THE DESIGN AND IMPLEMENTATION OF A LANGUAGE ENVIRONMENT FOR EVALUATING THE PROGRAMMING TASK

by

Cyril Shiu-Chin Ku

Thesis submitted to the Faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

in

Computer Science and Applications

APPROVED:

Timothy E. Lindquist, Chairman

Roger W. Ehrich

Robert C. Williges

May, 1982
Blacksburg, Virginia

ACKNOWLEDGMENTS

It has been a long time since I started my college career in 1976. Thank God for all the gifts given to me, many difficulties have been overcome and several goals have been satisfied since then. A master's degree is another milestone in my life. Without the following sources of support, this milestone would never have been reached. It is time to give them a hearty thanks.

I am deeply indebted to my advisor, Dr. Timothy E. Lindquist, for his indispensable advice and valuable guidance without which, my thesis would not have been completed. And I am particularly grateful for his understanding, his patience, and his faith in me, which undoubtedly have encouraged and inspired me to greater effort in accomplishing my research.

I would like to express my appreciation to other members of my graduate committee: Dr. Roger W. Ehrich and Dr. Robert C. Williges, for their enlightening advice, and to Dr. H. Rex Hartson, Dr. John W. Roach, Joseph F. Maynard, Deborah H. Johnson, and all the people in the ONR research group, for their valuable suggestions.

I would also like to thank the Office of Naval Research for its support in carrying out my research. The research described in this thesis was supported by ONR under contract number N00014-81-K-0143 and work unit number NR SRO-101. The effort was support by the Engineering Psychology Programs, Office of Naval Research, under the technical direction of Dr. John J. O'Hare.

Now a few words for my parents. Their continual encouragement and emotional support have made the writing of my thesis a much easier task.

Finally, a very special thanks goes to my elder brother Sonny (Siu-chung). His perception, aspiration, and personality always exert immense influences on me. He has contributed a great deal to the shaping of my outlook on the world. Probably I would never have had a college degree without his love and concern. It is to him that this thesis is dedicated.

Cyril S. Ku
Blacksburg, Virginia
May 1982

TABLE OF CONTENTS

Chapter I

INTRODUCTION

1.1 DEFINITION OF PROGRAMMING ENVIRONMENT

A programming environment, which may include the software, hardware, managerial, or social environment, is a set of tools that the programmer uses to develop software. In this thesis, however, a narrower view of the environment is taken. It excludes the hardware, managerial, and social environments. Thus, a programming environment can be defined as an environment which consists of software tools integrated in a virtual computer environment that provides automated support covering different phases of a programming task. If the virtual computer environment is a high level programming language, then programming environment is also called language environment. Integration of tools means that the tools exist in a single environment, so that there is no explicit context switching like invoking an editor, which the user should get out of the editor before another software tool can be used.

1.2 EVOLUTION OF PROGRAMMING ENVIRONMENTS

The history of programming environments started with software tools dating back to the early years of digital computers in the 1940's. At first, programs were directly coded by hand and bound to absolute addresses in physical computer memory. Later, relative and symbolic assemblers appeared. Now, there are editors to create or modify a file, loaders to aid in running a program in relocatable format, linkers to resolve global symbols and enhance modular programming, and high level language compilers or interpreters to tailor different applications. Assemblers, run-time libraries, utility routines, and documentation aids are some of the software tools which are available in today's typical programming environments.

During the 1950's and early 1960's, computer time and computer hardware were more expensive than labor. There was little motivation to improve software tools in order to increase programmer productivity or improve software quality. Therefore, little improvement on the environment for programming occurred during these years.

In the mid-1960's, operating systems as we know them today began to evolve. Operating systems, ranging from batch to time-sharing, made possible more efficient and economical utilization of computer resources. During this period, the applicability of computers increased and hardware technologies advanced. A decline in the cost of hardware, together with an increase in the cost of software leaves us in a situation vastly different from the 1950's and 1960's. A gap exists between computer hardware development and software technology. Hardware advances are made at an accelerated pace while the software development technology remains in a stagnant stage. Thus, software systems are usually characterized as unreliable, unextendable, over-cost and behind schedule, and in constant need of maintenance. This is known as the "software crisis".

In the late 1960's, the term "software engineering" was born to describe attempts to use engineering principles to produce economical, efficient, and quality software [BUXT70, NAUR69]. In recent years, because of the software crisis, there has been great interest in increasing the productivity of programmers and the quality of software produced. The improvement of software tools (i.e., the development of more effective programming environments) would be a step toward a solution to the crisis. Although a good environment is hard to define, it must ease the effort required to develop quality programs. "Easiness" implies, among other things, a user-friendly system, and "quality" means the degree to which it satisfies such intangible issues as reliability, verifiability, modifiability, maintainability, transferability, and efficiency.

Early software tools were simple, performing only one function, and usually focusing on the coding activity. Current tools are more complex, multi-functional, covering different phases of the software life cycle. Unfortunately, usually independent of other tools, current software tools define their own vocabulary, and focus their support on isolated aspects of the software life cycle. Thus, these tools can not easily be combined into a consistent programming environment. Current research efforts are trying to integrate different tools into a uniform environment or provide interfaces among all the tools so that a truly user-friendly environment may be produced.

1.3 RELATED EFFORTS

The programming environment is a rather new research area in computer science. It is still in its infancy, and there is no uniformity in terminology. Programming system, development environment, development support system, software engineering environment, and integrated tool system are often used interchangeably.

Research has been done in such areas as generalized environment, user-friendliness, and tool integration in terms of the programming environment [BRAN81, HUNK80, RIDD80]. Many successful programming environments have been constructed in the past. Four of the more popular environments are UNIX[1] [RITC78], INTERLISP [TEIT75], Cornell Program Synthesizer [TEIT81], and LISPEDIT [ALBE81]. These programming environments are briefly described in the following paragraphs.

UNIX is a time-sharing system developed at Bell Laboratories. After it was invented in 1969, versions were implemented on the PDP-11 family of computers, the VAX-11 series, and successfully transported to the Interdata 8/32 [JOHN78]. UNIX was designed to make programming easy for the programmer. It consists of small, separated tools which do not force a programmer to follow a specific programming methodology. Thus, UNIX is suitable for small size projects, because new tools can be created easily by individual programmer using the system. It is not suitable for large projects that require many programmers following a uniform methodology [MITZ81]. (For large projects, the Programmer's Workbench UNIX system, PWB/UNIX, is more appropriate [DOLO78]).

INTERLISP is an interactive programming environment with tools integrated into the high level language LISP. These integrated tools include debugging facilities and a language-specific editor. INTERLISP has the capabilities of

----

[1]UNIX is a trademark of Bell Laboratories.

correcting errors automatically and modifying system functions. One of its specific features is the programmer's assistant. The goal of which is to cooporate with a user in the program development process and to free the user from concentrating on the problem being solved.

The Cornell Program Synthesizer is a syntax-directed programming environment which uses PL/CS as its base language. It is syntax-directed, because both editing and execution of the program are done using the syntactic structure of the language as the basic unit. For example, the language construct:

```
IF condition
  THEN statement
  ELSE statement
```

is considered to be a syntactic unit. The editor can manipulate this whole unit in a derivation tree. For ordinary text editors, this language construct would only be considered as three lines of text. The synthesizer also uses this syntactic structure as the basic unit for execution so that incomplete programs can be executed. The system enforces the concept of top-down development of programs, from abstraction to generation of the detailed code. Among other features, it uses "template" or abstract computational units of a language. These templates enforce syntactically correct development of programs. The system has special display facilities which enable the user to see the global structure of a program on the screen.

LISPEDIT is a highly interactive programming environment that uses LISP as the base language. It is based on the philosophy that a single unified environment will increase a programmer's productivity. The environment consists of a display system, an editor, an interpreter, a compiler, a static analysis subsystem, and a file system. All these tools are integrated into a single environment.

Related works in progress on programming environments include GANDALF [HABE79] at Carnegie-Mellon University, PASES [SHAP80] at Yale University, and COPE [ARCH80] at Cornell University. The Stoneman report of the U. S. Department of Defense gives a comprehensive design specification of APSE [STON80]. APSE is a programming support environment for the Ada[2] language. It is now under development by the Air Force and the Army.

----

[2]Ada is a trademark of the U. S. Department of Defense (Ada Joint Porgram Office).

1.4 PROBLEM FORMULATION

As far as human-factors are concerned, the design of user-friendly human-computer interfaces should start with the user. That is, experimental data should be gathered on system use. These data are very valuable for the design of human-computer interfaces because decisions can be oriented toward specific user traits. After a system has been implemented, experimental studies should be performed and the data obtained from the studies should be used to guide future modifications and designs.

All of the programming environments mentioned above have the same purpose -- making the programming task a simple, easy, and user-friendly activity. In as much as each of these systems serve the purpose, they are successful environments. However, a need exists for experimental studies that can be used to guide future designs of user-friendly systems.

User-friendliness should be included as an element of software quality whenever the software has a human-computer interface. Unfortunately, the design criteria used to build a user-friendly system is not generally based on experimentally tested data but on speculation of user needs, unsubstantiated principles, and implementation ease [LIND81b].

One important aspect is to evaluate a system from the user's point of view, not from the designer's or implementer's. Unfortunately this is not common practice in computer science research in user-friendly system design -- most of the design decisions are not based on experimentally validated data from the user.

The goals of a programming environment are to improve software quality and increase programmer productivity, that is, to improve the software development cycle. However, the state-of-the-art of software measurement and prediction are far from satisfactory. There are still no software metrics to accurately measure such issues as software quality and programmer productivity. Research is needed to identify and examine the basic issues of human-computer interactions as they relate to the programming environment, and to measure the effects of human-computer interfaces and different types of tools upon the productivity of programmers and quality of software. Quantitative data from experiments are needed to make decisions on and to pursue new directions.

1.5 OVERVIEW OF EXPERIMENTS

Actually, many experiments have been performed on specific features of computer software technology. Experiments on programming language features include [GANN77] which has investigated the statically typed and typeless languages, [WATE79] experimented the structure of loops. [LITE76] studied errors in COBOL, while [YOUN74] analyzed errors in ALGOL, BASIC, FORTRAN, and PL/1. Static and/or dynamic analyses have been extensively used to examine different programming languages, some examples are: [KNUT71] on FORTRAN, [ALEX75] on XPL, [CHEV78] and [SALV75] on COBOL, [BING76] on APL, [ZELK76] on PL/1, and [CLAR77] on LISP. How people debug programs was investigated by [GOUL75]. Relationship between problem complexity and program complexity was examined by [WOOD79]. [CHRY78] studied the programming productivity issue. [COME79] experimented with the top-down design methodology.

These are just a sample of experiments conducted during the past decade. These experiments were done on existing systems not designed for experimentation. Therefore, sometimes modification of the system was necessary, and experiments conducted could only investigate a single issue. Moreover, because of the narrow aspects that can be examined on these systems, related experiments could hardly be performed, and so, the relationship between different behavioral issues was difficult to obtain.

1.6 PEEP

A different approach is being taken at Virginia Tech to achieve the goal of a programming environment. This environment is called PEEP (Pascal Environment for Experiments on Programming). The name of this programming environment reflects its unique feature, i.e., it is a language environment to conduct experiments. Currently, experiments on software technology are done on existing systems which have their own applications, specific environment, and emphasis on particular aspects of the software life cycle. PEEP is designed as an environment solely for research on language environment architecture and for conducting experiments.

The language Ada and its associated support environment APSE (Ada Programming Support Environment) [STON80] of the Department of Defense provides another motivation for the development of PEEP. Ada, working closely with an environment, indicates that the current trend of large scale software system is to have software tools communicate with each other. This, together with the fact that Ada recognizes programming as a human activity, provides evidence that the experiments and the development of PEEP can provide valuable insights for future software systems.

This thesis describes the requirement, design, and implementation of PEEP, and describes what the system looks like to a user.

1.7 OUTLINE OF THESIS

Chapter II describes the needs and features of PEEP. Chapter III gives an overview of what PEEP looks like to a user focusing on the external features of the language environment while Chapter IV details the internal structures. Chapter V is devoted to the discussion of the algorithms to implement the design. The final chapter, Chapter VI, explores the possible extensions and improvements for PEEP.

Chapter II
REQUIREMENTS

As PEEP is a language environment for conducting experiments to evaluate the programming task, the needs of the language environment are reflected in the nature of the experiments that are to be conducted on the system. Evaluation of the programming task is based on two general categories of experiments: examination of the human-computer interaction as it relates to program development and investigation of certain high level programming language features. Of specific interest are the level of interactiveness and the user's preconceived idea of the semantics of a programming language. In this chapter, different features of PEEP are presented to reflect the requirements of PEEP for the experiments.

2.1 LEVEL OF INTERACTIVENESS

One of the requirements of PEEP is to provide a flexible human-computer interface for evaluation of the programming task. For example, the learnability of PEEP and the efficiency-of-use of PEEP may be compared to the level of interactiveness being used.

A level of interactiveness is defined based on the unit of communication among coding, translation, and execution services. There are two levels of interactiveness in PEEP.

The first level is the program level, level 0, which is the same as batch mode of operation. The second level is called level 1. It is at the statement level and it uses the programming language statement as a unit of communication between the software tools.

For an example of program development at level 0, consider an interactive system which has an editor that allows a user to prepare a program, a language processor to compile the program, and an executor to execute the program. One first uses the editor to prepare a program, then this program is entered into the language processor to obtain executable code. The unit of communication between the editor and the language processor is at the program level, or level 0 of interactiveness. When the internal representation of the program is executed, the level of interactiveness is also at level 0 because the unit of communication is the entire program's executable code. Currently, most data processing activities use an interactive mode of operation. But from the above example, the argument can be made that most of today's time-sharing systems are used as if they were batched with the terminal replacing the keypunch machine and the card reader.

To more usefully employ the power of the computer in the construction of a program, a higher level of interactiveness is needed. The second level of interactiveness, level 1, uses the statement as a unit of communication. The user may enter a program statement by statement, and the translator compiles each line as it is entered. Further, the executor is able to compute each statement immediately. If the above mentioned interactive system has interactive level 1 capability instead of level 0, the procedures for preparing, compiling, and executing a program look like the following: first, the editor is used to create a program statement. This line is communicated to the compiler for immediate translation. Errors are reported and the user can then invoke the editor to correct the line. At level 1, execution can begin even though a program has not been completely entered. This is true since level 1 of interactiveness provides communication of individual statements to the executor. As seen from this scenario, integration of tools is necessary at level 1.

