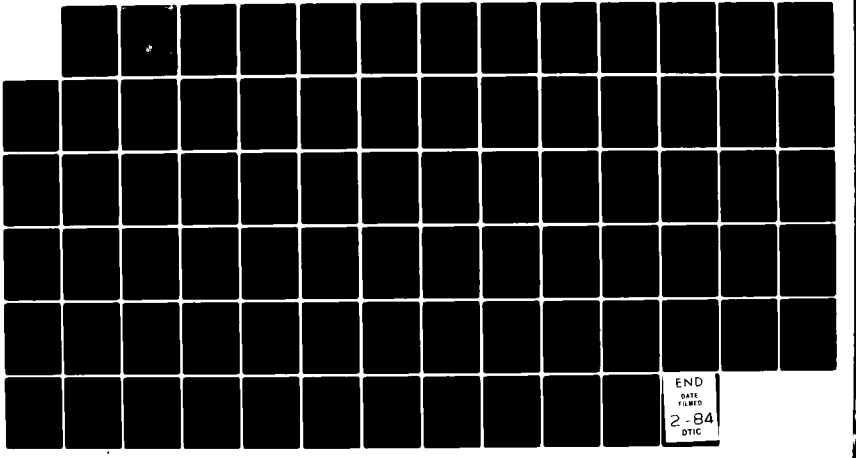
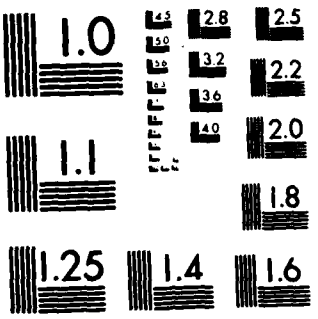
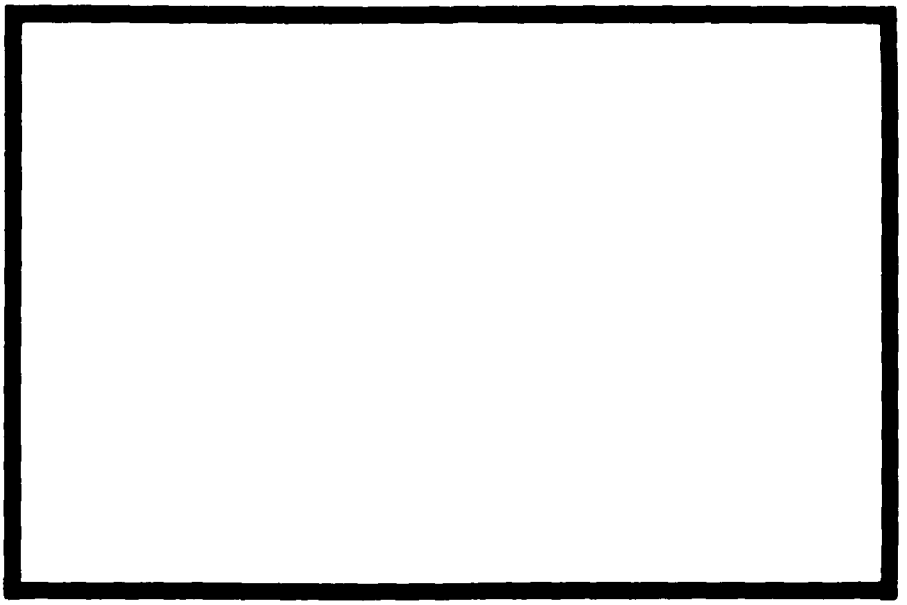
AD-A137 474 PEEP: A PASCAL ENVIRONMENT FOR EXPERIMENTS ON PROGRAMMING(U) VIRGINIA POLYTECHNIC INST AND STATE UNIV BLACKSBURG COMPUTER S.. C S KU ET AL. SEP 82 CSIE-82-9 1/1

UNCLASSIFIED N00014-81-K-0143 F/G 5/8 NL



END
DATE FILMED
2-84
DTIC










# Virginia Polytechnic Institute and State University

Computer Science
Industrial Engineering and Operations Research
BLACKSBURG, VIRGINIA 24061

2

CSIE-82-9 September 1982

# PEEP: A Pascal Environment for Experiments on Programming

Cyril S. Ku

Timothy E. Lindquist

TECHNICAL REPORT

Prepared for
Engineering Psychology Programs, Office of Naval Research
ONR Contract Number N00014-81-K-0143
Work Unit Number NR SRO-101

