opcode for a clause. Note that if a predicate has only one rule defining it, the first opcode will be 'onlyClause'. If the predicate has more than one defining rule, the first clause will begin with the opcode 'firstClause', intermediate clauses will start with the opcode 'interClause', and the last will start with the opcode 'lastClause'. As rules are asserted and/or retracted for a given predicate, these opcodes are modified (or 'patched') to maintain this relationship. This is why the opcode 'onlyClause' is needed as a placeholder.

2.4 Prolog Machine #3 - Structured Terms

Atoms do not provide a very rich domain in which to compute. Therefore, Prolog provides the functor term, a record-like structure. The syntax for a functor term is:

<term> ::= '[' <atom> { <term> } ']'

The term is delimited by square brackets, starts with an atom, called the principle functor of the term, and can have zero or more arguments. Functor terms are never evaluated, i.e. they do not represent functions. Each functor, like a predicate, has a fixed arity, or number of arguments. Prolog deduces the arity by the first instance it sees, and enforces it thereafter.

Two functor terms unify if and only if their principle functors are the same, and all of their terms unify. Prolog represents a functor term by the pair <functor, address>, where 'functor' is a new tag, and address points to a memory location containing a functor number, followed by n terms, where n is the arity of the functor. Prolog maintains a functor table, and as each new functor is found in the source code, Prolog creates an entry for it, storing the functor number and arity.

A new Prolog data structure, called the heap, is used to store the functor number and argument terms. Space could be allocated on the stack but, as we shall see, when a Prolog clause is finished executing, functor terms which were created by that clause often continue to exist.

Therefore, using stack space would prevent the reuse of space used for variables, which can be deallocated when a Prolog clause finishes executing. In fact, one of the most important distinctions between the Prolog stack and heap is that stack space can be recovered when the last clause of a procedure (i.e. set of clauses with the same head predicate) is finished executing, whereas heap space is only recovered when a clause fails.

Another common Prolog data structure is the list. In its simplest form, a list is something like

(grape apple)

i.e., a series of atoms delimited by parentheses. We also wish to allow lists to contain other kinds of terms (including other lists), so a list is really any number of terms surrounded by parentheses. Note that 'any number' includes zero, so we also allow the empty list ( ), also called nil. Formally, this kind of list is defined as

<term> ::= '(' { <term> } ')'

One more list form is common, although its importance may not be immediately apparent

(grape apple . ?rest)

This represents a list whose first element is 'grape', second element is 'apple', and ?rest represents further elements in the list. This represents a list which is not complete, but if it were completed, it would contain at

least two, and possibly many more, elements. Formally, this form is defined as

<term> ::= '(' <term> { <term> } '.' <term> ')'

Internally, lists are really built up of cons cells. The phrase cons cell, and the following description of the representation of lists, is derived from the programming language Lisp. Think of a cons cell as a pair of terms, represented as:

(<term> . <term>)

i.e. as a parenthesized pair of terms, separated by a period. Then the list above is really an abbreviated form for the fully-consed term:

(grape.(apple.nil))

The following is also a perfectly reasonable list:

(beef (grape apple) (beans peas))

Its fully-consed form is:

(beef.((grape.(apple.nil)). (beans.(peas.nil))))

where the list of fruits is seen to be embedded in the list of foods.

The fact that a cons cell is of fixed size implies the possibility of representing such lists using a functor called cons, and representing a cons cell as:

[cons <term> <term>]

In fact, however, lists are so common in Prolog that, rather than storing the functor number and two terms, we create a new term tag, called 'list', and represent a list term using the pair <list, address> where address points to a pair of terms.

Two cons cells unify if and only if their first terms unify, and their second terms unify. Unification of lists which are not in their fully-consed form is then accomplished by converting both lists to their fully-consed form, then applying this definition. Converting to fully-consed form is defined as follows:

1. The empty list, (), is converted to 'nil' everywhere.

2. If a list is of the form (<term> .. <term> . <term>)
   convert to (<term> .. (<term> . <term>))

2. If a list is of the form (<term> .. <term>)
   convert to (<term> .. (<term> . nil))

Repeated applications of the above rules will result in a fully-consed form from any valid list.

To support functor term, list creation and unification, a number of new data structures and opcodes must be introduced. There is an array of registers, SReg[0], SReg[1], etc., that is used to build lists and functor terms which are parts of other lists and functor terms. There is also an S register that points to a term on the heap that is being instantiated.

First let's look at the new 'put' instructions, which build up lists and functor terms to be passed as arguments:

instruction	effect
putNil into A<n>	Put the constant nil into register A<n>. Nil is a special constant which represents an empty list.
putList into A<n>	Put a new list term into A<n>, containing a reference to a new cons cell, allocated on the heap. Make S register point to base of new cons cell.
putSubList into S<n>	Put a new list term into SReg, containing a reference to a new cons cell, allocated on the heap. Make S register point to base of new cons cell.
putFunc into A<n>	Put a new functor term into A<n>, containing a reference to a new array of terms, allocated on the heap. Make S register point to the first of these terms.
putSubFunc into S<n>	Put a new functor term into SReg, containing a reference to a new array of terms, allocated on the heap. Make S register point to the first of these terms.

Next, let's look at the new 'get' instructions, which unify arguments passed in the argument registers with the terms in the head of clauses:

instruction	effect
getNil from A<n>	If A<n> contains a reference term, dereference it and replace the <undef> term with the term <nil>; else if A<n> doesn't contain the term <nil>, backtrack.
getList from A<n>	If A<n> contains a reference term, dereference it and replace the <undef> term with the term <list, <ref>> where <ref> is a reference to a new cons cell on the heap; set S to point to the first term in the cons cell; else if A<n> contains a list term, set S to point to the first term in the cons cell; otherwise, backtrack.
getSubList from S<n>	If SReg<n> contains a reference term, dereference it and replace the <undef> term with the term <list, <ref>> where <ref> is a reference to a new cons cell on the heap, and set S to point to the first term in the cons cell; otherwise, if SReg contains a list term, set S to point to the first term in the cons cell; otherwise, backtrack.
getFunc from A<n>	If A<n> contains a reference term, dereference it and replace the <undef> term with a functor term containing a reference to a new array of terms allocated on the heap, and set S to point to the first of these terms; otherwise, if A<n> contains a functor term, set S to point to the first of these terms; otherwise, backtrack.
getSubFunc from S<n>	If SReg<n> contains a reference term, dereference it and replace the <undef> term with a functor term containing a reference to a new array of terms allocated on the heap, and set S to point to the first of these terms; otherwise, if SReg contains a functor term, set S to point to the first of these terms; otherwise, backtrack.

Next, let's look at the new 'unify' instructions, which unify arguments in the SReg registers with the terms in the head of clauses:

instruction	effect
uniConst <trm>	if S points to an <undef> term, replace it with <trm>; else if the term S points to is <trm>, succeed; else fail.
uniVar V<n>	unify the term S points to with V<n>
uniStruct from S<n>	Unify the contents of SReg<n> with the term pointed to by S.

2.5 Cut

The effect of a cut in a clause is to delete any choice points created during the running of the current clause, in addition to any choice point referencing alternatives to the current clause. In order to accomplish the cut, the current choice point in effect when the clause was invoked must be stored somewhere. This "somewhere" should have the property that its space is deallocated when the current clause succeeds or fails, unless its running creates choice points. This suggests storing a reference to the last choice point created at the time the clause is invoked in the current environment. Our PLM inplementation stores this choice point reference (which we call a 'cut pointer') on the stack, just before the RR and E fields. As we will see later, space for more than one cut pointer will be needed in clauses containing a 'not' or 'or' goal. For implementing the standard Prolog cut operator, space is allocated at E-3 (i.e. 3 positions before where the environment pointer points) on the stack.

In order to implement cut correctly, we need to store the value of register B (the most recent choice point) that is in effect before an onlyClause, firstClause, interClause, or lastClause instruction is encountered. At this time, however, the allocEnv instruction has not yet been encountered; that is, an environment has not yet been created. Therefore, the S register (which is otherwise used only while building up or breaking down lists) is used to store the cut pointer temporarily until the environment is created. The following series of events is used to implement cut:

1. When an onlyClause or firstClause instruction is encountered, the value of B is copied to the S register. When an interClause instruction is encountered, the B field of the current choice point (i.e., the choice point which was topmost when the clause was invoked) is copied to the S register. When a lastClause instruction is encountered, the topmost choice point is removed, after which B is copied to S.