2.1.1 The Interactive Levels for Coding and Translation in PEEP

At level 0, PEEP enters a program into a source file without communicating with the translator. After the source program has been prepared, the translator compiles the whole program as in the batch system described in page 14.

When PEEP is operating at level 1, the translator compiles each line immediately after it is entered. At this level, syntax errors are checked and reported whenever a statement is entered. As the coding continues, other errors such as multiple declaration, non-declared types, and assignment or operation of wrong types are reported. Currently PEEP assumes that at level 1 of interactiveness for coding and translation one statement is entered for each text line.

2.1.2 The Interactive Levels for Execution Services in PEEP

Execution at level 0 is just like the batch mode operation. The whole program is executed and results will be printed if there is output statement in the program. There are two major debugging facilities in PEEP at level 0: snapshot dumps and trace facilities. These facilities are taken from the ten levels of source debugging described for the Ada Programming Support Environment (APSE) [FAIR80]. The debuggers in APSE provide comprehensive and extensive debugging features both in batch and in interactive style. Level 0 debugging facilities in PEEP are now briefly described.

A snapshot dump is a source level representation of the state of a program. It is a listing of the values of all the variables involved. A programmer can use it before and after a program or certain statements to see the changes. However, it is a programmer's responsibility to interpret the output of the dumps.

A trace facility provides snapshots of changes to selected variables. It permits output of changes in data values after each statement is executed. The advantage of a trace facility over snapshot dumps is its selectivity. Snapshot dumps may produce a lot of irrelevant information and the cause of an error may not be apparent.

At level 1, the user can cause execution of an operation within a statement,[3] an entire statement, or a compound statement. Therefore, execution of partially completed program is possible. Examples of statement execution and operation by operation execution are shown in the next chapter (Chapter III). In the debugging process, a break point assertion in the form of an assert statement can be set at

----

[3]The execution of an operation within a statement is apparently at a higher level, i.e. level 2. But the execution services use the statement as a unit to execute an operation, so the interactive level is still at level 1. (An operation can not be executed without all the information in a statement.)

different program units. An assert statement such as:

```
assert( c > 9 )
```

can be placed before a statement, a compound statement, a procedure, or the entire program. The scope of this assert statement is the program unit in which the assert statement is placed. If a program element violates the assertion, the program will stop where the violation occurs and the programmer can specify different actions to continue execution, modify the program, or alter data.

2.2 BINDING STRATEGIES

High level programming languages can be classified into two major categories: compiled languages and interpreted languages. One difference between them is the binding strategies they employ. Compiled languages use the static binding strategy. This means that the binding of most program names to some particular characteristic (e.g. the relationship between the variables and their declarations) occurs at compilation time. Interpreted languages use the dynamic binding strategy, in which most of the binding occurs at execution time. With languages like FORTRAN, ALGOL, COBOL, PL/I, and Pascal, execution efficiency is the main concern; most of the bindings are performed during translation time. For languages such as APL, SNOBOL, and LISP, flexibility is of prime consideration; bindings are delayed until execution time.

ALGOL, PL/I, and Pascal, also called the block structured languages, employ the static scoping rule. Based on the block or procedure in which an idenfifier is used, a scoping rule determines how an identifier reference is resolved to its declaration. The static scoping rule, or static binding strategy, can be stated as follows [GHEZ82]: if an identifier is declared in a block or a procedure B, it is visible in B, but not in blocks or procedures that enclose B. However, the identifier is visible to all blocks or procedures that are nested within B except when the same name is redeclared in an enclosed block or procedure. In the exceptional case, the local declaration masks the global declaration.

The dynamic scoping rule, or dynamic binding strategy, uses the most recent association to resolve identifier references. Since the binding of variables occurs at execution time, languages that use the dynamic binding strategy usually have no declarations for variables, because the type of a variable is data dependent. EL1 is a language that allows both strategies in one language [WEGB74]. For experimental purposes, PEEP also employs both strategies, although not simultaneously.

2.2.1 PEEP and Binding Strategies

PEEP uses Pascal as the base language; so, its scoping rule is static. But the experimenter can specify either static or dynamic scoping, so that the subject can write two identical programs with different binding strategies. This is used in analysis of program development and programmer performance as it relates to the name referencing environment [LIND81a].

Figure 2.1 gives an example of a Pascal program which shows the differences between the two scoping rules. If the static scoping rule is used then the program operates as follows: When procedure PROC1 is executed, the reference to X in the WRITELN statement is resolved using the static scoping rule to the X declared in the program BINDING. This is true since PROC1 has no locally declared variable with the same name. The value printed for X by this program using the static scoping rule is zero because X was assigned zero before PROC2 was called.

If the dynamic scoping rule is used, the binding of a variable uses the most recent association. In Figure 2.1, procedure PROC1 is called from procedure PROC2. When PROC1 is executed, the most recent association for the declaration of variable X is in PROC2. The value of X printed by PROC1

```
program BINDING (output);
  var X : integer;

  procedure PROC1;
    begin
      WRITELN (X)
    end;

  procedure PROC2;
    var X : integer;
    begin
      X := 1;
      PROC1
    end;

  begin
    X := 0;
    PROC2
  end.
```

Figure 2.1: A Pascal program demonstrating the scoping rules

in this case is one. This is true since the dynamic scoping rule binds the X in the WRITELN statement to the X declared in PROC2.

Chapter III

THE USER INTERFACE TO PEEP

While Chapter II gives an overview of the full capabilities of PEEP for the experiments, this chapter describes what PEEP, at level 1 of interactiveness looks like to a user.

3.1 SCREEN LAYOUT

Figure 3.1 shows the screen format on the terminal when using the language environment. (Figures in this chapter are not drawn to scale.) The first version of PEEP has been implemented on a VAX-11/780 computer under the VMS operating system. PEEP is terminal dependent, working on a VT100 terminal with Advanced Video [VT1079]. PEEP changes the screen from the normal 80 characters per line to 132 characters per line, and divides the screen into three regions by drawing two vertical lines. These regions have special meanings in developing programs and are now discussed.

All commands are entered when the cursor is in column 1, which is the command region. The commands are all immediate and are not echoed on the screen; that is, when a legal command is typed in column 1, the action is taken immediately and no carriage return is needed. The commands for PEEP at level 1 are all one-character commands that provide program entry and execution facilities.

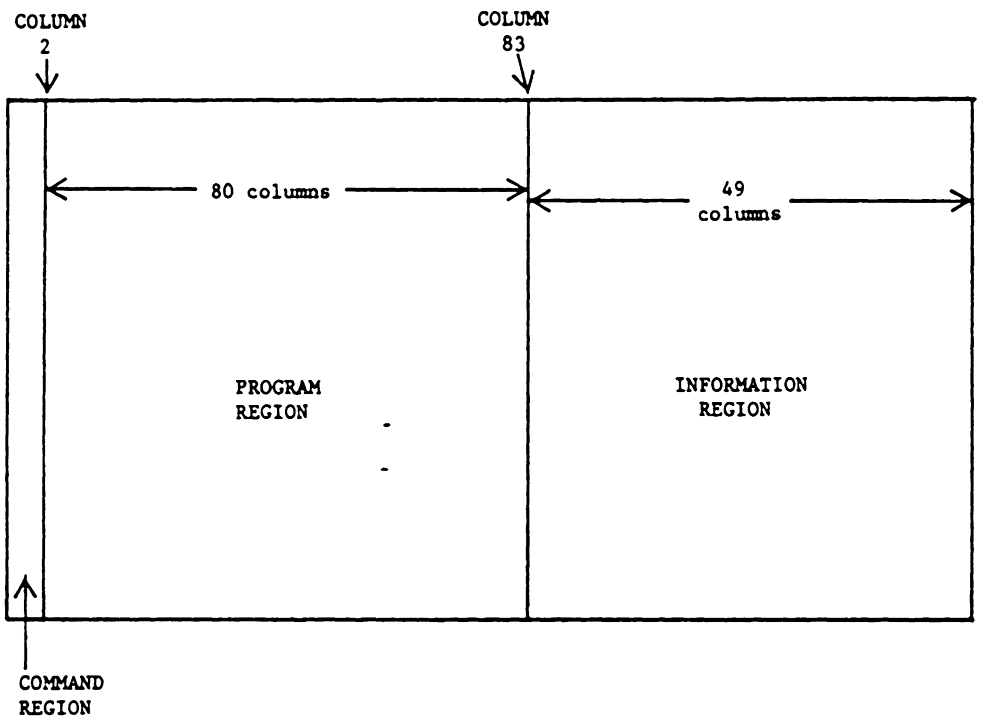


Figure 3.1: Screen format

Columns 3 to 82 constitute the program region in which the user enters the text of a program. After a line of program is typed into this region, the cursor goes back to the command region (first column).

The third region has 49 columns that make up the right side of the screen. This is the information region, and it is used for displaying information and messages from the language environment to the user. For example, syntax errors appear in this region.

3.2 CAPABILITIES OF PEEP

The translator of PEEP recognizes a subset of Pascal consisting of all the features of a full Pascal language except the GOTO statement, input and output statements, and the declarations and usages of records and sets. The following commands are recognized at interactive level 1:

1. D -- moves the cursor down one line, the cursor will not move if it is at the last line of a program.

2. E -- allows the entry of a new line of program text, the cursor will go from the command region to the first column of the program region.

3. O -- executes a single operation within an executable statement, an error message will be shown in the information region if "O" is entered for an unexecutable statement. An executable statement is defined to be a computional statement such as IF statement, iteration statement, assignment statement, or BEGIN statement.

4. S -- executes a statement, an error message will be shown if "S" is entered for an unexecutable statement. If "S" is requested for a compound statement then the entire statement will be executed.

5. U -- moves the cursor up one line, the cursor will not move if it is at the top line of a program.

3.3 PROGRAM DEVELOPMENT EXAMPLE

The following is an illustration of the use of PEEP at level 1. When PEEP is initiated, the cursor moves to the upper left hand corner of the screen. The information region displays a message indicating that the system is expecting a new Pascal program to be entered (Figure 3.2). The user can give the command "E" for entering a line of Pascal program. If the command "E" is typed in column 1,

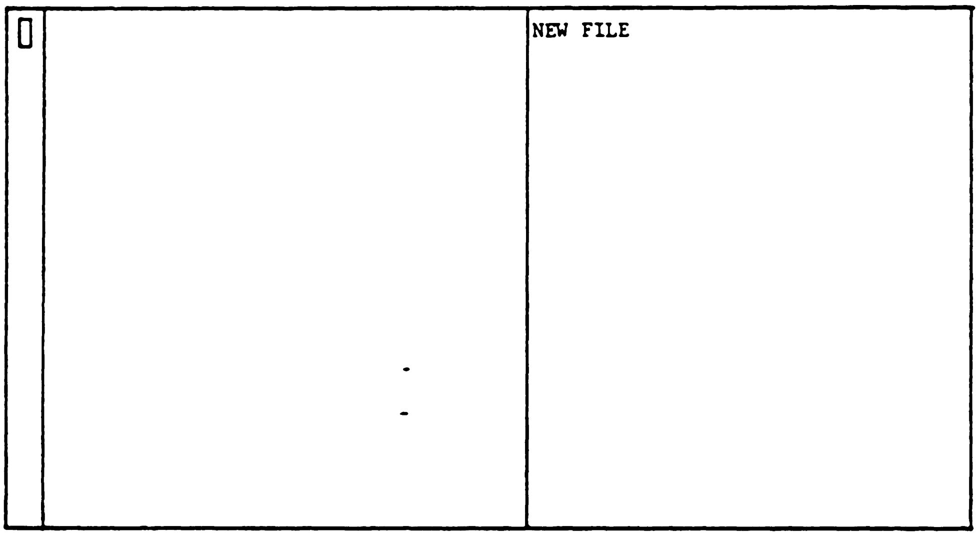
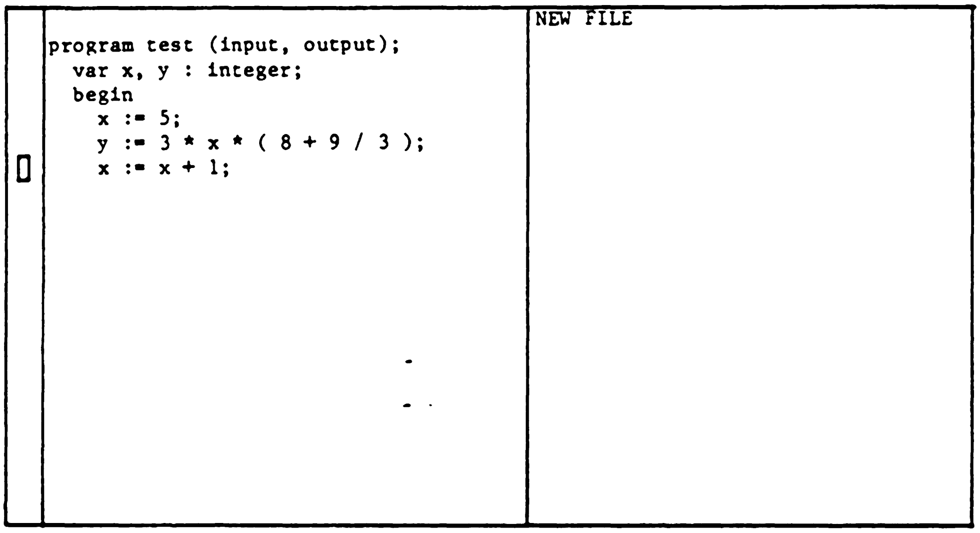


Figure 3.2: PEEP expecting a new Pascal program

the cursor moves to the first column of the program region and the user can enter a line of Pascal. Upon entering a carriage return, any syntax errors that exist in the line of code will appear in the information region and the cursor returns to the command region. Every time the user wants to enter a line of text, the "E" command must be used. Figure 3.3 shows that the user has entered six lines of code.