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

| REPORT DOCUMENTATION PAGE | | READ INSTRUCTIONS BEFORE COMPLETING FORM |
|---|---|---|
| 1. REPORT NUMBER<br>CSIE-82-9 | 2. GOVT ACCESSION NO.<br>AD-A137474 | 3. RECIPIENT'S CATALOG NUMBER |
| 4. TITLE (and Subtitle)<br>PEEP: A Pascal Environment for Experiments On Programming | | 5. TYPE OF REPORT & PERIOD COVERED<br>Technical<br>6. PERFORMING ORG. REPORT NUMBER |
| 7. AUTHOR(s)<br>Cyril S. Ku<br>Timothy E. Lindquist | | 8. CONTRACT OR GRANT NUMBER(s)<br>N00014-81-K-0143 |
| 9. PERFORMING ORGANIZATION NAME AND ADDRESS<br>Computer Science<br>Virginia Polytechnic Institute & State University<br>Blacksburg, VA 24061 | | 10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS<br>61153N42; RR04209;<br>RR0420901; NR SR0-101 |
| 11. CONTROLLING OFFICE NAME AND ADDRESS<br>Office of Naval Research, Code 422<br>800 North Quincy Street<br>Arlington, VA 22217 | | 12. REPORT DATE<br>September 1982<br>13. NUMBER OF PAGES<br>73 |
| 14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) | | 15. SECURITY CLASS. (of this report)<br>Unclassified<br>15a. DECLASSIFICATION DOWNGRADING SCHEDULE |

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

programming environment, binding strategy, level of interactiveness, human factors

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This report describes the requirements, design, and implementation of a software package that can be used to perform quantitative studies on certain aspects of the programming task. Of specific interest will be experiments with the level of interactiveness of the human-computer interface relating to identifier scope-rules.

The software package for the conduct of those experiments is an interactive programming-environment called PEEP. Its base language is Pascal and its

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

20. ABSTRACT (continued)

design is based on a semantic model of computation. Storage representations and the implementation of these semantic models are described. The model depicts the compile-time structure, run-time structure, and realization of the static and the dynamic scoping-rules.

## ACKNOWLEDGMENT

This research was supported by the Office of Naval Research under contract number N00014-81-K-0143 and work unit number NR SRO-101. The effort was supported by the Engineering Psychology Group, Office of Naval Research, under the technical direction of Dr. John J. O'Hare. Reproduction in whole or in part is permitted for any purpose of the United States Government.

CONTENTS

# 1. INTRODUCTION

The trend toward developing systems with friendly human-computer interfaces is well underway. Unfortunately, the underlying criteria used to build user-friendliness into a system are too often based on speculation rather than hard data. To a large extent, the same objection can be made to the design of programming languages and methodologies. With the design goal to provide a usable tool to produce reliable software, too often decisions are based on unsubstantiated principles, or are based on implementation ease. Experimentally validated human-variables and metrics that can be used in making design decisions are difficult to establish, but they are needed to guide the shift to user-friendly interfaces and when applied to programming language design will improve the software development process.

In this report we describe the requirements, specifications, and design of a software package that can be used to perform quantitative studies on the programming task as it relates to the human-computer interface and language design. The system we describe is a programming environment in the software sense; that is, it is not the physical, managerial, or social environment, but instead, it is the virtual language computer that the programmer uses to convert software designs into programs. Programming environments include all types of software development tools whose histories date

back to the early years of computing. Among these tools are assemblers, compilers, interpreters, linkers and loaders, editors, programming languages, run-time libraries, utility routines and documentation aids. In the software sense, this collection makes up the environment in which the programmer must exist while composing software. Currently, related work is being done in such areas as generalized environments, user-friendliness, and tool integration as it relates to the programming environment [BRAN81, HUNK80, RIDD80]. And, we are seeing a healthy shift toward experimental validation of much of the work in these areas. Many successful programming environments have been constructed in the past. Four of the more popular environments are UNIX* [RITC78], INTERLISP [TEIT75], Cornell Program Synthesizer [TEIT81], and LISPEDIT [ALBE81]. Related works in progress include GANDALF [HABE79] at Carnegie-Mellon University, PASES [SHAP80] at Yale University, and CORE [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** language. It is now under development by the Air Force and the Army.