2. When an allocEnv instruction is encountered, an environment is allocated on the stack. The allocEnv instruction has two arguments specifying the number of variables, and the number of cut pointers required. In any clause which contains a cut, at least one cut pointer will be allocated. The value of the S register is then copied to E-3.

3. When a cut instruction is encountered, B is set to E-3, effectively removing any choice points created since the

current predicate was invoked. If the resulting B is non-NULL, the trail is cleaned up by setting any address on the trail which is now above B to NULL. Also, the HB, TR, and TOS registers are set to the values they had when the current choice point was created.

2.6 Prolog Machine #4 - Builtin 'or' and 'not'

The Or predicate

The syntax for an 'or' goal is as follows:

<goal> ::= '(' or { <goal> } ')'

The prolog code generated for an "or" goal is as follows. If there are no subgoals, a fail instruction is generated (the goal will always fail). If there is only one subgoal, code is generated for the goal as if it weren't imbedded in an 'or' goal. Otherwise, the generated code looks like this:

```
            orFirst, else goal2:
            <code for subgoal 1>
            jump  end:
goal2:      orLater, else goaln:
            <code for subgoal 2>
            jump end:
            orLater, else  0
            <code for subgoal n>
end:        <code following 'or'
goal>
```

where the code between the dotted lines is repeated for each subgoal after the first, and before the last. Generating code for the 'or' predicate requires the following new opcodes:

instruction	effect
orFirst, else <offset>	Create a choice point and set FR to PC+<offset>.
orLater, else <offset>	Restore registers from a previously created choice point. If <offset> is non-zero, set FR to PC+<offset>; otherwise, restore previous FR from the choicepoint and remove the choice point.
jump <offset>	Set PC to PC+<offset>.

The Not Predicate

The syntax for a 'not' goal is as follows:

<goal> ::= '(' not <goal> ')'

The prolog code generated for a 'not' goal looks like this:

```
        notFirst, else end:
        <code for subgoal>
        cut
        fail
        notLast
end:    <code following 'not' goal>
```

Generating code for the 'not' predicate requires the following new opcodes:

instruction	effect
notFirst, else <offset>	Create a choice point, setting the FR field to PC+<offset>.
notLast	Restore previous FR from the choicepoint and remove the choice point.
fail	Backtrack, i.e. fail the current goal.

2.7 Prolog Numbers

Prolog numbers are all in floating-point format. Since Prolog terms require room for a term tag, and Prolog terms must all fit into a 32-bit word, Prolog number terms consist of a tag, and a pointer which points to the actual floating-point format number. The number itself is stored on the heap.The following opcodes are used to handle Prolog numbers:

instruction	effect
putNum <number>, R<n>	Copy the number to the top of the heap, and place the term <number, H-1> in register n.
getNum <number>, R<n>	If register n does not contain the term <number, addr>, or if the number at address addr is not <number>, then fail.

2.8 Built-in Predicates

Prolog's built-in predicates are implemented as direct calls to C routines. A special opcode, 'ccall', is used. Its single argument is the address of the C routine that implements the predicate.

instruction	effect
ccall <addr>	Call the C routine at address <addr>. If the return value is false, the fail.

2.9 Last Call Optimization

Last call optimization improves performance in two ways. First, a 'return' instruction, rather than returning to the code which called the current clause, returns to the first clause in the calling chain that still has a goal to solve. This is sometimes called 'fast succeed.' More importantly, there is a space savings because an environment is deallocated before the last goal is called.

To understand last call optimization, consider the Prolog clause:

```
((unlucky ?x) if (bully ?y)(hates ?y ?x))
```

which generates the PLM code:

```
unlucky:  allocEnv 2
          getVar V0 from R0
          putVar V1 into R0
          call bully
          putVar V1 into R0
          putVar V0 into R1
          call hates
          deallocEnv
          return
```

When the PLM instruction "call hates" is being executed, only two instructions remain to be executed in the current clause, "deallocEnv" and "return." However, since the argument registers have already been filled in preparation for the call, there is no need at this point for the current environment to exist. In other words, the deallocEnv instruction could be placed before the call instruction. Since deallocating the current environment involves setting the return register RR to the value RR had when the current predicate was first called, we could simply transfer execution directly to the "hates" predicate - when it returns, the return register, RR, will be set to the place we would have returned to anyway. We therefore introduce the following new opcode, used in this scheme:

instruction	effect
goto <pred>	Set the PC to the start of the code for predicate <pred>.

Now, rather than ending the code for the last call in a clause with:

```
call <predicate>
deallocEnv
return
```

we end with:

```
deallocEnv
goto <predicate>
```

There is one problem with this scheme, however. Even though the argument registers are filled in preparation for the last call, it is possible that one or more of these registers contains a reference to a variable in the current environment. Before we deallocate the current environment, we must transfer any variable in the current environment which may be referenced by a term in an argument register to the heap. This is done during loading of the argument registers in preparation for the last call.

Variables in a clause may be classified as safe or unsafe. Safe variables are any variables that meet one or more of the following criteria:

1. The variable first occurs in the head. In this case, we know that head matching will bind the variable to a term (possibly another variable) which is in one of the argument registers - which cannot contain a reference to the current environment since the current environment didn't exist when the argument registers were filled. When the argument registers are later filled, no direct reference to

this variable can be created since the variable is no longer unbound (i.e. if involved in unification and it became a reference to a variable, it will be dereferenced, otherwise it will be copied).

2. The variable's first occurrence is in a list or functor term in a goal. In this case, a new unbound variable will be constructed on the heap as part of a cons cell (if a list) or a set of functor arguments (if a functor term), and the variable in the environment will be set to a reference to that new variable. As above, no direct reference to this variable can subsequently arise.

3. The variable only occurs in a single goal.

If a variable meets none of these criteria, it is considered unsafe. The putVar opcode normally generated for the last occurrence of one of these variables is replaced by a putUVar ("put unsafe variable") opcode.

instruction	effect
putUVar V<n>, R<n>	Dereference variable V<n>. If the result is a variable in the current environment, place a new variable on the heap, put a reference to that variable in register R<n>, and put the term <ref, H-1> into V<n>.

2.10 Environment Trimming

Environment trimming is not implemented in the VPI Prolog compiler. However, understanding it will clarify the next topic. Environment trimming allows the reduction in size of an environment under certain conditions during the execution of a clause as long as no choice points are created during the execution. Consider the following clause:

```
((P  ?x  ?y) if
    (R  ?x)
    (S ?y ?z)
    (T  ?z))
```

The variable ?x is used to pass a value to the first goal, but is not required thereafter. If the variables were ordered in the environment with ?z first, ?y second, then ?x last, then when the call to R completed, we could act as though the environment contained only 2 variables thereafter. In other words, when the next environment or choice point was allocated on the stack, the space used for variable ?x could be reused. Actually, as we will see, assuming that ?x is a safe variable (it is here), its space can be reused after the argument registers are loaded for the call to R and before the call to R actually occurs by decrementing the top of stack pointer.

2.11 Temporary Variables

Now consider a clause that contains only one goal, but possibly several variables. In this case, all the variables will be safe (since they either occur first in the head, or occur in only one goal). The environment trimming principle above would allow all of the variables to be deallocated before the first (and only) call. What's more, the RR register does not need to be saved at all if we use last call optimization. Hence, no environment need ever be created for such a clause. However, some space is needed for the variables during head matching, and filling of the argument registers. In this case we use a global array of temporary variables (T0..Tn). Even when an environment is required (e.g. the clause contains more than one goal), any safe variables that occur only in the head, and possibly the first goal, can be assigned to a temporary variable register. New opcodes required to manipulate temporary variables include:

instruction	effect
getTVar T<n> from A<m>	Unify T<n> with the contents of argument register A<m>. Note that A<m> is either an atom term, or a reference term containing a reference to an <undef> term on the stack.
putTVar T<n> into A<m>	Put the term T<n> into A<m>.
uniTVar T<n>	Unify T<n> with the term which the structure register S points to.