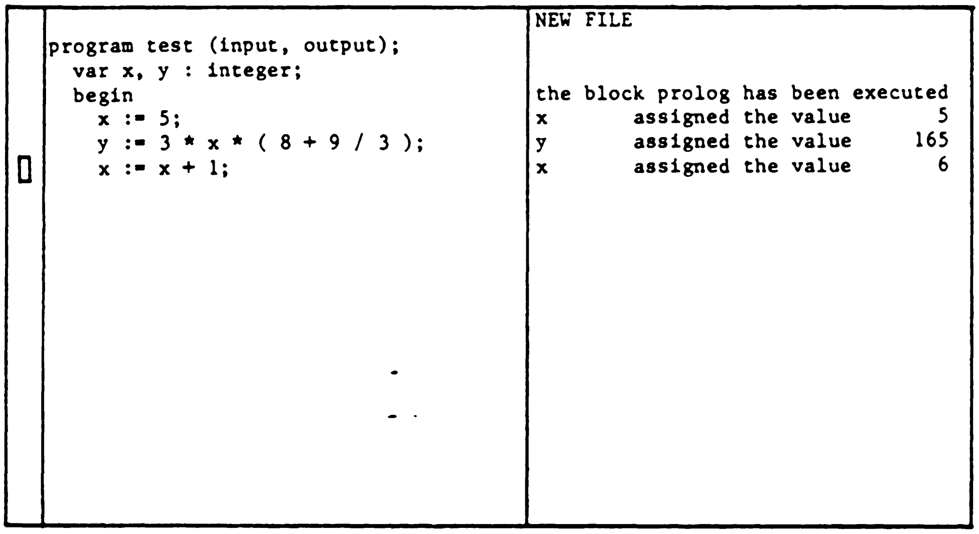
The user can start executing the program even though it has not been completely entered. This is done by moving the cursor next to a particular statement by using the "D" (down) or "U" (up) commands. Initially, the cursor should be moved to the keyword BEGIN (the beginning of a block), and the command "O" for operation should be entered. The message, "the block prolog has been executed" will appear in the information region (Figure 3.4) indicating that the interpreter is ready to execute statements of the block. Now, the user can execute the statements by moving the cursor to each statement and entering an "S" command for executing a single statement. The resulting effects are shown in Figure 3.4.



NEW FILE

```
program test (input, output);
  var x, y : integer;
  begin
    x := 5;
    y := 3 * x * ( 8 + 9 / 3 );
    x := x + 1;
```

Figure 3.3: A partially completed program in PEEP



```
program test (input, output);
  var x, y : integer;
  begin
    x := 5;
    y := 3 * x * ( 8 + 9 / 3 );
    x := x + 1;
```

```
NEW FILE


the block prolog has been executed
x       assigned the value       5
y       assigned the value     165
x       assigned the value       6
```

Figure 3.4: Three executed statements in PEEP

If the "S" command is entered repeatedly on the statement:

```
x := x + 1
```

the value of x will increment by 1 for each entry. Now, if the cursor moves up to the statement:

```
y := 3 * x * ( 8 + 9 / 3 )
```

and "S" is entered, the value of y will be changed because the value of x has been changed in the statement:

```
x := x + 1
```

If the user wants the correct values as if statements are executed sequentially for the first time, the BEGIN statement should be executed again by entering the command "O" as described above.

The user can execute a statement operation by operation. For example, the cursor can be moved to the statement:

```
y := 3 * x * ( 8 + 9 / 3 )
```

and the user can enter the "O" command. Following is the series of messages shown in the information region each time an "O" command is entered. Each message will erase the previous one.

```
multiply yields              15
divide yields                 3
plus yields                  11
multiply yields             165
y assigned the value        165
```

Chapter IV

DESIGN

4.1 OVERALL DESIGN

PEEP consists of five main modules: a command dispatcher, an editor, a translator, an interpreter, and a state examiner and modifier (Figure 4.1). Two data structures in PEEP store three different forms of the source program. The first data structure is the source file which contains the textual representation of a program. The second is a common storage for program representations that constitutes a snapshot of a program during execution. The snapshot consists of a general list representing the static (compile-time) structure of a program, and consists of a general list depicting the dynamic (run-time) structure of the program. These compile-time and run-time storage structures are the program and the record skeletons respectively. The structures are based on the semantic models of computation described in [JOHN73]. In these models, a program's structure, instructions, and identifiers are kept in the program skeleton. The record skeleton, similar to the functions of an activation stack, keeps the current state of execution. The semantic models of computation also give flexible implementations of different kinds of binding strategies.

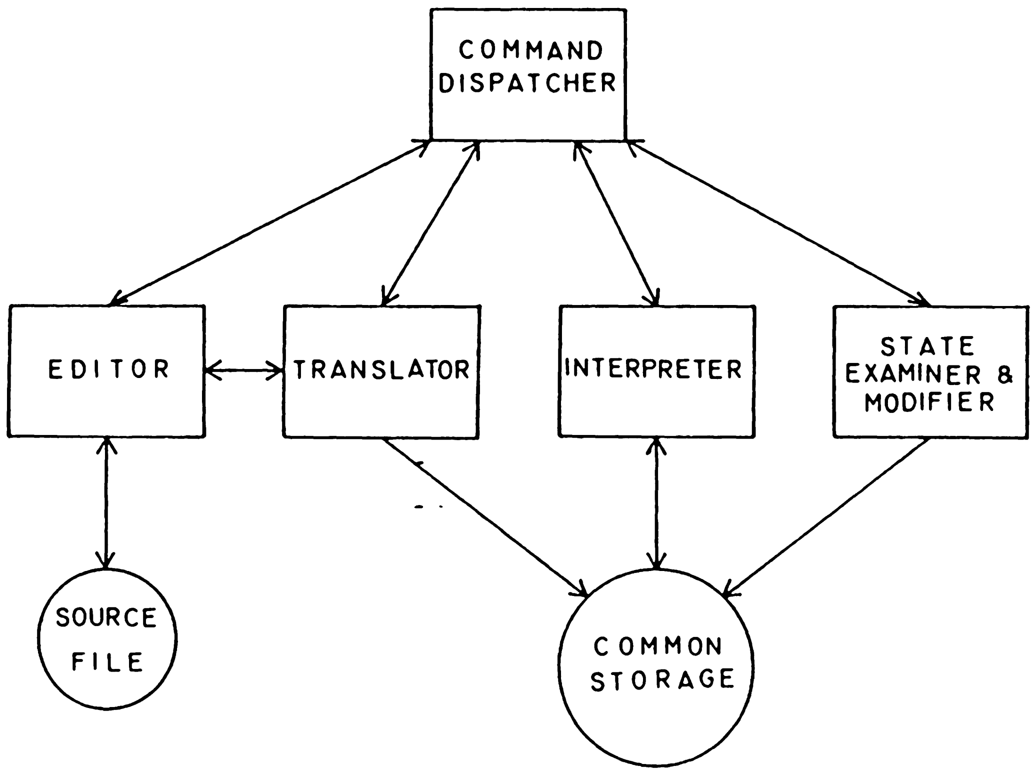


Figure 4.1: Overall design of PEEP

The command dispatcher is responsible for the invocation of the other four modules. When a module finishes its functions, it always returns back to the command dispatcher. The editor can create and modify a source program. The translator is for the creation of the program skeleton. It can be explicitly invoked by the command dispatcher, and can also be implicitly invoked by the editor every time a line of text is entered into the source file. In this way, the lexical, syntactic, and semantic functions of the translator can be carried out on each line of the program as soon as the line is entered. The record skeleton is built by the interpreter which carries out the execution and debugging functions of PEEP. The last module, the state examiner and modifier, uses the record skeleton, displays information regarding the state of execution, and changes the information in the record skeleton for testing and debugging purposes.

The following sections describe the representations of the three different forms of the source program in detail. Following the description of these storage structures, different binding strategies and their realization by the contours are presented.

4.2 TEXT STORAGE

The text storage is actually a disk file of the source program. Associated with the disk file is a storage system which has been developed at Virginia Tech for the experimental text editor SAM [EHRI81], and adopted for use in PEEP. The storage system is a virtual storage system, which is briefly discussed in the following.

The virtual storage system consists of three data structures -- a queue, the page tables, and the working storage (Figure 4.2). The queue is used for editing purposes which is not essential to the description here. The page table is a physically sequential list where each entry of the table contains the numbers of the pages in working storage. (In SAM, there are two page tables, one for the working storage of the primary file, and the other for the working storage of the secondary file. For simplicity and clarity, these are not discussed here). There are two separate parts in the page table, one contains the currently allocated pages, and the other contains free pages. The working storage consists of a number of pages of the source file. Each page consists a number of fragments which is a doubly linked list structure. A page has information about the length of each line, the number of lines in the page, and the number of free fragments.

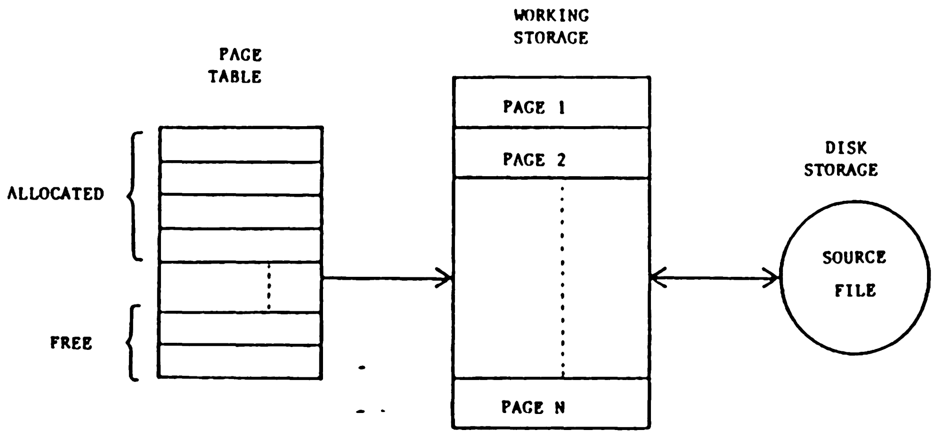


Figure 4.2: SAM's virtual storage system

When a line of text is entered into PEEP, it is put into a vector called the input buffer. Then, it is inserted into the fragment of a page in the working storage. A line occupies one or more fragments depending upon the line length. The content of the working storage is stored on a disk whenever a file is permanently stored. When a file is needed for editing, it is moved from the disk to the working storage. The page table is responsible for all the retrievals and insertions of the currently working page. When a line is edited, the line should be transfered from the fragment of a page in working storage to a vector. After being edited, the content of the vector is stored back into the fragment. Any changes in size of a line in the fragments can be very easily adjusted via the doubly linked list structure of the fragments.

4.3 PROGRAM SKELETON

Figure 4.3 shows a pseudo Pascal program with nested procedures. A pseudo program is used, so that the overall structure can be seen without the details that might cloud the whole picture. In the diagram, let, Dn, $n \geq 1$, represent certain declarations; let Sn symbolize certain instructions; and let Pn be the procedure names. If the procedure name appears in the instructions of a procedure, it means the

```
PROGRAM P1;
  D9; D10;

  PROCEDURE P2;
    D8;

    PROCEDURE P4;
      D4; D5; D6;
      BEGIN
        S4
      END;

    PROCEDURE P5;
      D1; D2; D3;
      BEGIN
        S5
      END;

    BEGIN
      S2; P4; P5
    END;

  PROCEDURE P3;
    D7;
    BEGIN
      S3; P2
    END;

  BEGIN
    S1; P3
  END.
```

Figure 4.3: A pseudo Pascal program

call statement to a particular procedure. The nesting nature of this Pascal program (Figure 4.3) can actually be represented by the block structure as shown in Figure 4.4. Figure 4.4 shows that this nesting structure is hierarchical, so it can also be represented by a general tree (Figure 4.5).

4.3.1 Program Contour

Each node of the general tree is a compound cell called the program contour. There are three subcells in the contour: the environment link, the declaration link, and the antecedent link. It is the environment link of the program contour that realizes the tree structuring of a program. The declaration link is a pointer which points to a circular list of declarations, while the antecedent link points to a general list called the code list representing the instructions in a program or in a procedure. There is a particular program contour called the root which has null environment and null antecedent links. Its declaration list consists of the four standard declarations: integer, real, boolean, and character. The program contour, with its environment link pointing to the root, represents the main program. All the other program contours, except the root contour, represent procedures in a program. The program contours together with