---

* UNIX is a trademark of Bell Laboratories.
** Ada is a trademark of the U. S. Department of Defense (Ada Joint Program Office).

## 1.1 THE NEED FOR EXPERIMENTS

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 and user-friendly activity. Inasmuch 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.

## 1.2 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 programming environment to conduct experiments. PEEP is designed as an environment solely for research on programming 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 provide another motivation for the development of PEEP. APSE, an integrated software development environment, indicates that the current trend of software systems is to have tightly-coupled tools. This, together with the fact that Ada recognizes programming as a human activity, motivates both experimentation in program-development activities and research into software architectures for highly-integrated development environments.

## 1.3 OUTLINE OF REPORT

Section 2 of this report gives an overview of what PEEP looks like to a user focusing on the external features of the programming environment while Section 3 details the internal structures of the system. The final section is devoted to a discussion of the algorithms used to implement the design.

# 2. THE USER INTERFACE TO PEEP

## 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, what we call level 0 of interactiveness. When the internal representation of the program is executed, the level of interac-

tiveness 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 (i.e., at level 0 of interactiveness).

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 Sub Section 2.1.

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 an 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*, an entire statement, or a compound statement. Therefore, execution of a partially completed program is possible. In the debugging process, a break-point assertion in the form of an assert statement can be set at different program units. An assert statement such as:

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

can be placed before a statement, a compound statement, a

---

* 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 cannot be executed without all the information in a statement.)

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 SCREEN LAYOUT

We now describe what PEEP (in its current version), at level 1 of interactiveness, looks like to a user. Figure 2.1 shows the screen format on the terminal when using the programming environment. (Figure is 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.

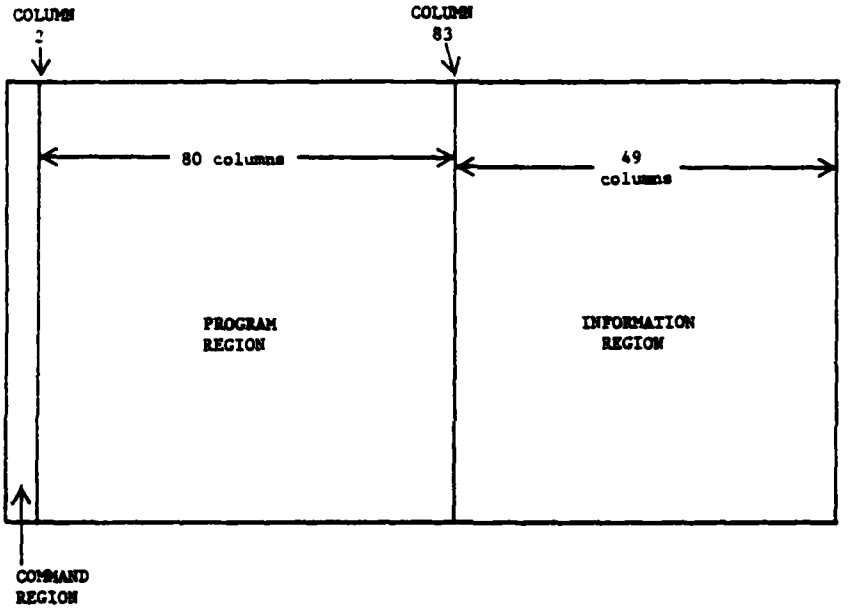



Figure 2.1: Screen format

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 programming environment to the user. For example, syntax errors appear in this region.

## 2.3 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, it-

eration 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.

## 2.4 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 2.2). The user can give the command "E" for entering a line of Pascal program. If the command "E" is typed in column 1, 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 2.3 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