2.12 Variable Predicates

In Prolog, a variable may occur in the predicate name position in a goal, as long as that variable is instantiated to a predicate name at run time when that goal is executed. This is often call a meta-predicate; we will simply call it a variable predicate. For example, consider the following Prolog code:

```
(assert
   ((P a b))
   ((Q a b))
   ((R b b))
    ((exec ?pred) if (?pred  a  b))
)
```

When the exec predicate is called, its argument will have to be instantiated to an atom, otherwise an error occurs. If ?pred is instantiated to P or Q, the call will succeed; if ?pred is instantiated to R, it will fail.

Where a variable predicate occurs in a clause, the usual call opcode is replaced by a callVarPred opcode:

instruction	effect
callVarPred V<n>	If variable V<n> is not instantiated to a predicate atom, generate error. Else, set RR to the instruction following the call, and set PC to the start of the code for the predicate

2.13 Variable Goals

In Prolog, a variable may occur in the goal position in a clause, as long as that variable is instantiated to a goal at run time when that clause is executed. This is often call a meta-predicate; we will simply call it a variable goal. For example, consider the following Prolog code:

```
(assert
   ((P a b))
   ((Q))
   ((R b b))
    ((exec ?goal) if ?goal)
)
```

When the exec predicate is called, its argument will have to be instantiated to a goal or an error occurs. If ?goal is instantiated to (P a b), the call will succeed; if ?goal is instantiated to (R a b), it will fail. Note that (P a b) and (R a b) are lists in this context. In addition to allowing a list as a goal, VPI Prolog also allows atoms and functor terms

as goals in a variable goal. For example, the following calls will also succeed:

```
(exec Q)
(exec [P a b])
```

Where a variable goal occurs in a clause, the usual call opcode is replaced by a callVarPred opcode:

instruction	effect
callVarGoal V<n>	If variable V<n> is not instantiated to a proper list, predicate atom, or functor term, generate error. Else, place arguments in the argument registers, set RR to the instruction following the call, and set PC to the start of the code for the predicate

2.14 Repeat

The repeat predicate does nothing more than create a choice point, where the code position after backtracking to the choice point created is the instruction following the repeat. Only a cut in the clause containing the repeat predicate can remove the choice point. The following opcode implements the repeat predicate:

instruction	effect
repeat	Create a choice point, setting the FR field to point to the following instruction.

2.15 Assert and Retract

At run-time, when an assert or retract is encountered by the PLM, the following kinds of terms are allowed as arguments to the assert and retract predicates:

1. An atom. The atom must be a known predicate name, and the predicate must have arity 0.

2. A functor term. The principal functor must be a known predicate name, and its arity must be equal to the number of arguments present in the functor term.

3. A list. In an assert, this list must be a proper list. In the case of a retract, this list may be a partial list. Its head must be either:

    a. A functor term meeting the conditions in 2. above.

    b. A proper list whose first element is an atom, which is the name of a known predicate, and whose length is n+1, where n is the arity of the predicate.

    The second member of the list, if present, must be the atom 'if'. Further members of the list must be either terms meeting requirements a and b above, or one of:

    c. A variable - representing a variable goal.

d. A proper list whose first element is an variable - representing a variable predicate.

There is no restriction on the form of the term which exists in the source code except that it must be possible, by instantiating zero or more variables, to achieve one of the above forms at run-time.

At run-time, the low-level code which implements an assert or retract requires the following:

1. The predicate number of the head atom. This is an index into a global 'structure table' in which is stored information about each predicate and functor known by the system.

2. Terms representing each head argument.

3. A term representing the tail (everything but the head) of the clause.

If the head of the clause is a functor term or proper list, then the code which is generated can include this information. If not (for example, if the thing to be asserted is just a variable, which will be instantiated to a clause at run-time), then the code generated cannot include this information. Asserts and retracts that allow inclusion in the code of the predicate number are called optimized asserts and retracts. The four new opcodes required to implement assert and retract are:

instruction	effect
assert <kind>	Split the clause in A0, then execute the assertOpt code
retract <kind>	Split the clause in A0, then execute the retractOpt code
assertOpt <kind>, <prednum>	see below
retractOpt <kind>, <prednum>	see below

When a plain assert or retract opcode is encountered, clause splitting occurs. Clause splitting generates a predicate number, n terms representing the head arguments, where n is the arity of the predicate, and another term representing the tail of the clause. After clause splitting, the same process used by the optimized assert and retract can be used to accomplish the assert or retract. Clause splitting can also result in an error condition if the conditions outlined at the beginning of this section are not met.

When an assertOpt is encountered, the predicate number is used to determine the arity n of the predicate. The registers A0 thru A(n-1) contain the head arguments. Register An contains a term representing the tail of the clause to be asserted (if there are no subgoals, then An contains the atom nil). If the Prolog flag assertcheck is set, a check is made for existing choice points for the given predicate; there should be none, else an error condition is raised. If the Prolog flag staticcheck is set, a check is made to determine whether the given predicate is a static predicate; if so, an error condition is raised. If no error occurs, PLM code is generated and added to the global rule base.

When a retractOpt is encountered, the predicate number is used to determine the arity n of the predicate. The registers A0 thru A(n-1) contain the head arguments. Register An contains a term representing the tail of the clause to be retracted (if there are no subgoals, then An contains the atom nil). If the Prolog flag retractcheck is set, a check is made for existing choice points for the given predicate - there should be none, else an error condition is raised. If the Prolog flag staticcheck is set, a check is made to determine whether the given predicate is a static predicate; if so, an error condition is raised. If no error occurs, the first clause (in PLM code form) which unifies with the terms in A0..An is deleted from the global rule base.

2.16 Rule Indexing

Consider a predicate 'capital' which records the capitals of the countries of the world. Some instances of the 'capital' predicate would be:

```
(capital  "United States"  "Washington, DC")
(capital  France  Paris)
(capital  England  London)
...etc.
```

Now consider the query:

```
(capital  India  ?cap)
```

In the PLM, as described so far, the list of 'capital' clauses would be searched linearly until a match is found. If the list is very long, this search takes time proportional to the length of the list since half of the list must be searched in the average case. PLM rule indexing employs a hash table technique to locate the matching clause quickly.

After the last clause has been asserted for a predicate, an indexing block is generated for a given predicate if 1) the arity of the predicate is not zero, 2) there are at least 2 clauses for the predicate, 3) the first argument in the head of all clauses is a non-variable, and 4) the predicate has not been declared as dynamic (which would imply the existance of run-time asserts and/or retracts for the predicate).

An indexing block consists of a switch block, followed by a series of hash tables, followed by a series of try blocks. At run-time, a call to an indexed predicate causes the PC to point to the switch block for the predicate. The switch block transfers control by setting the PC, depending on the type of term in the first argument, and generates a key from the term. If the first argument is a variable, the PC is set to point to the first clause for the given predicate. If there is no clause defined with that term type as its first argument, backtracking occurs; if there is only one clause with that term type as its first argument, the PC is set to point to that clause (the PC, however, will point to the instruction after the firstClause, interClause, or lastClause instruction, so no choice point will be created); otherwise, the PC is set to point to a hash table for that term type.

A hash table begins with an initial 'hash' instruction, which has the hash table size as one of its arguments. The previously generated key, and the hash table size, are used to index into the hash table, obtaining a new address for the PC to point to. This address will be either the address of a clause (actually, the address past the initial firstClause, interClause, or lastClause instruction), or the address of a try block.

A try block is needed only if, even after indexing, more than one clause still might match the first argument of the call. A try block consists of an initial firstTry instruction, a (possibly empty) series if interTry instructions, followed by a final lastTry instruction. These instructions are analogous to the instructions firstClause, interClause, and lastClause, except that firstTry and interTry set the FR field of a choice point to the next try instruction.

The new opcodes required for rule indexing are:

instruction	effect
switch <5 addresses>	Dereference the term in A0. If a function, evaluate it. Then set PC to one of the 5 following addresses depending on whether the term in A0 is 1) a variable, 2) a list, 3) a number, 4) an atom, or 5) a functor term. If the address is NULL, backtrack. Otherwise, if the first argument is an atom, number, or functor term, generate a key for hashing.
hash <size> <table of addresses>	Calculate (key MOD <size>), and use to index into the following table. Set PC to the given table entry. If NULL, backtrack.
firstTry <address>	Create a new choice point, put the address of the folowing instruction into FR, and set PC to <address>.
interTry <address>	Restore a previous state from the choice point pointed to by B, put the address of the following instruction into FR, and set PC to <address>.
lastTry <address>	Restore a previous state from the choice point pointed to by B and remove the choice point from the stack (restoring B from the choice point), and set PC to <address>.