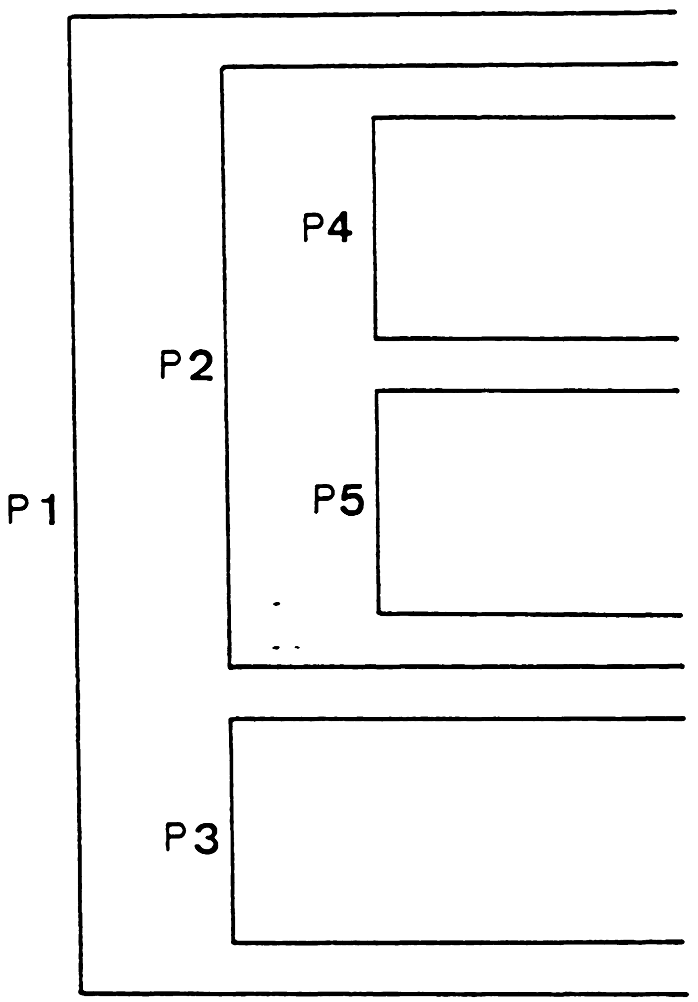


Figure 4.4: Nesting representation of the Pascal program

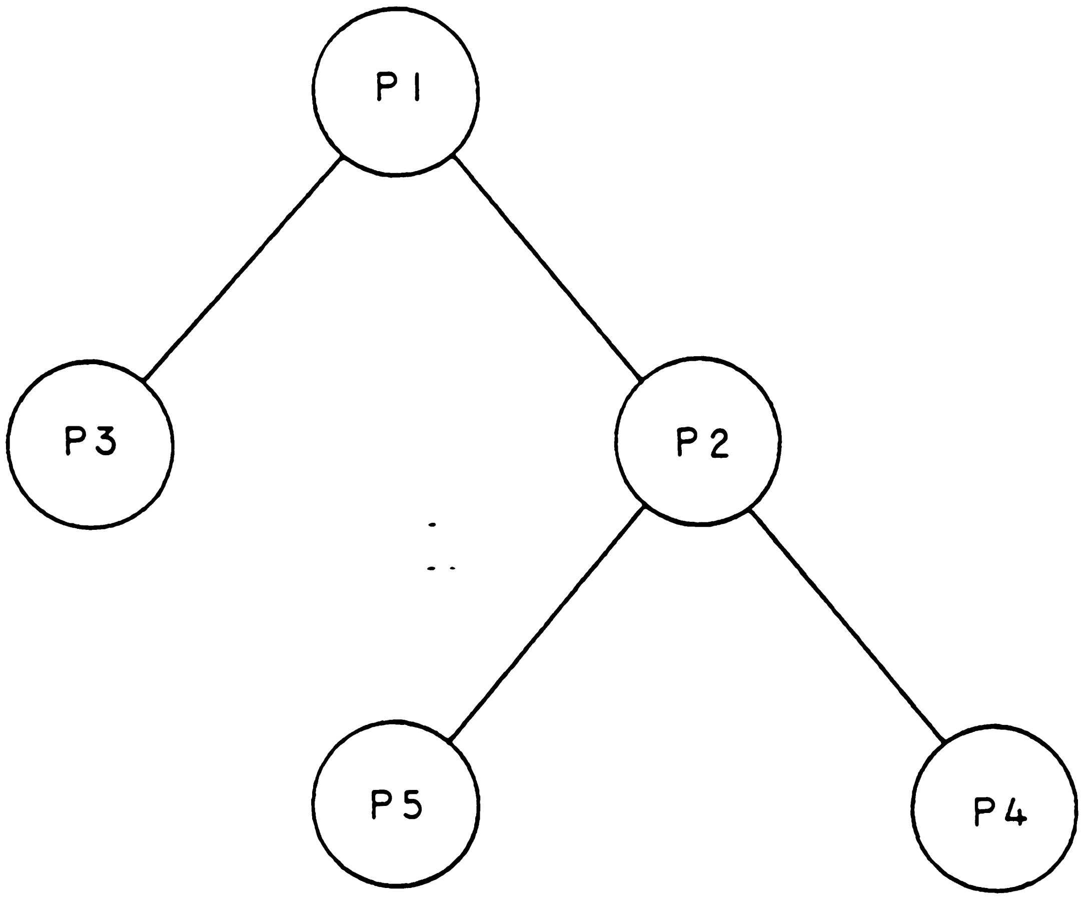


Figure 4.5: Tree representation of the Pascal program

their associated declaration list and code list, when linked together by the environment links, constitute the program skeleton. The different subcells of a program contour are shown in Figure 4.6. As can be seen from the figure, the program contour has an identification subcell named PROGRAM.

4.3.2 Declaration List

The declaration link of a program contour points to a circular list of declaration nodes. Each declaration node contains the declaration of an identifier in a particular program or procedure with the exception of the declaration nodes of the root contour. The declaration list of the root contour always contains declaration nodes of the standard types in Pascal, namely, the integer type, the real type, the boolean type, and the character type. The occurrence of an identifier in a declaration node is said to be a declaration occurrence of that identifier. No two distinct declaration nodes of a program contour can have the same identifier. A declaration node contains three major fields: an identification field, a link field, and an information field. The identification field specifies what kind of declaration that the declaration node indicates. The link field contains a pointer to the next declaration node; since the declaration list is circular, the link field of the last

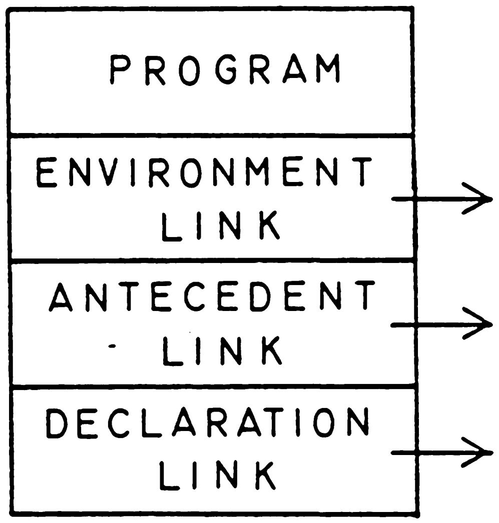
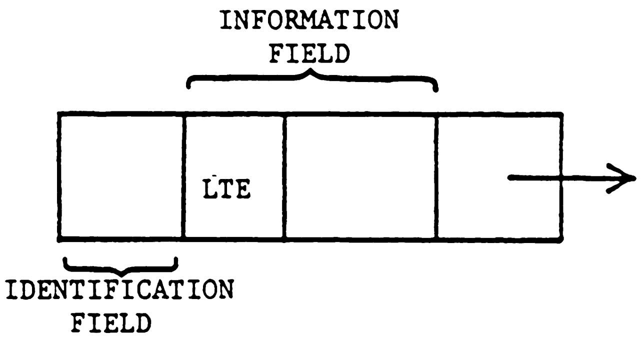


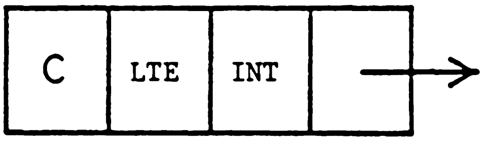
Figure 4.6: A program contour

declaration node points to its program contour. Whether the information field has two or more subfields depends on the different sorts of declarations. The subfields of an information field have various data, one of them is lexical table entry. A lexical table is simply a one-dimensional array (with negative indexes), each element of the array is called a lexical table entry, which contains all the identifiers except keywords in a Pascal program.

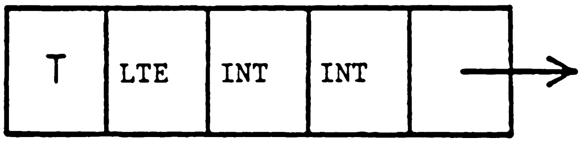
Figure 4.7 (a) shows the general format for a declaration node, and 4.7 (b), (c), (d), (e), and (f) show the different fields of the declarations of constant (CONST), type (TYPE), variable (VAR), procedure, and function respectively. For CONST declaration node, the information field consists of one lexical table entry (LTE) and one integer subfield (INT). The TYPE declaration node has a lexical table entry and two integer subfields in the information field. The information field of VAR, procedure, and function declaration nodes have similar formats. They all contain lexical table entries and a subfield for a pointer. The pointer subfield of VAR points to a declaration node which contains the type of the variable, while the pointer subfield of procedure or function points to a program contour which represents their corresponding procedure or function. Some declarations involve only one declaration node, others may



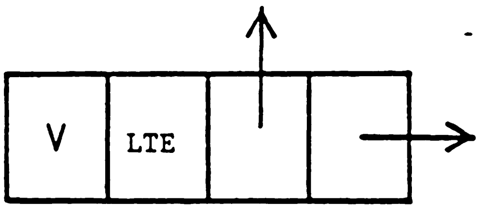
(a)



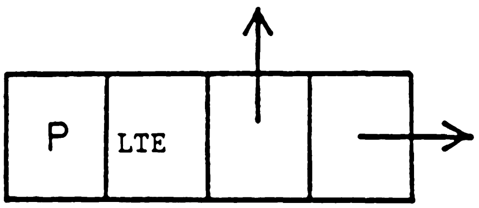
(b)



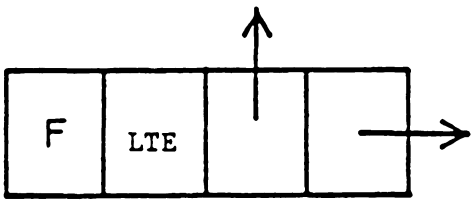
(c)



(d)



(e)



(f)

Figure 4.7: Declaration node formats

involve more than one declaration node. Figure 4.8 gives different examples of declarations and their node representations.

4.3.3 Code List

The code list is a general linked list structure which represents the instruction codes of a program or procedure. An occurrence of an identifier in a code list is said to be a reference occurrence of the identifier. There are two general forms of a code list, one for the main program, and the other for the procedure or function. The code list which belongs to a main program has the keyword PROGRAM which indicates that the list is for the main program. The list also has a program name, a file name sublist, and a statement sublist. It has the following general structure:

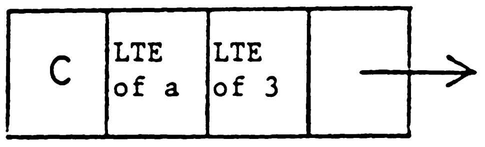
```
(PROGRAM, program_name, (file_names), (statements))
```

Procedures and functions have different keywords, and the file name sublist is replaced by a formal parameter sublist:

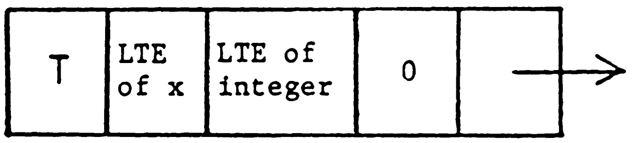
```
(PROCEDURE, procedure_name, (formal_parameters),
              (statements))

(FUNCTION, function_name, (formal_parameters),
      return_variable, (statements))
```

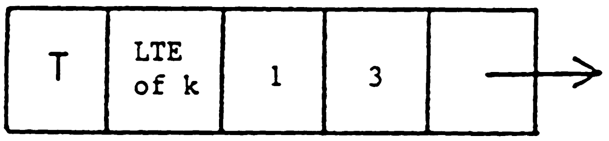
CONST a = 3;



TYPE x = integer;



TYPE k = 1..3;



TYPE w = (x, y);

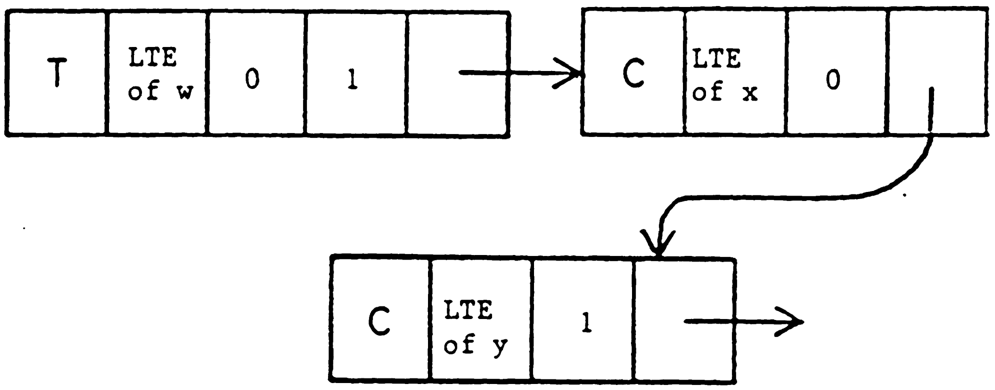
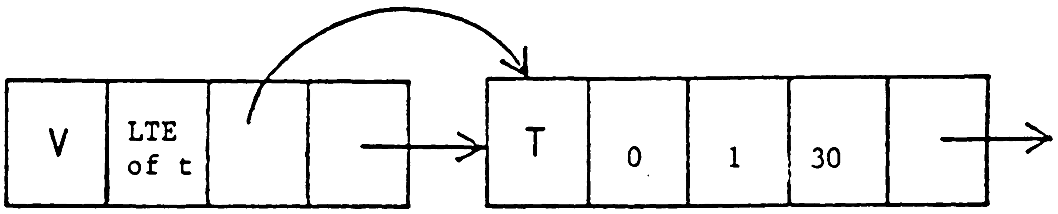


Figure 4.8: Examples of declaration nodes

VAR t : 1..30;



VAR s : (a, c);

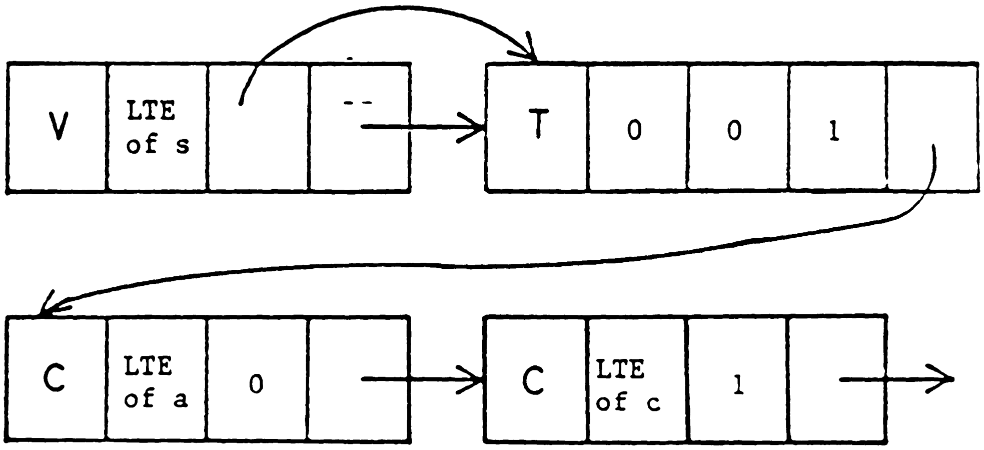


Figure 4.8: cont'd

The different kinds of statements in the statements sublist are given below:

EXPRESSION:

```
        (operator, left_operand, right_operand)
```

example:

```
                    x + y
                  (+, x, y)
```

ASSERT STATEMENT:

```
              (ASSERT, (expression))
```

example:

```
               ASSERT(x > 3 - y)
            (ASSERT, (>, x, (-, 3, y)))
```

ASSIGNMENT STATEMENT:

```
            (:=, variable, (expression))
```

examples:

```
                    u := 8
                  (:=, u, 8)

                a := p + m * c
            (:=, a, (+, p, (*, m, c)))
```

CASE STATEMENT:

```
 (CASE, (expression), (constant, (statement)) , ... )
```

example:

```
                 CASE b OF
                   3 : x := y;
                   9 : x := y + 1
                 END

 (CASE, b, (3, (:=, x, y)), (9, (:=, x, (+, y, 1))))
```