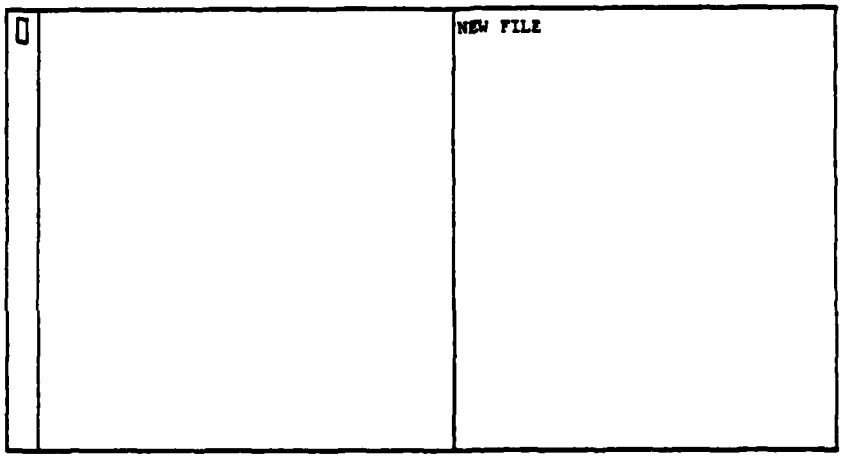



Figure 2.2: PEEP expecting a new Pascal program

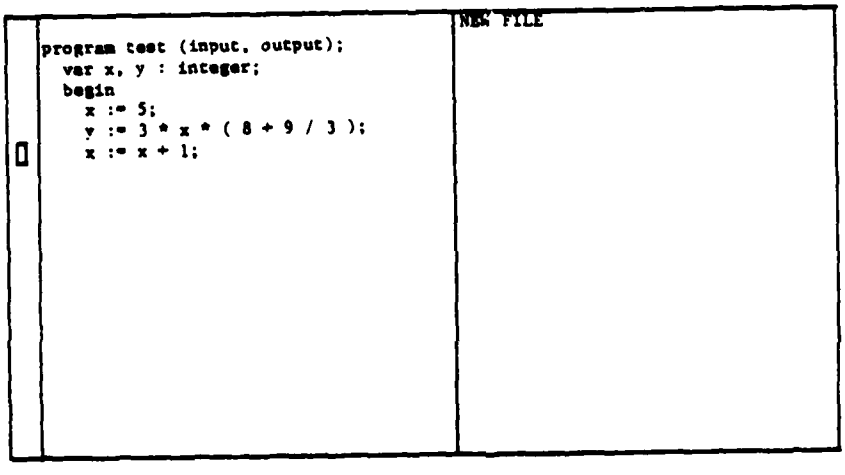



Figure 2.3: A partially-completed program in PEEP

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 2.4) 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 2.4.

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.