2.17 Debugging Facilities

Debugging facilities include the ability to "step" into a predicate call, completely "solve" a predicate call, and run until a particular predicate is invoked. This facility relies on the existence of a "debug" instruction

before each predicate call, including before any "not" or "or" goal. The format of the debug instruction is:

instruction	effect
debug <kind>	If the BOOLEAN variable OneStep is set, then temporarily terminate Prolog execution is such a way that execution may be resumed.

Chapter 3

Using the VPI Prolog Compiler

This chapter describes using VPI Prolog, including how to start VPI Prolog, how to load PROLOG source files, and methods for debugging programs. VPI Prolog is an interactive environment, so programming is easier than programming with a compiled language. Programs are more easily debugged in such an interactive environment.

3.1 Calling VPI Prolog and Loading Files

The command line to invoke VPI Prolog is:

```
$ prolog { <filename> }
```

where $ is the operating system prompt (which may differ between computers). VPI Prolog displays several lines of information, then reads the PROLOG source code contained in the files specified as arguments. By convention, PROLOG source files have a ".hc" file name extension (which stands for Horn Clause, the logic upon which the PROLOG language is based). You need not include this extension when specifying source file names, Prolog will add the extension before opening the files. The input files may contain commands, assertions, and queries. If the quit command is not encountered in the source files, VPI Prolog then presents the prompt ?-, indicating that it is ready for input from the

user. Commands consist of a command word, possibly followed by arguments (for example the name of a file to be operated on), all on a single line. To quit Prolog, simply enter the command "quit" at the prompt.

As an example of a command, consider the "echo" command. The user can enter the command:

```
?- echo <filename>
```

and any text appearing on the terminal screen thereafter will be placed in a file called "<filename>.ech". Echoing can be disabled later with the command

```
?- echo off
```

VPI Prolog input may be placed in an external file using any text editor, for example, **vi** on Unix systems. There are three basic methods for retrieving input from external files:

1. Include the external file's name on the command line. For example, the following will invoke VPI Prolog, reading in the file "start.hc" before displaying the Prolog prompt:

   ```
   $ prolog  start
   ```

2. Use the "consult" command at the prompt to read an external file. The format of the consult command is:

   ```
   consult <filename> [verbose]
   ```

For example, the command:

```
?- consult  readme
```

will read input from a file named "readme.hc". Note that the ".hc" ending is automatically appended if not present. Input will be read by VPI Prolog as if typed at the prompt, and any input which is valid when typed at the prompt is valid in the file. If the optional 'verbose' argument is present, input will be displayed on the terminal as it is being read.

3. Invoke "consult" as a predicate, calling it as part of a Prolog program. However, the file being consulted this way may not contain any queries.

3.2 Environments

An environment consists of a set of clauses. Environments may be saved during one session, and then later retrieved during a later session. This is much faster than using consult again since parsing is completely bypassed. An environment is stored in a <name>.env file on your disk (it is not a human-readable ASCII file). When Prolog is running, the set of all currently defined rules is called the current environment, and has a name. Initially, the current environment name is set to the empty string. When the user enters the "quit" command, if the current environment has a non-empty name, the environment is automatically saved. The current environment name, if not empty, is displayed as part of the Prolog prompt.

The following commands control saving and loading of environments:

setenv <name> - set the current environment name

saveenv - save the current environment in the file <current env. name>.env & set the current environment name to the empty string

saveenv <name> - save the current environment in the file <name>.env

clearenv - abolish all rules & set current environment name to the empty string

loadenv <name> - restore environment with the given name

The easiest way to load a previously saved environment is to include the name of the environment file on the command line. In this case, the ".env" ending must be included to distinguish it from a PROLOG source file. The environment is retrieved before any source files are read.

3.3 Self-Starting Programs

After Prolog has loaded all the files included on the command line (including an environment file, if specified), Prolog checks to see if a zero-argument predicate named beginExecution has been defined. If so, it is executed next. This allows a program to be self-starting. Of course, Prolog source files may contain queries, so that loading a source file may result in execution starting without user intervention; however,

environments are simply sets of clauses, so this capability allows the programmer to create an environment, and have execution begin automatically simply by invoking Prolog with the name of the environment file on the command line.

3.4 Search Directories

When searching for PROLOG source files and environment files, VPI Prolog first searches the current directory (defined by the local operating system). You may specify additional directories for Prolog to search. These directories may be specified on the command line as noted below. If PROLOG source files whose names appear on the command line are not in the current directory, the directory names must be specified on the command line. In addition, the addsearchdir command can be used to specify search directories:

```
addsearchdir { <directory path> }
```

For example:

```
?- addsearchdir /usr/bob/prologfiles      ; Unix example
?- addsearchdir user:[roach.deighan]      ; VAX example
```

The format of a directory path depends on the host operating system. Prolog simply concatenates the given directory path with the file name and extension, and asks the operating system to open the file with that name. Note, however, that on Unix systems, Prolog knows to separate

the directory path and file name with a "/" if not present at the end of the directory name.

3.5 Command-Line Options

The following command line options (presented here in the style used on Unix and VAX VMS systems) may be included on the command line:

-n	- don't display initial help information
-d <directory>	- add <directory> to the search directory list - multiple directories may be specified if separated by commas
-h <size>	- allocate a heap array of <size> bytes
-s <size>	- allocate a stack array of <size> bytes
-e	- use Edinburgh syntax
-k	- quit if any error occurs; useful for batch execution

The heap and stack size may be an integer, optionally followed by a K (for kilobytes) or M (for megabytes). For example, to specify a 500 kilobyte stack, and a 2 megabyte heap, you can invoke Prolog like this:

```
$ prolog -s500K -h2M
```

To display the current heap and stack sizes in VPI Prolog, use the following command:

```
?- stackusage
```

3.6 On-line help

On-line help is available via the help command. The following invocations of help should help you get started with the on-line help system:

```
?- help              ; equivalent to "help help"
?- help topics
?- help general
?- help syntax
?- help builtins
```

Help is available for a number of topics and for each command, builtin predicate, and builtin function. Typing "help topics" displays a list of topics for which help is available.

For on-line help to be available, a file named "prolog.hlp" must be found by Prolog. Prolog first checks the current directory for the file "prolog.hlp". If not found, VPI Prolog searches a directory which is specified at compile time, and varies depending on which machine Prolog is compiled for. Normally, "prolog.hlp" will be in this directory, and the user need not have his own copy of the help file.

3.7 Interrupting Prolog execution

Programs with infinite loops can be easily created; a bug known as "left recursion" often occurs in PROLOG programs. To help the

programmer, a means of breaking computations has been provided. The C language allows signals to be defined to perform specific actions. In VPI Prolog, the signal called SIGINT interrupts a computation and displays the VPI Prolog prompt. The user may then continue the computation by entering a carriage return, or enter the 'debug' command and step through the currently executing goals. The SIGINT break allows the user to take stock of the situation and decide whether an infinite loop has occurred.

The method for generating the signal SIGINT is system-dependent. On UNIX systems, the command (at the UNIX prompt):

```
$ stty intr ^B
```

sets the keyboard character Ctrl-B to generate the SIGINT signal.

If SIGINT does not work, the user may resort to the SIGQUIT signal. This terminates Prolog altogether, and the next thing the user sees is the system prompt.

On UNIX systems, the command (at the UNIX prompt):

```
$ stty quit ^C
```

sets the keyboard character Ctrl-C to generate the SIGQUIT signal.

Chapter 4

Programming in VPI Prolog

In this chapter, the syntax and semantics of the language are presented. The reader should get some idea of how programs are written as well as the capabilities of the language. Learning is best achieved by doing, so the user is encouraged to run Prolog and to try some simple programs as the chapter progresses. The syntax of VPI Prolog is not the same as Edinburgh versions, but this should not be bothersome due to the simple structure of the rules.