IF STATEMENT:

```
(IF, (expression), (then_statement), (else_statement))
```

example:

```
            IF c > 5 THEN a := w * r
                     ELSE a := w

   (IF, (>, c, 5), (:=, a, (*, w, r)), (:=, a, w))
```

REPEAT STATEMENT:

```
          (REPEAT, (statement), (expression))
```

example:

```
             REPEAT e := e + 1 UNTIL e > 99

        (REPEAT, (:=, e, (+, e, 1)), (>, e, 99))
```

WHILE STATEMENT:

```
            (WHILE, (expression), (statement))
```

examples:

```
                  WHILE t DO a := a + 1

             (WHILE, t, (:=, a, (+, a, 1)))


                WHILE t < 4 DO a := a + b

         (WHILE, (<, t, 4), (:=, a, (+, a, b)))
```

FOR STATEMENT:

```
    (FOR, (expression), (expression), (statement))
```

example:

```
            FOR i := 100 DOWNTO 1 DO
                g := h + 2


            (FOR, 100, 1, (:=, g, (+, h, 2)))
```

PROCEDURE AND FUNCTION CALLS:

```
     (CALL, procedure_name, (actual_parameters))
```

example:

```
                    p(a, j, k + 8)

             (CALL, p, (a, j, (+, k, 8)))
```

COMPOUND STATEMENT:

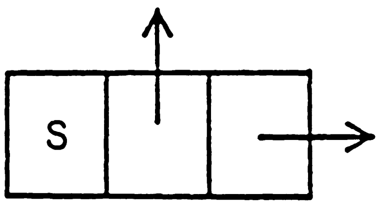
```
(;, (statements), (statements))
```

example:

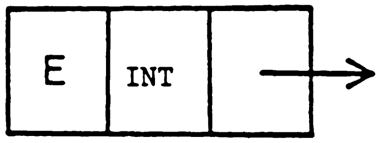
```
BEGIN
  a := x + y;
  b := f;
  c := k
END
```

```
(;, (;, (:=, a, (+, x, y)), (:=, b, f)), (:=, c, k))
```

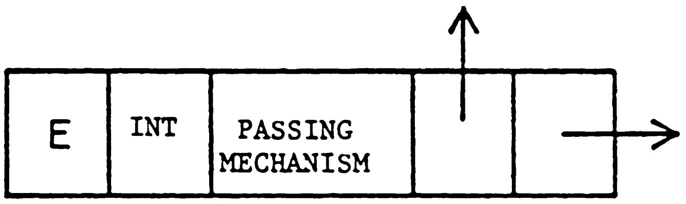
There are two kinds of nodes in a code list: elementary and sublist nodes. All the elementary nodes have a field for an integer and a pointer field to the next node. However, elementary nodes for the parameters of procedures and functions have an extra field to specify the passing mechanism (by reference or by value) and one more pointer field to indicate the parameter type (i.e., the pointer field has a pointer pointing to a declaration node which specifies the type of a parameter). The integer field of an elementary node either contains a lexical table entry, or a keyword or operator number. The lexical table entries are represented by negative numbers, while keywords and operators are represented by positive numbers, so that a virtual processor can distinguish which is which. (The actions of the virtual processor is presented in Section 4.7). Figure 4.9 shows the three different kinds of nodes in a code list.



(a) SUBLIST NODE



(b) ELEMENTARY NODE



(c) ELEMENTARY NODE FOR PARAMETER

Figure 4.9: Code list node formats

4.3.4 Diagram of Program Skeleton

Figure 4.10 shows the program skeleton of the sample Pascal program of Figure 4.3. This figure gives a general structure of the program skeleton; the detail structures can be easily figured out from the descriptions of the program contour, declaration list, and the code list. Let Pn, n $\geqslant$1, represent the program contours; let Sn symbolize the code lists; let Dn be the declaration nodes; and let DPn denote the declarations for the procedures and functions. In the root contour, the declaration nodes of I, R, B, and C correspond to the Pascal types of integer, real, boolean, and character respectively. In Figure 4.10, the following declarations are assumed: D1 is declared to be a constant; D2, D4, and D9 are types; D5 and D6 are variables of type D4; D3 is a variable of type D2; D7 is a variable of type D9; D8 is a variable of type CHAR; and D10 is a variable of type REAL. (For clarity, the pointers from a VAR declaration nodes are using dash arrows).

4.4 RECORD SKELETON

While the program skeleton of a Pascal program is fixed during execution, the record skeleton depends on the program execution. Record skeleton consists of record contours, which are created whenever a program, a procedure, or a

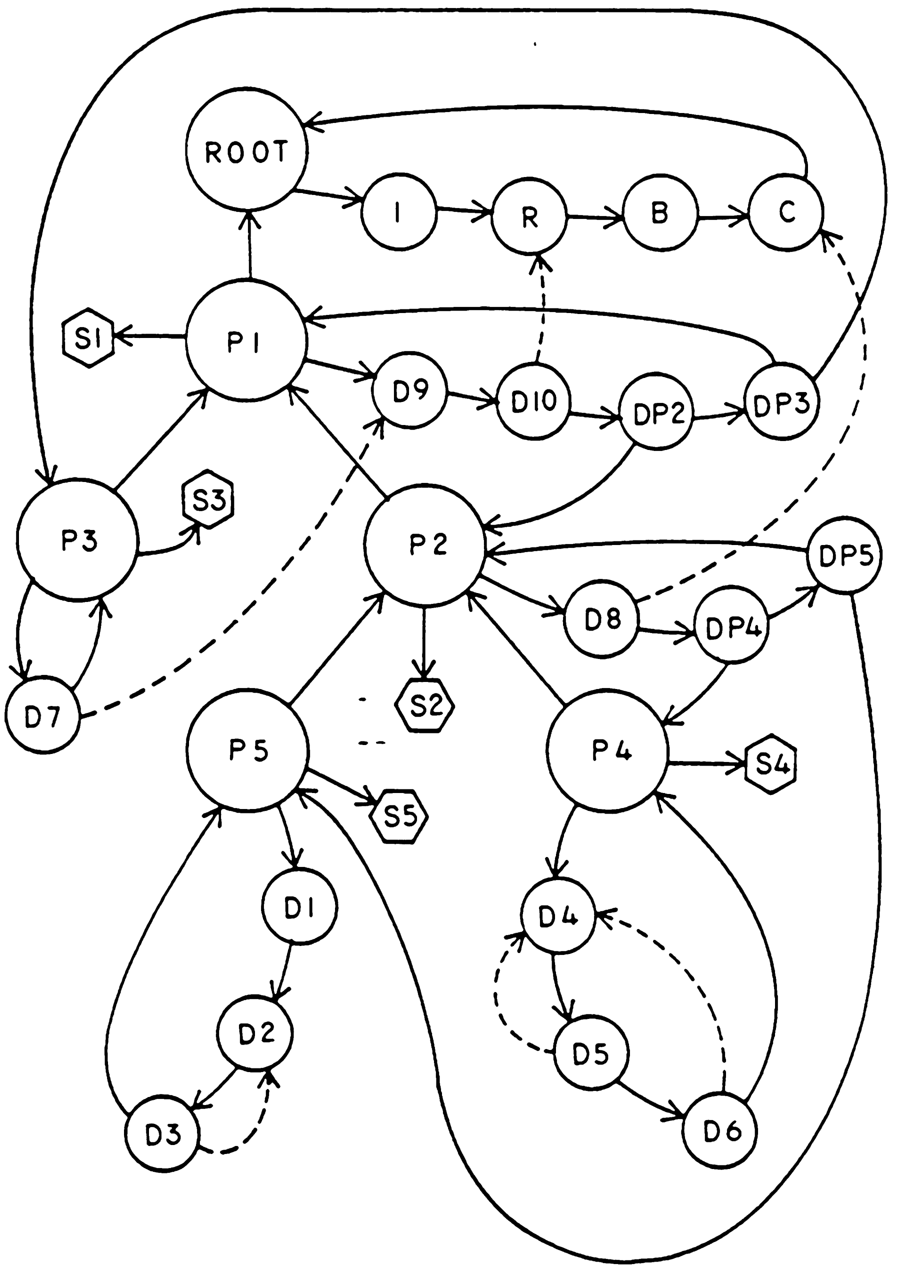


Figure 4.10: Program skeleton of Figure 4.3

function is invoked either recursively or non-recursively. The record contours work like an activation stack.

4.4.1 Record Contour

Isomorphic to a program contour, a record contour has three subcells: the environment link, the association link, and the antecedent link. The environment link which points to another record contour, is determined by the binding strategy employed. (The binding strategies will be discussed in Section 4.6). As described in [JOHN73], each activation record, A, is a record contour whose antecedent link, points to some program contour, B; B is said to be the antecedent of A while A is said to be a descendent of B. The association link is a pointer to a circular list of association nodes. The record contours, which consist of the association lists and antecedent links, when linked together by the environment links, constitute the record skeleton. Figure 4.11 shows the subcells of a record contour. Similar to the structure of a program contour, the record contour has an identification subcell known as RECORD.

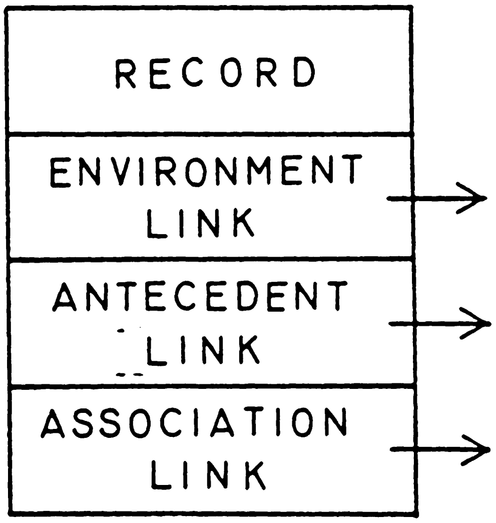


Figure 4.11: A record contour

4.4.2 Association List

The association list, which is a circular list of association nodes, is pointed to by the association link subcell of a record contour. Each association node corresponds to a declaration occurrence, except there is no association node for the type of an identifier. An association node contains the value of an identifier; the value is encountered in the reference occurrence and put into the value field of the association node. Similar to the declaration node, an association node contains four fields: a lexical table entry field, a type field, a value field, and a link field. The lexical table entry field contains the lexical table entry of the identifier. The type of the identifier is stored in the type field, while the value of the identifier, which depends on its type, is in the value field. The link field has the same function as the link field of a declaration node. Thus, the link field points to the next association node or points to a record contour. Figure 4.12 (a) shows the general format of an association node, and 4.12 (b), (c), (d), and (e) show the specific examples.

4.4.3 Diagram of Record Skeleton

Figure 4.13 shows the structure of record skeleton and its relationship with the program skeleton. Again, let Rn,