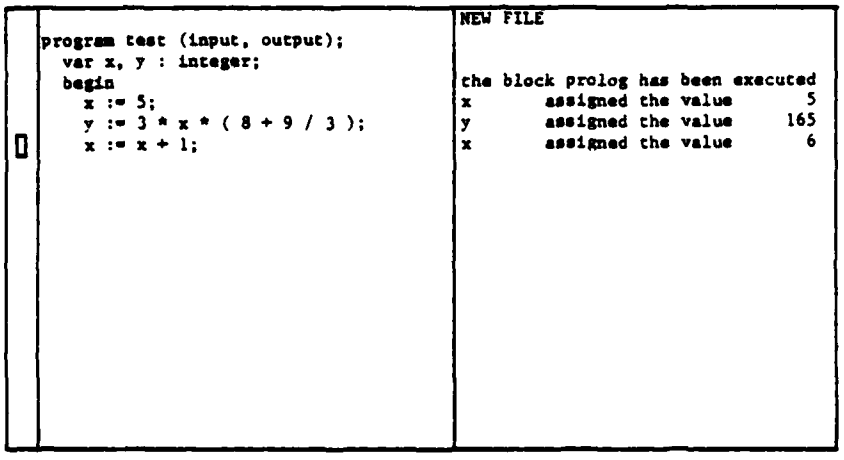



Figure 2.4: Three statements executed in PEEP

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
```

# 3. DESIGN

## 3.1 OVERALL DESIGN

PEEP consists of five main modules: a command dispatcher, an editor, a translator, an interpreter, and a state examiner and modifier (Figure 3.1). 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.

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.

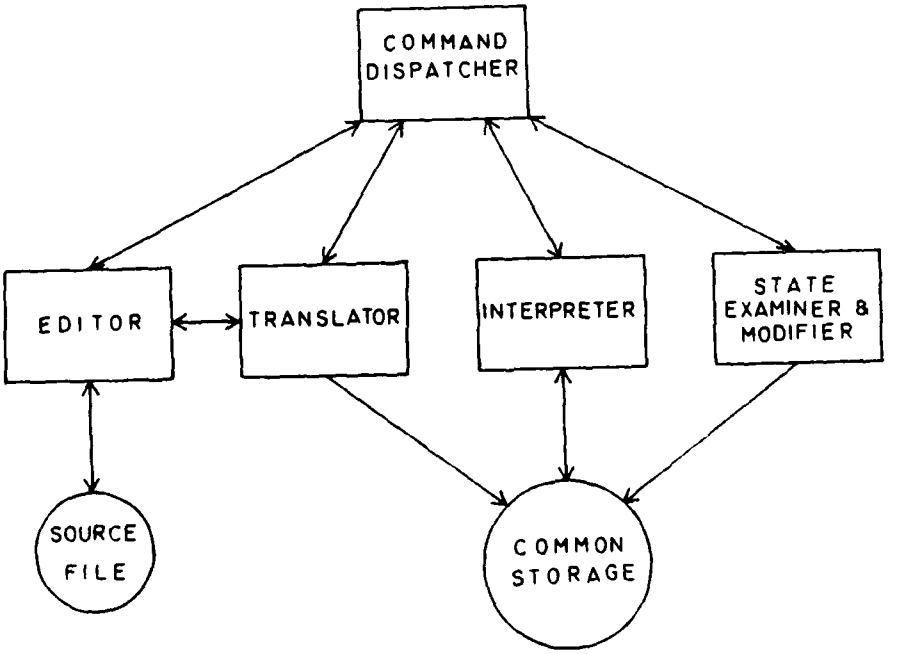



Figure 3.1: Overall design of PEEP

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 subsections describe the representations of the three different forms (text storage, program skeleton, record skeleton) of the source program in detail. Following the description of these storage structures, different binding strategies and their realization by the contours are presented.

## 3.2 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 (Figure 3.2), which is now briefly discussed.

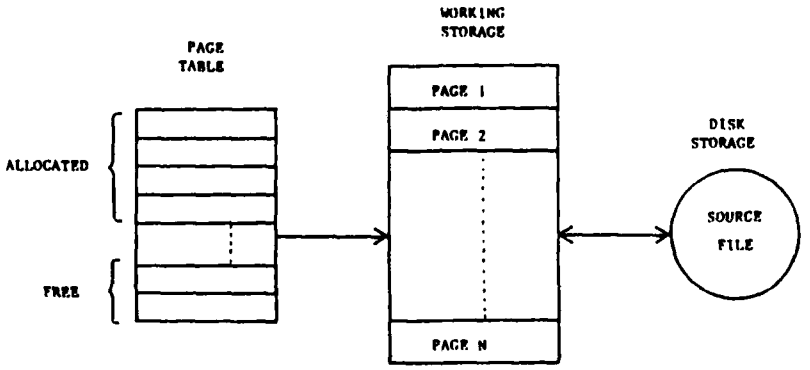



Figure 3.2: SAM's virtual-storage system

The virtual-storage system consists of three data structures -- a queue, the page tables, and the working storage. 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 of a number of fragments which form 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.

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 current working-page. When a line

is edited, the line should be transferred 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.

## 3.3 PROGRAM SKELETON

Figure 3.3 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 call statement to a particular procedure. The nesting nature of this Pascal program (Figure 3.3) can actually be represented by the block structure shown in Figure 3.4. Figure 3.4 shows that this nesting structure is hierarchical, so it can also be represented by a general tree (Figure 3.5).

### 3.3.1 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

```
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 3.3: A pseudo-Pascal program