4.1 Defining Prolog Rules

Programs in Prolog are made up of collections of rules, each of which has a particularly simple format. To define a collection of rules, the source file should contain one or more sections of the following form:

(assert { <rule> })

Each rule has the form:

(<head> [if { <subgoal> }])

There may be zero subgoals. The <head> and each <subgoal> are called relational expressions. The format of a relational expression is:

(<predicate name> { <term> })

The <term>s are called arguments (there may be zero arguments). Consider this rule, for example,

```
((zebra ?x)  if
     (ungulate  ?x)
     (blackstriped  ?x))
```

which says that ?x is a zebra if ?x can be shown to be an ungulate and if ?x can be shown to have black stripes (?x is a PROLOG variable; in VPI Prolog, variables begin with a question mark). Notice that the names of relations and the number of arguments are constructed by the programmer to fit the problem domain. The outer parentheses demarcate the boundaries of a rule; inside the outer parentheses are relations, each of which must also be enclosed by parentheses.

The zebra rule can be viewed in several different ways. The left side can be viewed as a goal and the right side as a list of subgoals to be reached or demonstrated. Alternatively, it can be viewed as a rewriting system in which the left hand side can be rewritten as the elements on the right hand side. In either case, a pattern on the left must be matched and replaced by new patterns from the right. The rule syntax with relations on both sides of the "if" is the most general form, but one other form derives from it. With no right hand side a rule looks like

```
(<goal> if )
```

and means that the "goal" is defined to be a fact, since when the left hand side is matched no subgoals need be proven to demonstrate the

truth of the left hand side. Note that the "if" part of such a rule is optional in this case. Thus, facts may be expressed as:

```
((blackstriped henry) if )
```

or as

```
((blackstriped henry))
```

4.2 Static vs. Dynamic Predicates

By default, when VPI Prolog sees a rule defining a given predicate, it expects to see a series of consecutive rules for that given predicate. Once a command, query, end-of-file, or rule for another predicate is encountered, Prolog considers that predicate completely defined, and will not allow any more rules to be defined for that predicate. Also, none of the rules for that predicate may be retracted (i.e. removed). Such a predicate is called static. If the user wants a particular predicate to be dynamic, so that rules for that predicate may be asserted and/or retracted at any time (including run-time), the user must declare the predicate to be dynamic using the command

```
dynamic { <predicate> }
```

Also, if a call to a predicate is attempted at run-time, and no rules exist for that predicate, an error will be generated unless the given predicate is dynamic.

Often, the user wants to define a predicate in several files, and with definitions of other predicates interspersed with its definitions. In this case, he should use the command

multifile { <predicate> }

to declare that the predicate(s) will be defined in multiple files. When all rules have been defined for that predicate, he can use the command

closepred { <predicate> }

to indicate that all rules for that predicate have been defined.

The user may also declare a predicate to be static directly, using the command:

static { <predicate> }

This tells Prolog that the predicate will not be used dynamically, and allows the programmer to list all predicates defined in a file at the top of the file, in one of the commands: static, dynamic, or multifile, thus creating a contents list for the file.

4.3 Term Syntax

A term may be an atom or a number (both of which are referred to as being atomic), or a compound term. A term represents some entity about which a logical statement may be made. In the above example, henry is a term representing an animal about which we may make the logical statements "henry is blackstriped" and "henry is a zebra".

An atom (or string atom) is a value consisting of a sequence of characters which may be enclosed in single or double quotes, and must be enclosed in quotes if it contains any of the following characters: '.', '(', ')', '[', ']', ':', ';', a blank, a quote mark, or an apostrophe. It must also be enclosed in quotes if it starts with a '?' so it will not be taken as a variable. In the preceding example, henry is an atom. Other lexical conventions concerning atoms such as including control characters in an atom are covered in Appendix 1.

The atom nil can also be written as (), i.e. a left parenthesis followed by a right parenthesis. This is because nil is used to represent an empty list in rules which manipulate lists.

A number may contain a decimal point, and may include an 'E' or 'e' followed by an exponent. If a number contains a decimal point, it must include a digit both before and after the decimal point. A number is never equal to an atom. For example, the atom "123" is taken to be an atom of length 3 because of the surrounding quotes, and is not the same as the number 123. Functions are available for converting a number to an atom and vice versa.

A term may also be a compound term. There are 2 kinds of compound terms: functor terms and lists. A functor term consists of an atom, called the functor, followed by zero or more terms, called the arguments, all enclosed in square brackets. For example, the following is a functor term:

[student john CS]

Any two functor terms with the same functor must always be followed by the same number of arguments (called the arity of the functor). VPI Prolog determines the arity of each functor from the first time is sees that functor used in a functor term. Internally, functor terms can be stored compactly since the size of the term is fixed.

A functor term may be evaluable. A functor term with an evaluable functor is evaluated (i.e., replaced with the result of evaluating the term as a function) when it must be unified with another non-variable term. For example, there is a function called "strcat" which returns the concatenation of its arguments. The term

[strcat abc def]

is recognized as an evaluable functor term since "strcat" is the name of one of the built-in functions, and when it is evaluated, the result is the atom "abcdef".

A list consists of an arbitrarily long sequence of terms enclosed in parentheses. A list is a general purpose method of grouping data elements together. For example,

(eggs butter milk (bluecheese cheddarcheese) cereal)

is a grocery shopping list containing a sublist of cheeses to buy. Arbitrary nesting of lists is permitted; thus any level of list complexity can be achieved. Examples of other lists include vocabulary lists, class

rosters, airline flights between cities, library catalogs, genealogy tables, parts inventories, legal chess moves from a given position, television shows for a given day, and so on. Lists are an important tool for Prolog programmers.

A variable is a term that looks like an atom, but starts with a '?' and cannot be enclosed in quotes. A question mark by itself: ?, is called a void variable, and is not considered the same as any other variable, even another ? in the same clause. A variable represents an unknown quantity, whose value may become known during execution of a goal.

A term is called a ground term if it contains no variables. Atoms and numbers are examples of ground terms. A functor term is a ground term if each of its arguments are ground terms, and a list is a ground term if every term in the list is a ground term.

4.4 Variable Predicates and Variable Goals

There are 2 additional forms which are allowed as a subgoal, i.e. after the "if" . The first is a goal in which the predicate name is replaced by a variable, e.g.

```
(assert
    ((P ?x ?y) if
        (?x a b)
        )
    )
```

At run time, ?x must be bound to an atom, otherwise Prolog stops and issues an error message. This implies that when the rule is defined, the variable ?x must occur earlier in the rule, otherwise it could never be bound when used. The second form is one where the entire subgoal is replaced by a variable:

```
(assert
    ((P ?x ?y) if
        ?x
        )
    )
```

in this case, the variable must be bound to a list or functor term at run time, and the first element of that list, or the principle functor of the functor term, must be an atom. Once again, this implies that the variable must occur earlier in the rule.

4.5 Query Syntax

The following should be typed when the user wants to invoke VPI Prolog to compute an answer :

({ <goal > })

Essentially, the Prolog system is being asked to use previously defined rules to establish the conjunction of the subgoals. A typical program has the following form (a semicolon (';') begins a comment; Prolog ignores it and any characters remaining on that line) :

```
(assert          ; assert some rules
    (<head>)
    (<head> if )
    (<head> if <subgoal1> ... <subgoaln>)
    ...
 )

(<goal1> <goal2> ... ) ; goals to solve
```

The construct

(assert { <rule> })

defines the rules to be used, and the construct

({ <goal> })

is called a query and defines the questions (that is, goals) that Prolog answers using the rules. To simplify entering the query, the following short forms may be used: 1) if the query consists of a single goal, the outer pair of parentheses surrounding the query may be omitted and 2) if the query consists of a single goal, and the entire goal is entered on a single line, the parentheses surrounding the goal may also be omitted.