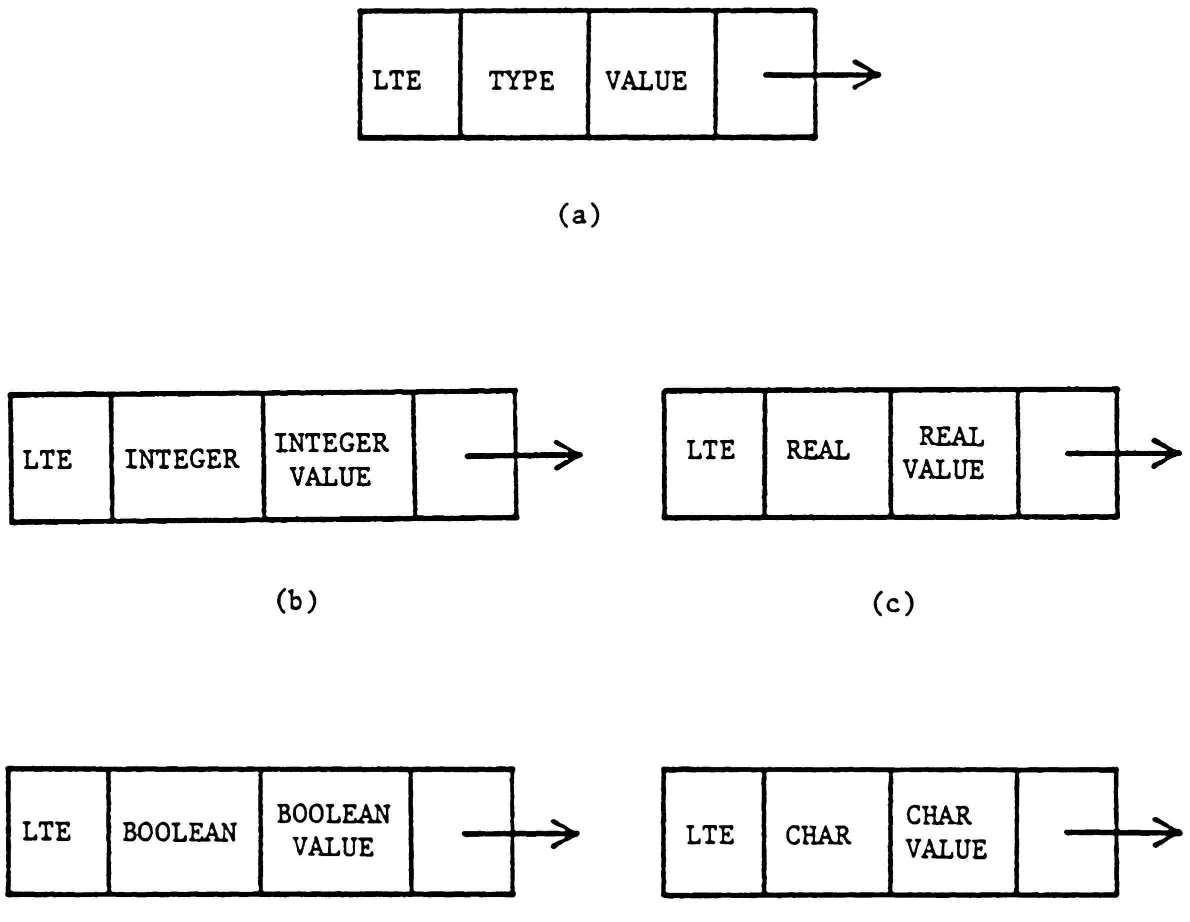


Figure 4.12: Association node formats

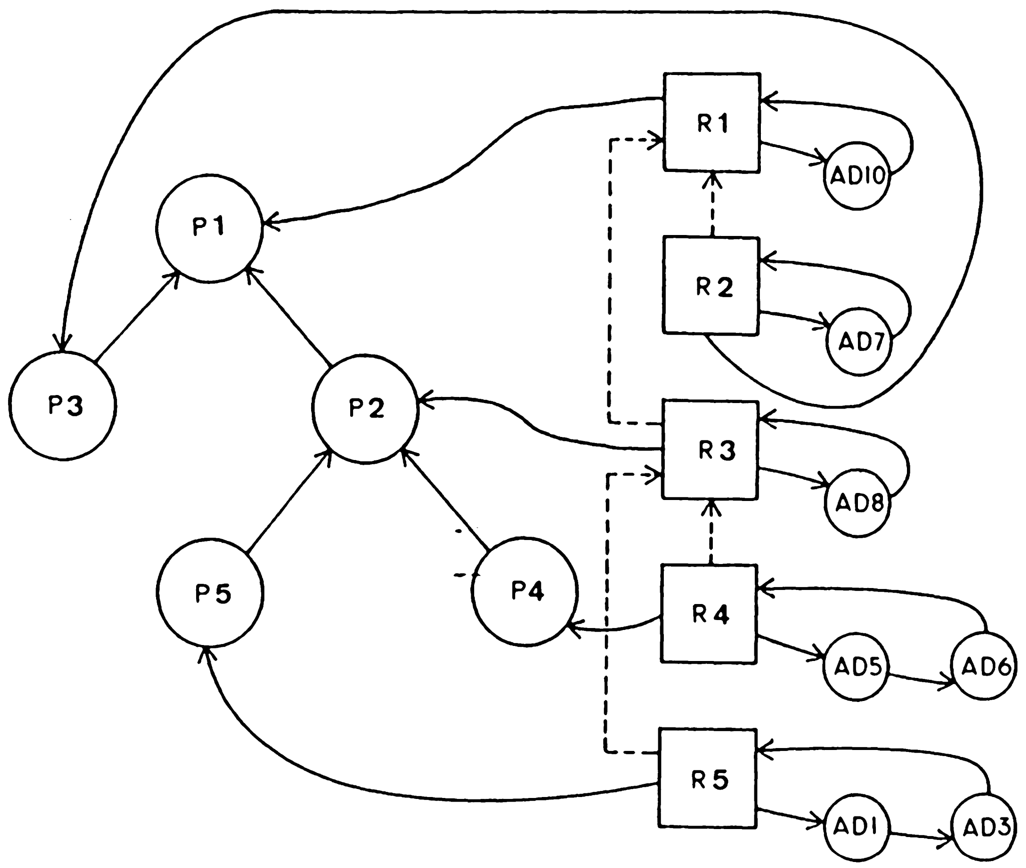


Figure 4.13: Record and its associated program skeleton

$n \geqslant 1$, represent the record contours; and let ADn denote the association node corresponding to the declaration node Dn in the program skeleton (refer to Figure 4.10). In Figure 4.13, the environment links of the record contours are shown with dash arrows, which are determined by the binding strategy employed (discuss in Section 4.6).

The construction of the record skeleton is briefly described here; further detail is discussed in Section 4.7. When the main program P1 is executed, R1 is created with an antecedent link pointing to the program contour P1. P3 is called from P1, so another record contour (R2) is created with its antecedent link pointing to the program contour P3. This process goes on for record contours R3, R4, and R5.

4.5 VIRTUAL PROCESSOR

There is a virtual processor in the semantic models of computation. The virtual processor is isomorphic to the register structure of a hardware processing unit; its function is to carry out the computation of the semantic models. One of the important concepts of a virtual processor is the label register which consists of an ordered pair:

<ip, ep>.

The ip is an instruction pointer which must point to an instruction in some code list, while the ep is an environment pointer, which is either null or points to a record contour. In a given snapshot, the ip of the virtual processor points to the next instruction to be executed; the ep of that processor determines the immediate access environment for the processor. The actions of the virtual processor are described in Section 4.7.

4.6 ENVIRONMENTS

In [JOHN73], an environment is described to be either the null sequence of contours, or consists a sequence of contours $<C_0, C_1, \ldots , C_n>$, such that $n \geqslant 0$, the environment link of C_n is null, and for $0 \leqslant i < n$, the environment link of C_i points to C_{i+1} . The first member (C_0) of a non-null environment is called the top member of that environment, and the last (C_n) is called the bottom member which is actually the root of a tree.

4.6.1 Environment Binding Strategies (EBS)

Two mechanisms, which producing pointers to record contours, are essential in defining various identifier binding and environment binding strategies, are discussed below:

Let E be the record environment, such that

$E = \langle E_0, E_1, \ldots, E_n \rangle$, where $n \geqslant 0$:

1. <u>The Dynamic Environment Binding Strategy (DYN)</u>:

The inputs to DYN are a pointer to record contour E_i and record environment E. The output pointer is null if and only if E is null. If E is non-null, the output points to the top record contour of E.

Symbolically:

$$DYN(\uparrow E_i, \langle E_0, E_1, \ldots, E_n \rangle) \dashrightarrow \uparrow E_0$$

where $0 \leqslant i \leqslant n$

(In the thesis, the symbol "$\uparrow$" means "a pointer to").

2. <u>The Static Environment Binding Strategy (STAT)</u>:

The inputs to STAT are a pointer to record contour E_i and record environment E. the output pointer is a copy of E_i.

Symbolically:

$$\text{STAT}(\uparrow E_i, \langle E_0, E_1, \ldots, E_n\rangle) \text{-->} \uparrow E_i$$

where $0 \leq i \leq n$

4.6.2 Identifier Binding Strategy

As discussed in [JOHN73], the purpose of identifier binding, and of the search mechanism which realizes it, is to provide for each environment and each identifier an association between a reference occurrence of that identifier and some declaration occurrence of that identifier in some contour of the environment. A binding may be regraded as a set of pairs of the form:

<identifier, pointer>.

For every identifier in a program, the binding contains exactly one such pair. The pointer in such a pair either is a null pointer or points to a contour in the environment whose declaration list or association list has an occurrence of the identifier. If a pointer is null, that identifier is free; otherwise, the identifier is bound to a contour pointed to by the pointer. In realizing this pair, a search mechanism is needed. This is introduced below:

Absolute Highest Search Mechanism (AH):

Let C be an environment and let I be an identifier. The operational steps taken in locating a contour in C associated with I are as follows [JOHN73]:

1. If C is null, return a null pointer.
   AH(I, <null>) --> null

2. If C is non-null, conduct an iterative search to determine the minimal index j such that $0 \leq j \leq n$ and C_j has a declaration occurrence of I; if the search fails, return a null pointer; otherwise, return a pointer to the located contour of C_j.
   AH(I, $<C_0, C_1, \ldots, C_n>$) --> null or $\uparrow C_j$

4.6.3 Complete Binding Strategies (CBS)

Two complete binding strategies are used in PEEP. A complete binding strategy, which consists of two search mechanisms and one environment binding strategy, determines the binding method used in a language. Thus, CBS is a 3-tuple, consists of:

(PS, RS, EBS)

PS is the program contour search mechanism and RS is the record contour search mechanism. The two CBS used in PEEP are discussed in the following:

1. Dynamic Complete Binding Strategy (DYN CBS)

   The program and record contour search mechanisms are the absolute highest method, and the EBS is the dynamic environment binding strategy.

   DYN_CBS = (AH, AH, DYN)

2. Static Complete Binding Strategy (STAT CBS)

   The program and record contour search mechanisms are the same as DYN_CBS, but the EBS is the static environment binding strategy.

   STAT_CBS = (AH, AH, STAT)

4.7 EXAMPLES

The complete binding strategy determines the environment links of the program and record contours. Figure 2.1 in Chapter II gives an example of a Pascal program which yields different results with different binding strategies. Let us use that example to demonstrate how the actions of the virtual processor produce the record skeleton, and to show the realization of binding strategies using the environment links of the record skeleton.

First, let us see how the environment links of program contours are determined. Figure 4.14 shows the program skeleton of the Pascal program. The program contour search mechanism determines the environment links of the program contours. Since both the DYN_CBS and STAT_CBS use AH for the program contour search mechanism, the program skeleton is the same for both strategies. BINDING is the program contour for the main program. It has declaration nodes for the variable X and for the two procedure declarations. PROC1 has its declaration in BINDING, when the AH search mechanism applies to the identifier PROC1, it yields a pointer to the contour BINDING:

$$\mathrm{AH}(\mathrm{PROC1},\ \langle\mathrm{BINDING}\rangle) \dashrightarrow \uparrow \mathrm{BINDING}$$

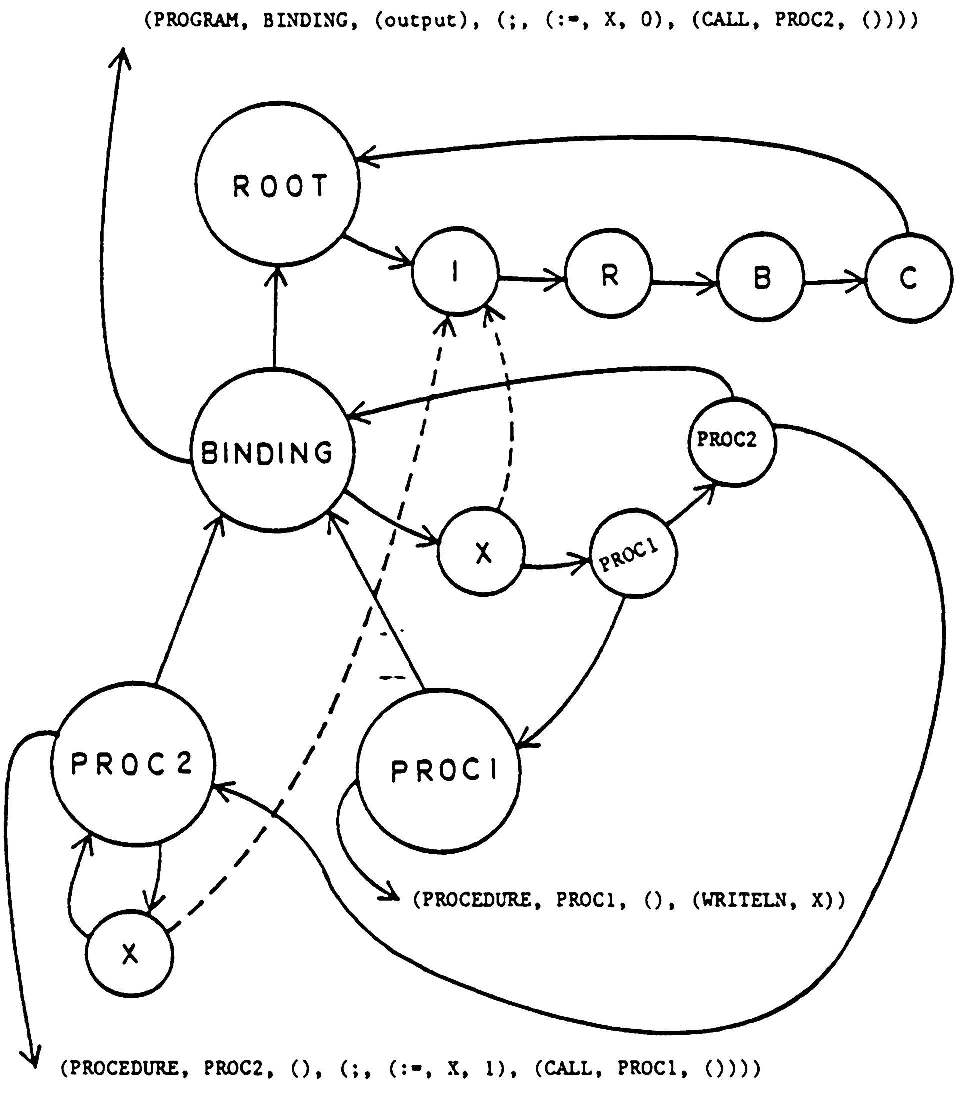

(PROGRAM, BINDING, (output), (;, (:=, X, 0), (CALL, PROC2, ())))
ROOT
I
R
B
C
BINDING
PROC2
X
PROC1
PROC2
PROC1
X
(PROCEDURE, PROC1, (), (WRITELN, X))
(PROCEDURE, PROC2, (), (;, (:=, X, 1), (CALL, PROC1, ())))

Figure 4.14: Program skeleton of Figure 2.1

Therefore, the environment link of PROC1 points to BINDING. Similar situation happens to PROC2:

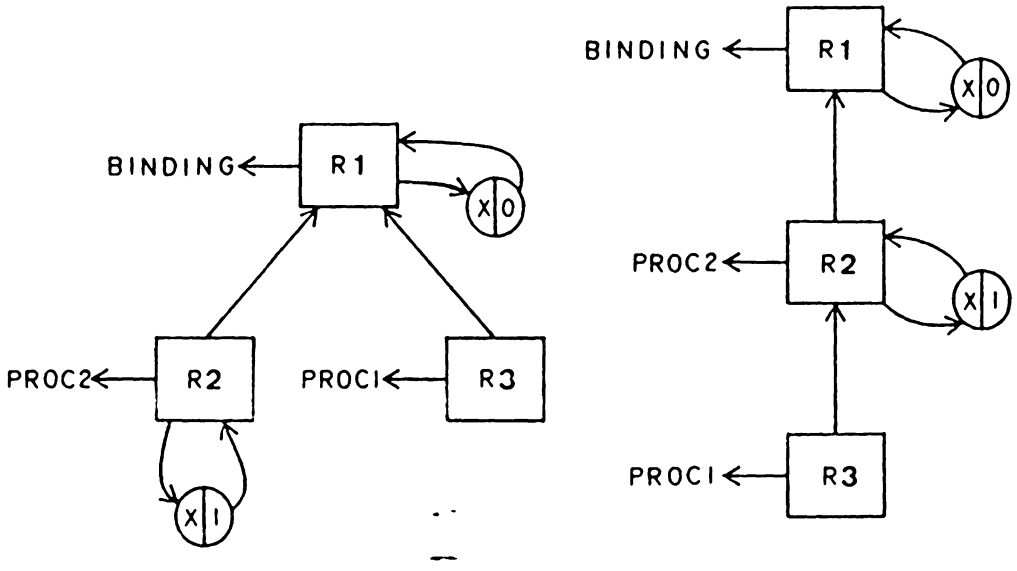
$$AH(PROC2, \langle BINDING \rangle) \rightarrow \uparrow BINDING$$

and so, the environment link of PROC2 also points to BINDING.