4.6 The VPI Prolog Debugger

Enter the VPI Prolog debugger via the debug command:

```
?- debug
D-
```

The prompt will change to reflect the fact that you are now in debug mode. In debug mode, you can use the following commands to step through your program:

command	effect
step [<int> [verbose]]	match the next rule; optionally, repeat <int> times. Verbose controls printing of goal stacks.
solve [<int>]	completely solve the next goal; optionally, repeat <int> times.
runto <pred> [<int>]	run until <pred> is the next goal, or until the <int>th rule for <pred> is being tried
choicepoints	display the list of saved choicepoints
callchain	display the list of active goals - i.e. goals invoked but not yet solved
query	display the original query, with currently instantiated variables
list <pred> [<int>]	display rules for the predicate <pred> or optionally, rule number <int> for predicate <pred>
rerun	restart the current query; asserted clauses are not automatically retracted.

Step will execute a single head matching step. Solve will take the next goal, and execute steps until that goal, any subgoals generated by that goal and their subgoals are removed from the goal list. In other words, if you take the current goal list, solve will either run until the goal list becomes that list with the topmost goal removed, or until the original topmost goal fails. Runto will execute steps until the given predicate is

the name of the next goal to be executed. Rerun will reset the current query and start executing it all over again. These commands are available only in debug mode.

Chapter 5

Database Capabilities

PROLOG is a relational programming language. Facts declared in PROLOG look like and function the same way that facts in large, relational databases do. Unfortunately, many of the powerful facilities of a relational database are not normally built into a PROLOG compiler. This chapter documents how the VPI Prolog system has been extended to include the facilities one normally finds in a relational database: most importantly, B+ tree indexing. Relational databases are not normally built within a powerful programming language like PROLOG and must therefore include a separate query processor. Using the query answering capabilities of VPI Prolog, however, we get a much more powerful query processor. The PROLOG computational model, moreover, does not require that facts be placed in third normal form in order to process queries efficiently. The resulting system therefore combines relational database and PROLOG relational language technology. This chapter describes how to use the database facilities built into VPI Prolog.

The specific relational database technology that has been incorporated into VPI Prolog includes a buffered demand paging system and a B+ tree indexing system. The user need not understand the internal details of paging and buffering; the details are transparent to

the programmer. In order to use the indexing system effectively, however, the programmer needs to know a few simple facts about the internal implementation of the system.

B+ tree indexing works by building an indexing structure in memory in addition to the actual data being indexed. Each argument of a relation may have its own separate indexing structure. The amount of storage being used can become considerable, possibly slowing the user's application due to excessive paging. We have therefore provided a facility for the user to limit indexing to a selected set of the arguments of the relation. Normally, a relation does not need to be indexed on all arguments, so large amounts of storage can be saved in this fashion. The ability to limit indexing structures is documented below in the "loaddb" command.

B+ tree indexing works by indexing one argument of a relation at a time. When a database goal is attempted, the database system selects an instantiated argument to use for indexing. If all arguments are variables, indexing cannot be used and all database facts for the given predicate are searched.

Although the processing of large databases is performed by routines separate from the normal PROLOG processor, asserting, retrieving, and retracting database facts, saving the compiled image, printing, etc. are entirely transparent and require no special programmer attention. Due to the large size of databases, however, we have designed a compact

database file input format and created a special predicate, loaddb (documented below), to load these files.

5.1 Asserting Database Facts

As with normal Prolog rules, database facts can be created by asserting the fact directly. However, some mechanism is needed to distinguish database facts from normal Prolog facts. This is accomplished by calling the predicate createdb, giving the predicate name as argument, before asserting any facts to the database. For example, a small database containing 3 facts can be created as follows:

```
((createdb capital))
(assert
  ((capital Virginia Richmond))
  ((capital "North  Carolina" Raleigh))
  ((capital Maryland Baltimore))
   )
```

If the programmer asserts to a given predicate without calling createdb, then later calls createdb with the given predicate, this is a programming error, and an error message will be displayed.

The loaddb predicate

The following predicate will load an entire database from a file formatted as described below:

```
(loaddb <+atom> <+atom> <+atom> [ <+atom> ] )
```

Arg1 is the predicate name; arg2 is the filename of the file containing the data. The default file name ending for a database file is ".db", and VPI Prolog will automatically append this ending to the given filename if not present.

Arg3 is called an index string, and it specifies which arguments will have an associated B+ tree for indexing. Arg4, which may be omitted, is called the type string, and is used during database loading to check the types of the arguments. The index and type strings each have one character per argument as follows:

index string characters

y | Y	yes, index on arguments in this position.
n | N	no, do not index on arguments in this position.

type string characters

x	perform no checking on the argument in this position.
N	the argument in this position must be a number.
L	the argument in this position must be a list.
A	the argument in this position must be an atom.

Type checking on the arguments is performed only during loading of the database. Prolog deduces the number of arguments from the length of the index string. It is not necessary that all the arguments for each fact be on a single line, except for the first line when using the first format. That is, when preparing an input file for a database relation with many arguments, the arguments may span two or more lines.

However, the last argument for each fact must be the last argument on its line.

The database input file must have the format:

```
<?term> <?term> ... <argn>
... etc.
```

where each line in the file contains a series of terms taken as arguments to the predicate. Each line becomes a separate fact in the database just as if the following were entered at the Prolog prompt:

```
?- ((createdb <predicate>))
?- (assert
    ((<predicate> <?term> <?term> ... <?term>))
    ((<predicate> <?term> <?term> ... <?term>))
    ... etc.
    )
```

An Example

Consider a database consisting of three arguments, where the first is a state name, the second is the state's population, and the third is a list of all cities in the state with population greater than 1,000,000. The following file (named state.db) is created:

```
"New York"   10000000    ("New York City" Buffalo)
California   50000000    ("Los Angeles"  "San Francisco")
Alaska       5000000     ()
```

Notice that an atom that contains blanks (or other special characters) must be enclosed in quotes. This database may be loaded with the query:

```
?-  ((loaddb state state.db ynn ANL))
```

Literature Cited

1. J. Backus, "Can programming be liberated from the von Neumann style? A functional style and its algebra of programs," Communications of the Association for Computing Machinery, September, 1977.

2. K. Bowen and R. Kowalski, "Amalgamating language and metalanguage," in K. L. Clark and S.-A. Tarnlund, Logic Programming, London: Academic Press, 1972.

3. W. Clocksin and C. Mellish, Programming in Prolog, Berlin: Springer, 1981.

4. H. Coelho and L. Pereira, Prolog By Example, Berlin: Springer, 1989.

5. R. Floyd, "Nondeterministic algorithms," Journal of the ACM, Oct. 1967, 636-644.

6. H. Gallaire and J. Minker, editors, Logic and Databases, New York: Plenum Press, 1978.

7. K. Godel, "On formally undecidable propositions of Principia Mathematica and related systems, I," see J. van Heijenoort, From Frege to Godel, A Sourcebook in Mathematical Logic, 1879-1931, Cambridge, MA: Harvard University Press.

8. R. Kowalski, Logic for Problem Solving, NY: North-Holland, 1979.

9. A. Newell and H. Simon, Human Problem Solving, Englewood Cliffs, NJ: Prentice-Hall, 1972.

10. E. Post, "Formal reductions of the genral combinatorial decision problem," American Journal of Mathematics, 1943, 197-268.

11. J. A. Robinson, "A machine-oriented logic based on the resolution principle," Journal of the ACM, January, 1965, 23-41.

12. E. Shortliffe, Computer-Based Medical Consultations: MYCIN, NY: Elsevier, 1976.

13. M. Stefik, "Planning with constraints," Artificial Intelligence Journal, 1981, 111-139.

14. D. Waltz, "Understanding line drawings of scenes with shadows," in P. Winston, The Psychology of Computer Vision, NY: McGraw-Hill, 1975.

15. David H. D. Warren, An Abstract Prolog Instruction Set , SRI International Technical Note 309, Oct. 1983.

16. R. Weyrauch, "Prolegomena to a theory of mechanized formal reasoning," Artificial Intelligence Journal, 1980, 133-170.

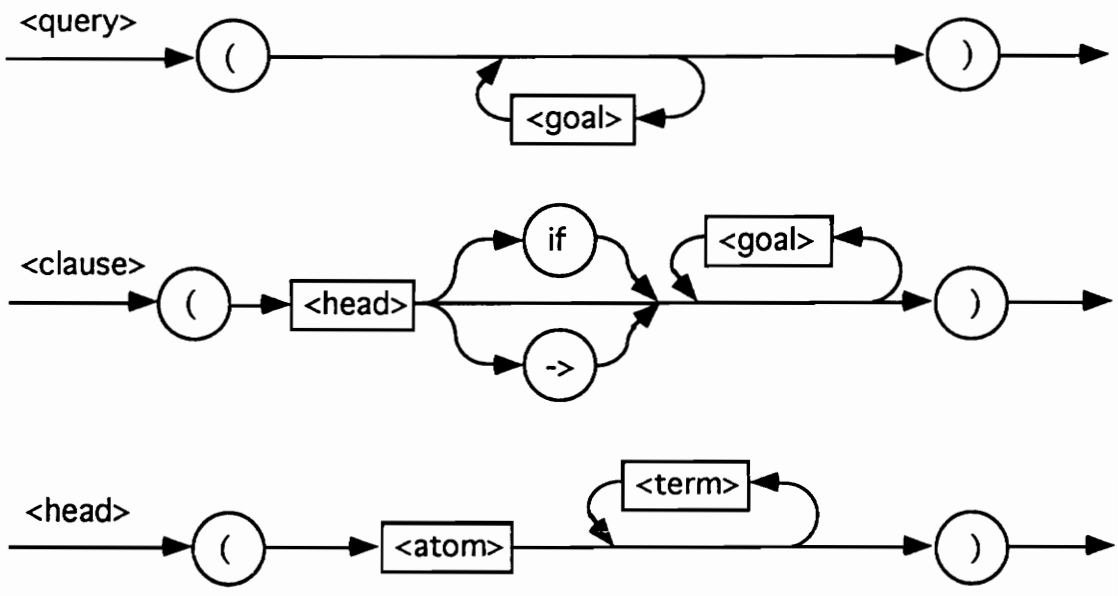
17. P. Winston and B. K. P. Horn, Lisp, Addison Wesley

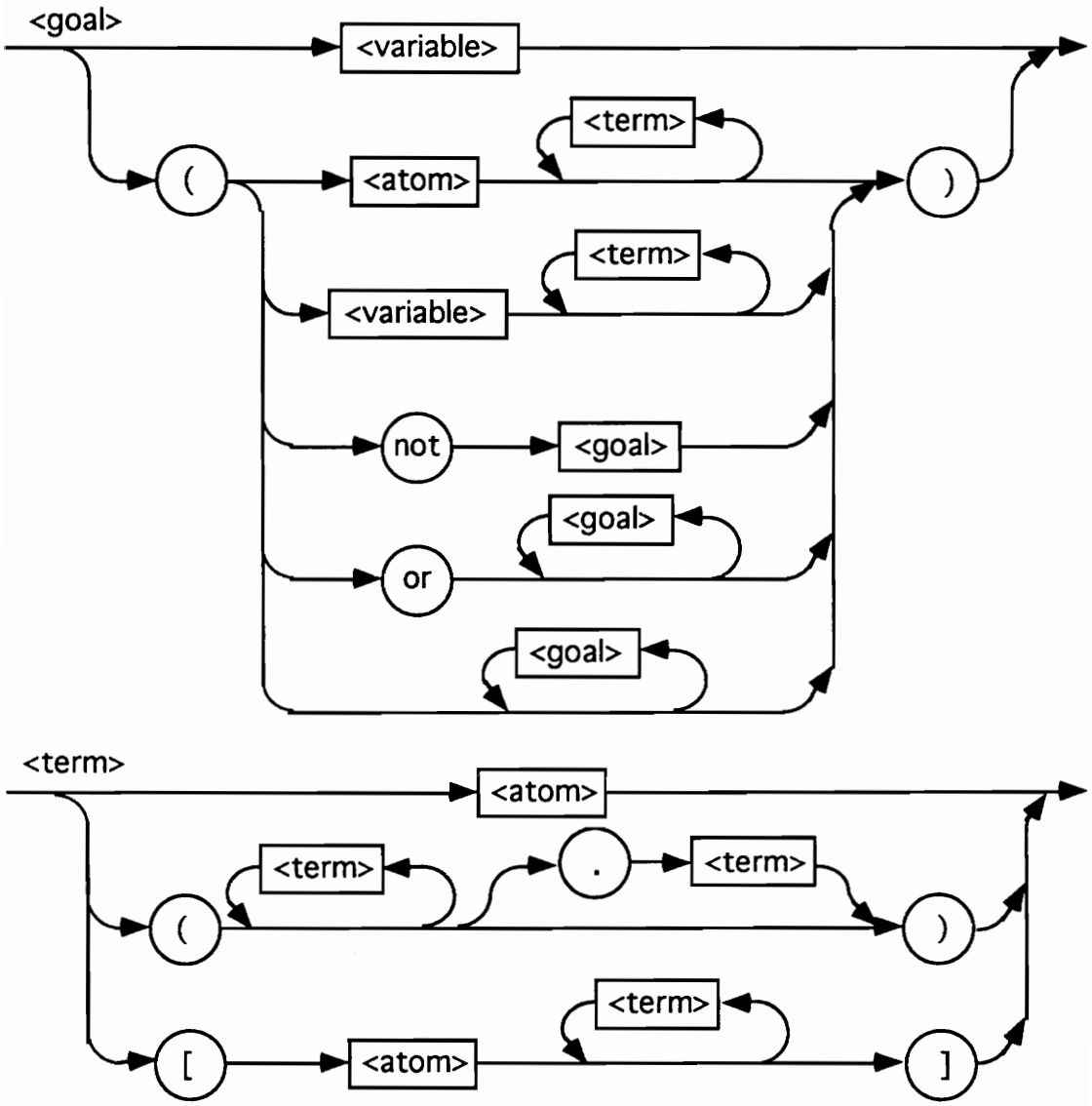
Appendix 1

VPI Prolog Syntax

The following syntax diagrams define valid constructs in VPI Prolog. Items within a circle or oval are to be taken literally. Items within a square or rectangle denote syntactic constructs defined elsewhere.

In each syntax diagram, the syntactic object described by the diagram appears on the left, over the incoming arrow. An instance of the object can be constructed by following the arrows and concatenating the objects encountered until the rightmost arrow is reached.