Now, the actions of the virtual processor which builds the record contours are described. At first, the ip of the virtual processor points to the PROGRAM code list while the ep is null. When this Pascal program is being executed, a record contour (R1) is created for the main program, and the antecedent link of this record contour points to the program contour BINDING (Figure 4.15(a)). When the ip moves to the code list:

(:=, X, 0)

ep points to R1; and based on the information from the code list, the assocation list of this record contour has an association node for the identifier X which has a value of zero. When the call of PROC2 becomes the next code list encountered by ip, another record contour (R2) is created,



(a) (b)

Figure 4.15: Record skeleton of Figure 2.1

and R2's antecedent link points to program contour PROC2. Now, assume the STAT_CBS is used:

STAT_CBS = (AH, AH, STAT)

(The first AH is for program contour search mechanism, it is determined at compile-time and has been shown in Figure 4.14). The AH record contour search mechanism, with the identifier PROC2 and the current environment <R1> as the parameters, produces:

$$AH(PROC2, \langle R1 \rangle) \dashrightarrow \uparrow R1$$

The STAT_CBS will take the pointer produced by AH as one of the parameters, and the current record environment as the other parameter, yields:

$$STAT(\uparrow R1, \langle R1 \rangle) \dashrightarrow \uparrow R1$$

Therefore, the environment link of R2 is pointing to R1. In executing PROC2, the ip points to the the code list:

(:=, X, 1)

and ep points to R2. After setting the association list based on the information of the code list and the declaration list of the program contour PROC2, the ip moves to the code list:

```
(CALL, PROC1, ())
```

which calls the procedure PROC1. A new record contour (R3) is created and using the STAT_CBS again:

$$AH(PROC1, <R2, R1>) \text{ --> } \uparrow R1$$
$$STAT(\uparrow R1, <R2, R1>) \text{ --> } \uparrow R1$$

Thus, the environment link of R3 points to R1 also. When PROC1 is being executed, the ip moves to:

```
(WRITELN, X)
```

in the code list:

```
(PROCEDURE, PROC1, (), (WRITELN, X))
```

ep points to R3. Since R3 does not have any association for X, the virtual processor following the environment link of

R3 to R1. In R1, X has the association node and the value of zero in it, so the value printed is zero.

If DYN_CBS is used in this Pascal program the record skeleton is different (Figure 4.15(b)). The following steps show why this is so.

When ip is at the code list (CALL, PROC2, ()); ep is at R1:

$$AH(PROC2, \langle R1\rangle) \dashrightarrow \uparrow R1$$
$$DYN(\uparrow R1, \langle R1\rangle) \dashrightarrow \uparrow R1$$

Thus, the environment link of R2 points to R1. When ip is at the code list (CALL, PROC1, ()); ep is at R2:

$$AH(PROC1, \langle R2, R1\rangle) \dashrightarrow \uparrow R1$$
$$DYN(\uparrow R1, \langle R2, R1\rangle) \dashrightarrow \uparrow R2$$

Therefore, the environment link of R3 points to R2. When the ip is at code list:

(WRITELN, X)

ep points to R3. Since R3 does not have any association for X, ep follows the environment link to R2. The value of X is associated with one and this is the value printed.

Chapter V
IMPLEMENTATION

The five modules described in the last chapter communicate with each other to constitute an interactive language environment. The communication among them can be implemented in two ways. First, all the modules can be in one process. That is, the modules communicate with each other in one program using procedure calls. Second, the five modules can be five different processes. In this case, a disk file should be used to store the program and the record skeletons for the modules to access or modify the skeletons. Since the five processes are in five different programs, synchronization mechanism should be established among the processes. This can be accomplished by using event flags or semaphores.

The first version of PEEP uses the first method. The following sections are a detailed description of how the different modules of PEEP were implemented on a VAX-11/780 computer under VMS.

5.1 COMMAND DISPATCHER

The command dispatcher acts like a master module in PEEP. It is responsible for the invocation of the editor, the translator, and the interpreter. After invocation, those modules return to the command dispatcher and await another command from the user. In the current version, the normal way to terminate PEEP is get into the edit mode of the editor, and to use the editor command FILE or QUIT. The pseudo code algorithm for the command dispatcher (CMDDSP) is given below:

```
PROCEDURE CMDDSP;
  BEGIN
    loop := true;
    WHILE loop DO
      BEGIN
        accept a character;
        CASE character OF
          E : BEGIN call INPUTMODE; loop := false END;
          S : call INTER(S);
          O : call INTER(O);
          U : move cursor up one line;
          D : move cursor down one line;
          CARRIAGE RETURN:
              call EDITMODE
        END {case}
    END {while}
  END; {CMDDSP}
```

5.2 <u>EDITOR</u>

SAM is a line-oriented text editor. In programming environments, structured editors or syntax-directed editors are usually used, for example: [ALBE81, TEIT75, TEIT81, SHAP80]. PEEP uses the text editor SAM because of its file handling, editing capabilities, and screen handling.

The detailed structure of SAM is not described in this thesis, interested readers should see [EHRI81]. In order to use the editor SAM for this interactive language environment, SAM has been broken down into three separate subroutines. They are:

1. INITSAM -- contains all the initialization for the SAM editor.

2. EDITMODE -- edit mode of SAM, carries out all the editing abilities of the editor.

3. INPUTMODE -- input mode of SAM:

```
PROCEDURE INPUTMODE;
  BEGIN
    accept a line of Pascal program and
      put it into an input buffer;
    insert this line into the working
      storage
  END; {INPUTMODE}
```

5.3 TRANSLATOR

The components of the translator are shown in Figure 5.1. The compiler-compiler and the driver routine LLDRV are called LLPARS [MORS79], which are system programs supplied by Digital Equipment Corporation using on the VAX-11/780 under VMS. The LL(1) translation grammar is a BNF-like

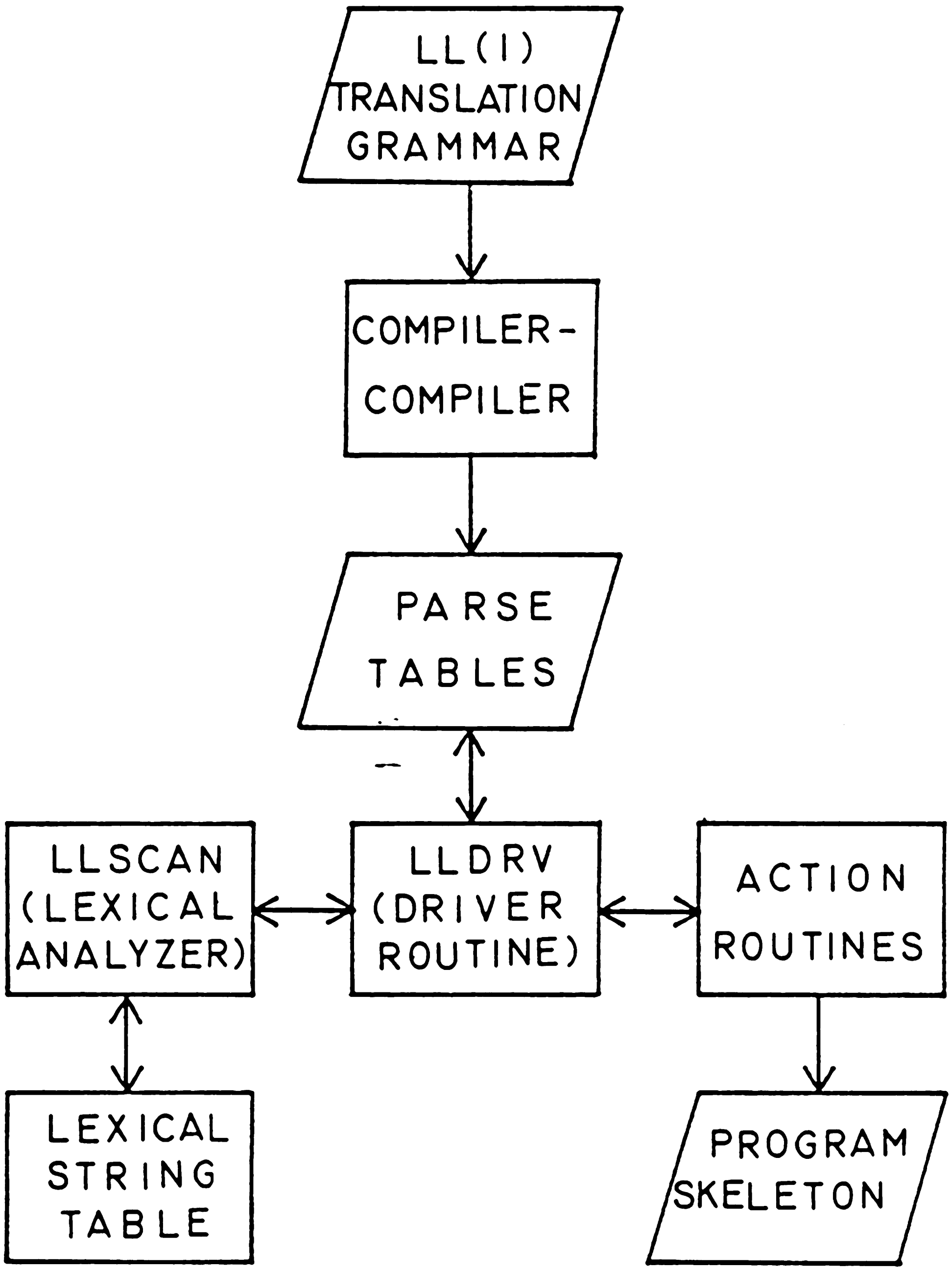


Figure 5.1: Components of the translator

grammar specification for the Pascal language. Since it is a translation grammar, action routine calls, together with terminals and non-terminals, are embedded in the production rules of the grammar, while the actual code of the action routines are put into a separate file. The lexical analyzer, LLSCAN, is a scanner for the translator. LLSCAN works with a lexical string table which contains all the identifiers except keywords in a Pascal program. Every time LLSCAN is called by LLDRV, the scanner returns with a token to the driver. As described in [MORS79], the actions of the different components of the translator are given in the following paragraph.

The compiler-compiler accepts the translation grammar as input and produces a set of parse tables. The driver routine linked with these tables, forms a parser for the language specified by the grammar. When a line of Pascal program is entered, the parser carries out a top-down parse of the input under control of the parse tables, calling the scanner, whenever necessary, to supply the next token in the input. Basically, LLDRV is a machine that executes a set of "moves" which depend on the current "state" of the machine, and the next token in the input. The state of the machine is contained in a pushdown stack. This stack contains a unique "bottom marker". Stacked on top of the bottom marker may be either terminals, non-terminals, or action routine calls. The machine selects the move to execute next on the basis of the symbol on top of the stack, and the next token, as follows:

1. If top-of-stack is a terminal, it should match the next token. If it does, pop the stack and scan for the next input token. If it doesn't, error.

2. If top-of-stack is an action routine, pop the stack and call the routine which will build part of the program skeleton.

3. If top-of-stack is a non-terminal, decide which production rule applies using the table. Pop the non-terminal off the stack, then push the selected right side onto the stack, symbol by symbol, so that its first symbol becomes the new top-of-stack. If no right side can be applied for the next token, error.

4. If top-of-stack is the bottom marker, terminate with success.

5.4 INTERPRETER

The interpreter is responsible for the building of the record skeleton. It simulates the working of the virtual processor described in the last chapter. In the current implementation, it recognizes the commands "S" and "O"; the former executes a single statement, and the latter executes each operation within a statement.

```
PROCEDURE INTER(opcode);
  BEGIN
    CASE opcode OF
      S : execute a statement;
      O : IF cursor is at the BEGIN statement
          THEN build the record skeleton based
            on the pair <ip, ep>
          ELSE execute a single operation
    END {case}
  END; {INTER}
```

5.5 STATE EXAMINER AND MODIFIER

This module is responsible for the examination and modification of the state of the program. It is essential for the debugging process as an aid to understanding program behavior.

5.6 FIRST VERSION OF PEEP

In addition to the algorithms described above, this section desrcibes the rest of the generalized overall algorithms for the first version of PEEP. The algorithms have been greatly simplified to give an idea of how the first version of PEEP is constructed.

The main program of PEEP initializes the command dispatcher, translator, interpreter and the SAM editor. Then LLDRV, the system subroutine of the parser for the Pascal language, is called.

```
PROGRAM PEEP;
  BEGIN
    initializes the command dispatcher;
    initializes the translator;
    initializes the interpreter;
    initializes SAM editor;
    call LLDRV
  END; {PEEP}
```

LLDRV, based on the parse tables produced by the compiler-compiler, calls LLSCAN for a token. If a correct token is found, then, LLDRV carries a top-down parse under control of the parse tables. If an erroneous token is found messages will be printed and LLDRV quits because error recovery routines have not been embedded in the grammar specification. The following algorithm is a simplified description of how LLDRV works as supplied by Digital Equipment corproation.

```
PROCEDURE LLDRV;
  BEGIN
    LOOP
      call LLSCAN;
      carries out a top-down parse of the token
        under control of the parse table
    FOREVER
  END; {LLDRV}
```

LLSCAN, the lexical scanner for the translator, looks at the input buffer. If the input buffer is empty, LLSCAN calls the command dispatcher; otherwise, it scans the input buffer and returns one token from the input buffer to LLDRV.

```
PROCEDURE LLSCAN;
  BEGIN
    IF input buffer is empty OR the input
      line has been scanned
    THEN call CMDDSP;
    scan the input buffer and returns a
      token
  END; {LLSCAN}
```

Chapter VI
FUTURE RESEARCH

The first version of PEEP serves as a test bed for further research on the language environment. As seen from this first version, many features mentioned in Chapter II have not been implemented. Many more capabilities can also be incorporated into PEEP. The following features, alternative implementations, suggested experiments, and possible extensions should be considered after evaluation of the feedback from the first version.

6.1 FEATURES

1. Static Analysis

Static analysis operates on the external syntactic structure of a program; that is, it checks the potential errors from the program skeleton. Currently, PEEP only does syntax checking and declaration checking on the language constructs. More static analysis can be checked from the static structure of a program without running it. The following static analyses from [FAIR80] are good examples: variables set but not subsequently used; isolated code segments that can never be executed; mismatches between the actual and formal parameter list; input parameters passed but never used; output parameters set but never used; etc.

2. Dynamic Analysis

Dynamic analysis is the testing function in software engineering. It operates when the program is in a state of execution. A dynamic analyzer provides capabilities such as selective tracing, statement execution counts, time analysis, results of run time assertion checking, and value ranges for selected variables [FAIR80].

PEEP may incorporate these abilities so that the testing function of the development process can be carried out in the language environment.

3. Debugging Facilities

In [FAIR80], ten source level debugging support facilities were described. They are:

1. Diagnostic Output Statements,

2. Structured Snapshot Dumps,

3. Selective Trace Facility,

4. Assertion Controlled Diagnostics, Dumps, and Traces,

5. Read Only Assertion Breaks,

6. Traceback Assertion Breaks,

7. Assertion Modifying Assertion Breaks,

8. State Modifying Assertion Breaks,

9. Local Source Modifying Assertion Breaks,

10. Program Modifying Assertion Breaks.

This is a hierarchy in which each higher level has more powerful debugging facilities than the lower one. These debugging facilities provide different levels of interactiveness in the logical program checking. These debugging features on the source code level give a good sampling of facilities to incorporate into PEEP.

6.2 IMPLEMENTATION

1. Structured Editor

In the current implementation of PEEP, the editor is in a separate environment. Once the editor is being used, it cannot go back into the language environment, because the editor can only change the text of the program but not the internal data structures. A structured editor should be used in this situation to change the text of a program as it appears on the screen and also to have the capabilities of changing the internal data structures (program and record skeletons) which represent the program. Since the internal data structure reflects the external construct of a program, the editor must be a language-specific editor.

2. <u>Use Disk File or Global Section</u>

As mentioned in the chapter on implementation (Chapter V), PEEP has been implemented, using one process. But the second method, i.e., using a disk file, seems to be a more logical approach for the development of a large software package such as PEEP. At least, it is easier to conceptualize the structure of PEEP by the use of five processes for the five modules.

In VAX-11/780 (VMS), global sections can be used. Global sections constitute an interprocess communications facility where several processes communicate through shared memory pages [LEVY80].

Thus, the program skeleton, record skeleton, and sharable codes which build these skeletons can be put into the global section. The five processes can be synchronized with the common event flags which are also available on the VAX-11/780 under VMS.

3. Full Pascal

Currently, PEEP only works for a subset of Pascal. Full Pascal should be used on the language environment so that PEEP would become a useful program development environment. To implement the extensions from a subset of Pascal to full Pascal, the first step is to incorporate the LL(1) translation grammar of full Pascal; then incorporate the associated action routines which build the program skeleton; and the final step is to expand the capabilities of the interpreter for building the record skeleton.

4. Other Base Language

Theoretically, high-level base language other than Pascal can be used in PEEP by changing the grammar, the associated action routines, and the interpreter so that the semantics of the new language can be properly interpreted. Actually, it will be much easier for PEEP to use another high level language if that language is ALGOL-like, or if it is a block structured language.

5. Terminal Independency

PEEP only works on terminals which have the capability of changing from 80 columns to 132 columns in the screen. The screen layout routine in PEEP is tailored to the specific characteristics of VT100 with Advanced Video Option. If a user uses PEEP on other terminals, PEEP will recognize the terminal type and an error message will be printed. Different screen layout routines suitable for different kinds of terminals should be put into PEEP, so that PEEP can be operated with other terminals.

6.3 EXPERIMENTATION

PEEP was developed so that experiments can be conducted in an interactive language environment. In Chapter II, two kinds of experiments are mentioned. These experiments, as well as more experiments on human-computer interface issues, display methods, and human-factors in systems will be conducted. Two examples which are important to future research on such systems are given below. These experiments should be designed to evaluate their significance in human-computer interactions.

1. Single Environment

   PEEP is designed to be a single environment system. That means the user can change, execute, or debug a program in only one mode without switching from one tool to another like the editor and compiler in ordinary programming systems. Key issues are whether such a single environment can increase a programmer's productivity and whether such a system can improve user-friendliness.

2. Program Development Methodology

   PEEP integrates tools in one highly interactive environment. Syntax and other static errors are reported as soon as the translator detects them so that the user can determine if everything is correct at the beginning of the program development process. An analysis and evaluation of such a development methodology are needed to determine its impact on software development.

6.4 EXTENSIONS

1. Dialogue Management System (DMS)

Better human-computer communication can improve programming productivity and reduce the number of programming mistakes. Interface features such as menu selection, graphics, and voice would be good communication media for the user.

Research is now being done at Virginia Tech for the development of DMS [ROAC82]. It is a system which provides flexible procedures for facilitating human-computer communications. PEEP can depend on DMS to optimize human-computer interfaces. DMS will provide the means by which the interface for the language environment can be readily changed with respect to dialogue content.

2. Program Verifier

More software tools can be incorporated into PEEP to make the language environment an effective program development software package. One of the software tools which draws a great deal of research attention is the program verifier. While interface issues aim at the programmer productivity aspect of programming environments, program verification addresses the software quality aspect. The program verifier gets involved in making assertions in predicate calculus at various points in a program text. The verifier then enters this program with the assertions to check the correctness of the program.

PEEP offers great potential to work with a verifier. Since PEEP allows execution at the statement level, the block or procedure level, and the program level, the assertions for each statement can always be checked. Thus, the verifier can check the correctness of a program incrementally.

6.5 FINAL COMMENT

The first version of PEEP is just the beginning of the research on this language environment. The main purpose of this research is the experimentation on human-computer interaction and decision behavior. PEEP is the medium to carry out these experimentations. Therefore, when a decision has to be made concerning efficiency and capability, capabilities always take precedence. Efficiency is not the main concern for the construction of PEEP.

The suggestions on implementations, experimentations, and extensions as described above are only speculations on features that might be incorporated into PEEP. It is too early to tell whether those suggestions are feasible or practical to implement.

BIBLIOGRAPHY

[ALBE81] Alberga, C. M., Brown, A. L., Leeman, G. B. Jr., Mikelsons, M. and Wegman, M. N., "A Program Development Tool," Conference Record of the 8th Annual ACM Symposium on Principles of Porgramming Languages, Williamsburg, Virginia, January 26-28, 1981, pp.92-104.

[ALEX75] Alexander W. G. and Wortman, D. B., "Static and Dynamic Characteristics of XPL Programs," Computer, Vol.8, No.11, November 1975, pp.41-46.

[ARCH80] Archer, J., Conway, R., Shore, A. and Silver, L., "The CORE user Interface," Technical Report, TR80-437, Department of Computer Science, Cornell University, Ithaca, New York, September 1980.

[BING76] Bingham, H. W. and Carvin, K. T., "Dynamic Usage of APL Primitive Functions," APL-76, ACM-STAPL Conference, Ottawa, September 22-24, 1976, pp.83-86.

[BRAN81] Branstad, M. A. and Adrion, W. R. (Eds.), NBS Programming Environment Workshop Report, U. S. Government Printing Office, Washington, 1981.

[BUXT70] Buxton, J. and Randell, B. (Eds.), Software Engineering Techniques, Report on a conference sponsored by the NATO Science Committee, Rome, Italy, OOctober 27-31, 1969, Published in April 1970.

[CHEV78] Chevance, R. J. and Heidet, T., "Static Profile and Dynamic Behavior of COBOL Programs," SIGPLAN Notices, Vol.13, No.4, April 1978, pp.44-57.

[CHRY78] Chrysler, E., "Some Basic Determinants of Computer Programming Productivity," Communications of the ACM, Vol.21, No.6, June 1978, pp.472-483.

[CLAR77] Clark, D. W. and Green, C. C., "An Empirical Study of the List Structure in LISP," Communications of the ACM, Vol.20, No.2, February 1977, pp.78-86.

[COME79] Comer, D. and Halstead, M. H., "A Simple Experiment in Top-Down Design," IEEE Transactions on Software Engineering, Vol.SE-5, No.2, March 1979, pp.105-109.

[DOLO78] Dolotta, T. A., Haight, R. C. and Mashey, J. R., "The Programmer's Workbench," The Bell System Technical Journal, Vol.57, No.6, July-August 1978, pp.2177-2200.

[EHRI81] Ehrich R. W., "SAM -- A Configurable Experimental Text Editor for Investigating Human factors Issues in Text Processing and Understanding," Technical Report, CSIE-81-3, Department of Computer Science and Department of Industrial Engineering and Operations Research, Virginia Polytechnic Institute and State University, Blacksburg, Virginia, September 1981.

[FAIR80] Fairley, R. E., "Ada Debugging and Testing Support Environment," SIGPLAN Notices, Vol.15, No.11, November 1980, pp.16-25.

[GANN77] Gannon, J. D., "An Experimental Evaluation of Data Type Conventions," Communications of the ACM, Vol.20, No.8, August 1977, pp.584-595.

[GHEZ82] Ghezzi, C. and Jazayeri, M., Programming Language Concepts, John Wiley & Sons, Inc., 1982, p.44.

[GOUL75] Gould, J. D., "Some Psychological Evidence on How People Debug Computer Programs," International Journal of Man-Machine Studies, Vol.7, No.2, March 1975, pp.151-182.

[HABE79] Habermann, A. N., "An Overview of the Gandalf Project," Computer Science Research Review 1978-1979, Carnegie-Mellon University, Pittsburg, Pennsylvania, 1979.

[HUNK80] Hunke, H. (Ed.), Software Engineering Environments, North-Holland Publishing Company, 1980.

[JOHN73] Johnston, J. B., "Identifier Binding and Access in Nested Declaration Computations," Proceedings of the Seventh Annual Princeton Conference on Information Sciences and Systems, March 22-23, 1973, pp.306-312.

[JOHN78] Johnson, S. C. and Ritchie, D. M., "Portability of C Programs and the UNIX System," Bell System Technical Journal, July-August 1978, pp.2021-2048.

[KNUT71] Knuth, D. E., "An Empirical Study of FORTRAN Programs," Software--Practice & Experience, Vol.1, No.2, April-June 1971, pp.105-133.

[LEVY80] Levy, H. M. and Eckhouse, R. H. Jr., Computer Programming and Architecture--The VAX-11, Digital Equipment Corporation, 1980, p.341.

[LIND81a] Lindquist, T. E. and Johnston, D. H., "An Empirical Evaluation of the Relationship between Programmer Performance and Scoping Strategies," Technical Report, Department of Computer Science, Virginia Polytechnic Institute and State University, Blacksburg, Virginia, August 1981.

[LIND81b] Lindquist, T. E. and Ku, C. S., "The Design of a Language Environment for Evaluating the Programming Task," Technical Report, Department of Computer Science, Virginia Polytechnic Institute and State University, Blacksburg, Virginia, August 1981.

[LITE76] Litecky, C. R. and Davis, G. B., "A Study of Errors, Error-Proneness and Error Diagnosis in COBOL," Communications of the ACM, Vol.19, No.1, January 1976, pp.33-37.

[MITZ81] Mitze, R. W., "The UNIX System as a Software Engineering Environment," Software Engineering Environments, Hunke, H. (Ed.), North-Holland Publishing Company, 1981, pp.345-357.

[MORS79] Morse, J. A., LLPARS Users Manual, Applied Research and Development, Digital Equipment Corporation, Maynard, Massachusetts, September 23, 1979.

[NAUR69] Naur, P. and Randell, B. (Eds.), Software Engineering, Report on a conference sponsored by the NATO Science Committee, Garmisch, Germany, October 7-11, 1968, Published in January 1969.

[RIDD80] Riddle, W. E. and Fairley, R. E. (Eds.), Software Development Tools, Springer-Verlag, Germany, 1980.

[RITC78] Ritchie, D. M. and Thompson, K., "The UNIX Time-Sharing System," The Bell System Technical Journal, Vol.57, No.6, July-August 1978, pp.1905-1929.

[ROAC82] Roach, J., Hartson, H. R., Ehrich, R. W., Yunten, T., and Johnson, D. H., "DMS: A Comprehensive System for Managing Human-Computer Dialogue," Proceedings Human Factors in Computer Systems, Gaithersburg, Maryland, March 15-17, 1982, pp.102-105.

[SALV75] Salvadori, A., Gordon, J. and Capstick, C., "Static Profile of COBOL Programs," SIGPLAN Notices, Vol.10, No.8, August 1975, pp.20-33.

[SHAP80] Shapiro, E., Collins, G., Johnson, L. and Ruttenberg, J., "PASES: A Programming Environment for Pascal," Computer Science Department, Yale University, April 1980.

[STON80] "Stoneman," Requirements for Ada Programming Support Environment, U. S. Department of Defense, February 1980.

[TEIT75] Teitelman, W., INTERLISP Reference Manual, Xerox Palo Alto Research Center, California, 1975.

[TEIT81] Teitelbaum, T. and Reps, T., "The Cornell Program Synthesizer: A Syntax-Directed Programming Environment," Communications of the ACM, Vol.24, No.9, September 1981, pp.563-573.

[VT1079]
VT100 User Guide, Digital Equipment Corporation, Maynard, Massachusetts, 1979.

[WATE79] Waters, R. C., "A Method for Analyzing Loop Programs," IEEE Transactions on Software Engineering, Vol.SE-5, No.3, May 1979, pp.237-247.

[WEGB74] Wegbreit, B., "The Treatment of Data Types in EL1," Communications of the ACM, Vol.17, No.5, May 1974, pp.251-264.

[WOOD79] Woodfield, S. N., "An Experiment on Unit Increase in Problem Complexity," IEEE Transactions on Software Engineering, Vol.SE-5, No.2, March 1979, pp.76-79.

[YOUN74] Youngs, E. A., "Human Errors in Programming," International Journal of Man-Machine Studies, Vol.6, No.3, May 1974, pp.361-376.

[ZELK76] Zelkowitz, M. V., "Automatic Program Analysis and Evaluation," Proceedings Second International Conference on Software Engineering, San Francisco, California, October 13-15, 1976, pp. 158-163.

THE DESIGN AND IMPLEMENTATION OF A LANGUAGE ENVIRONMENT FOR EVALUATING THE PROGRAMMING TASK

by

Cyril Shiu-Chin Ku

(ABSTRACT)

The thesis describes the requirement, design, and implementation of a software package that can be used to perform quantitative studies on certain aspects of a programming task. Of specific interest are experiments with the level of interactiveness of the human-computer interface relating to program construction and with language desgin principles relating to identifier scope rules.

The software package for conducting these experiments is an interactive language environment called PEEP. Its base language is Pascal and its design is based on Johnston's semantic models of computation. Storage representations and implementation of these semantic models are described. These models depicting the compile-time structure, run-time structure, and realization of the static and the dynamic scoping rules.

The evolution and current research of programming environments, user interface to PEEP, and future research on PEEP are also focuses of attention.