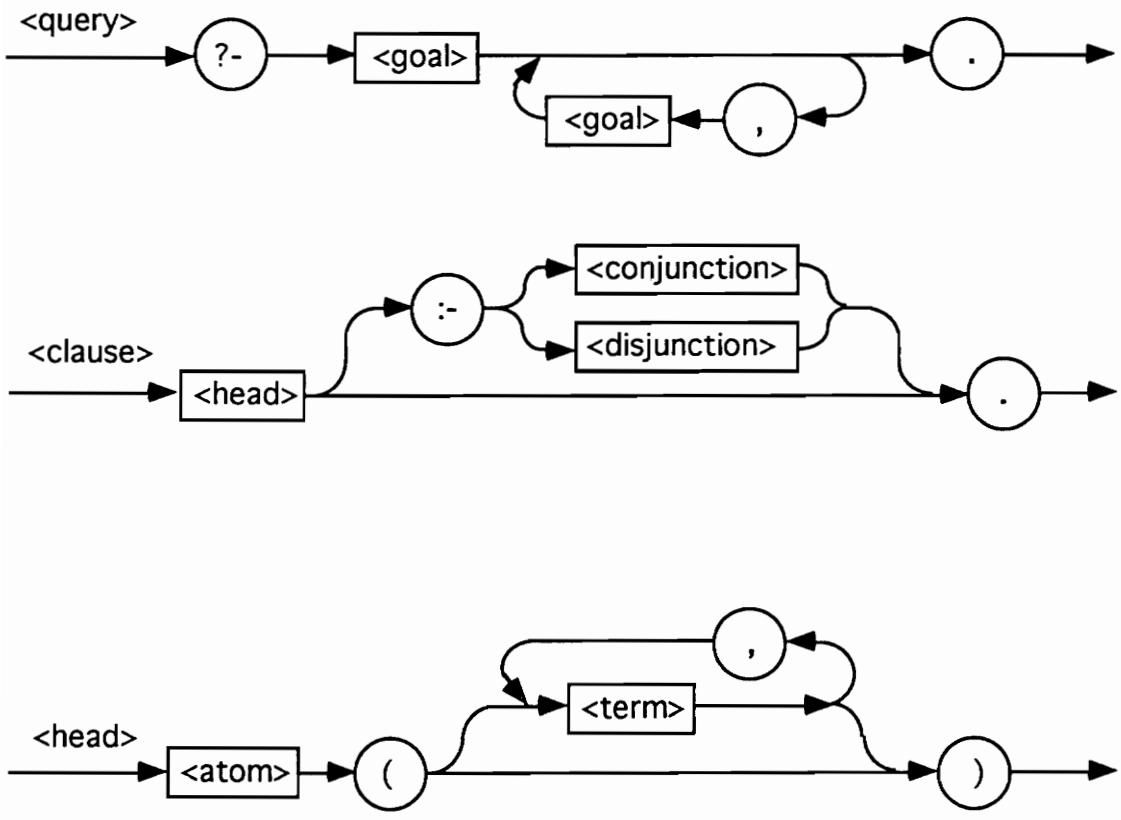


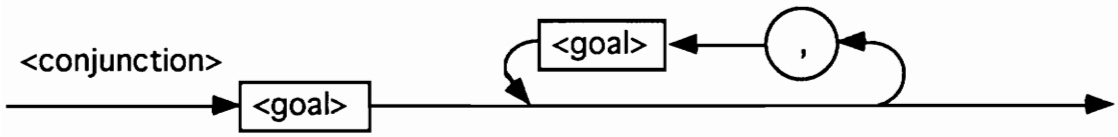

<goal>
<variable>
(
<atom>
<term>
)
<variable>
<term>
not
<goal>
or
<goal>
<goal>
<term>
<atom>
<term>
(
.
<term>
)
[
<atom>
<term>
]

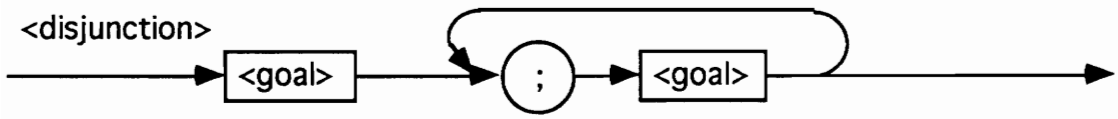
Appendix 2

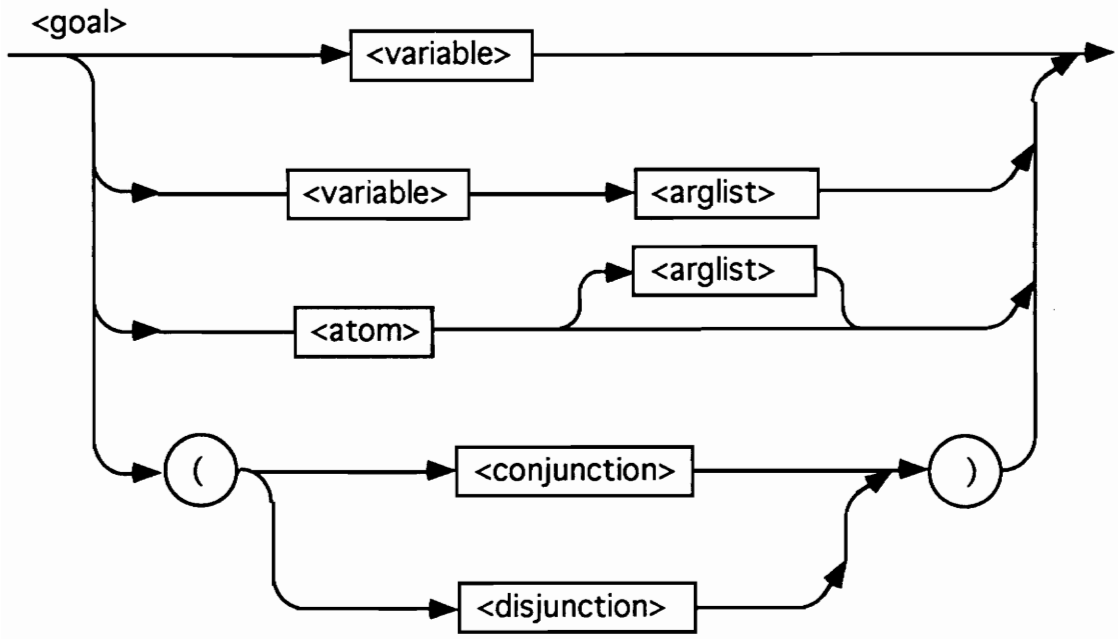
Edinburgh Syntax

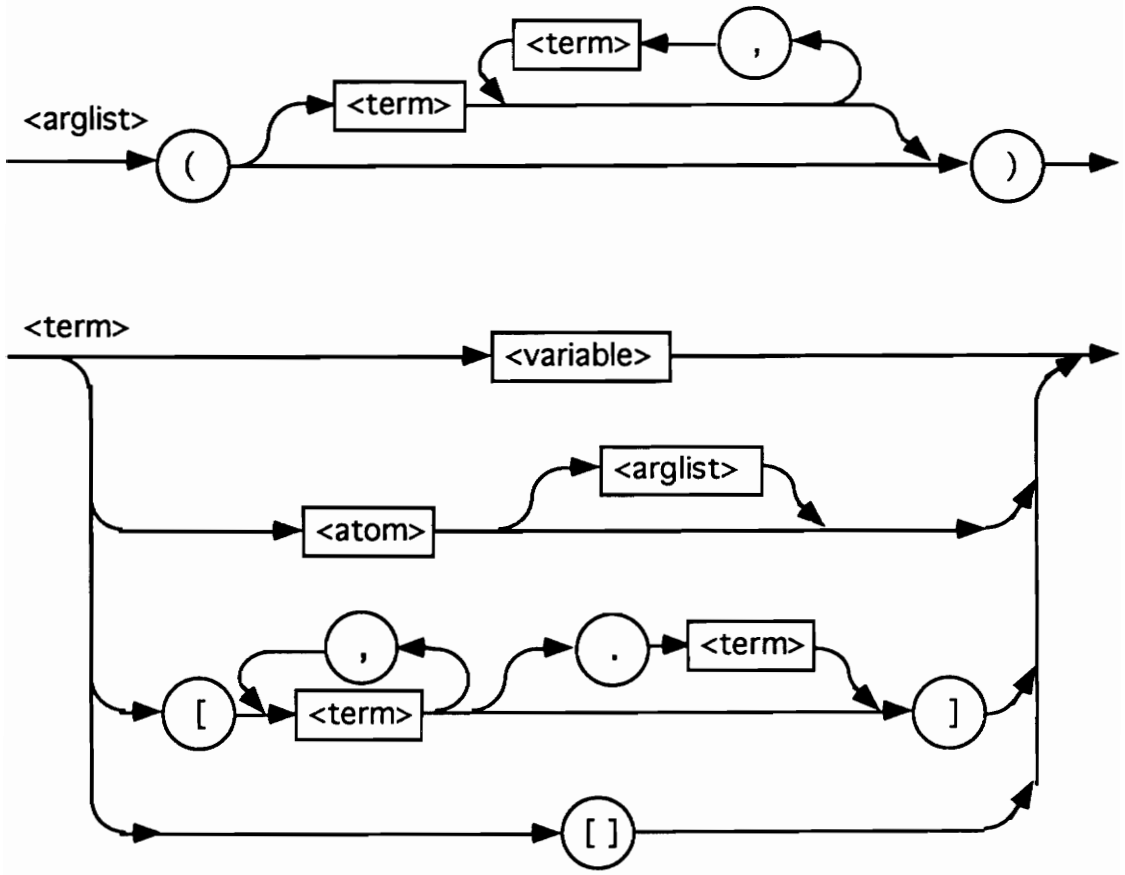
The following syntax diagrams define valid constructs in Edinburgh syntax. Items within a circle or oval are to be taken literally. Items within a square or rectangle denote syntactic constructs defined elsewhere.




<conjunction>
<goal>
<goal>
,


<disjunction>
<goal>
;
<goal>


<goal>
<variable>
<variable>
<arglist>
<arglist>
<atom>
(
<conjunction>
)
<disjunction>


<arglist>
(
<term>
<term>
,
)
<term>
<variable>
<atom>
<arglist>
[
<term>
,
.
<term>
]
[ ]