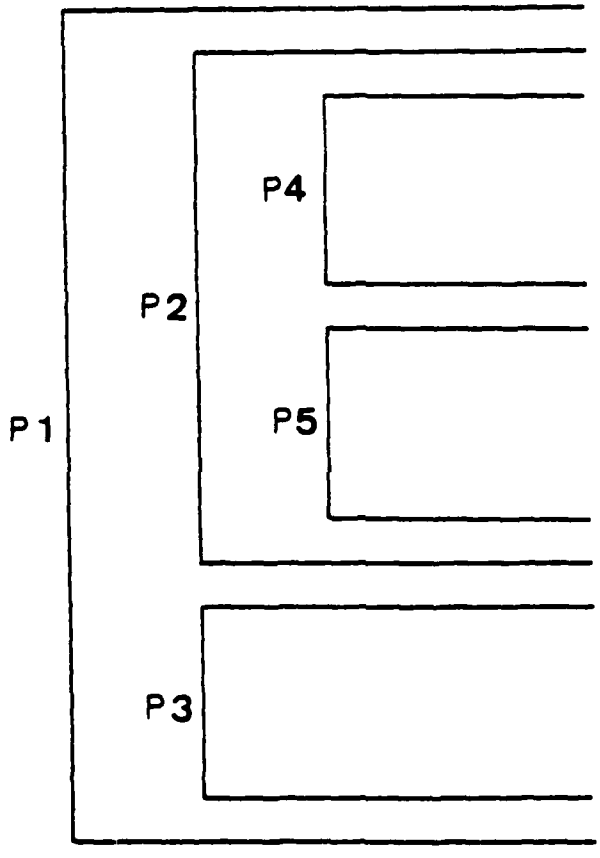



Figure 3.4: Nesting representation of the Pascal program

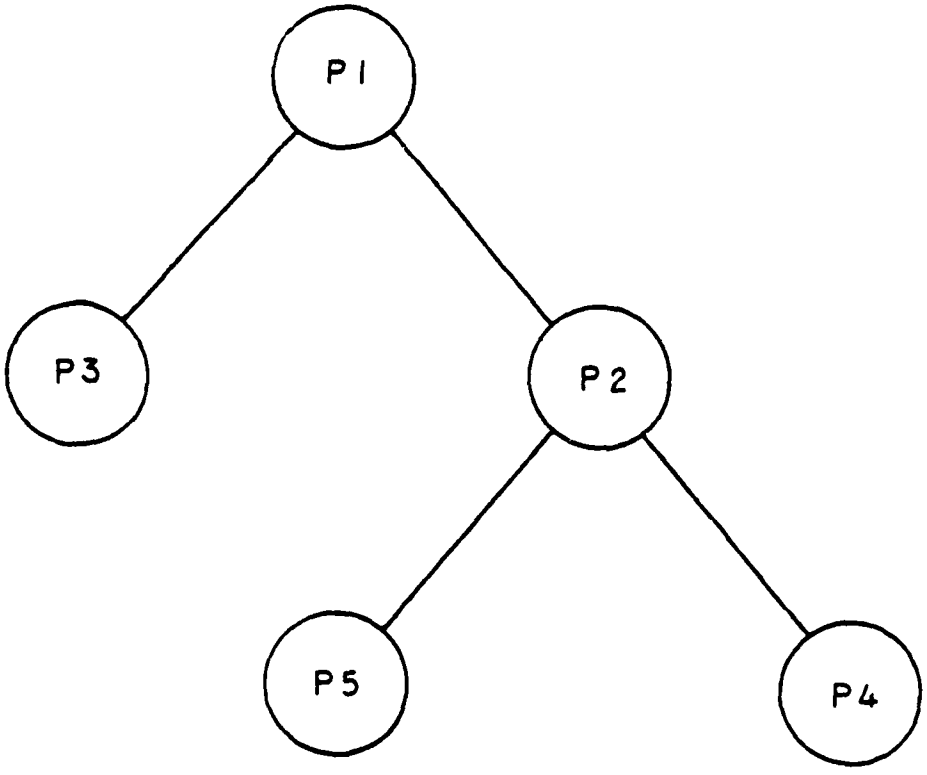



Figure 3.5: Tree representation of the Pascal program

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 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 3.6. As can be seen from the figure, the program contour has an identification subcell named PROGRAM.

### 3.3.2 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

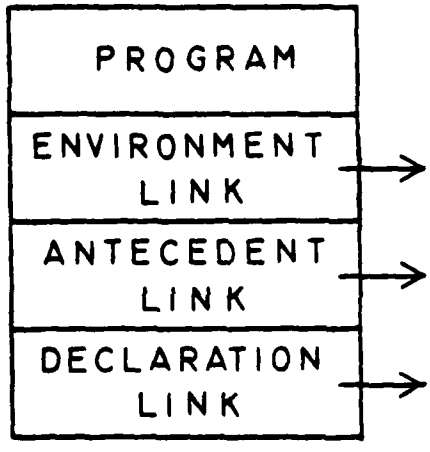



Figure 3.6: A program contour

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 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 a 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.

In figure 3.7, (a) shows the general format for a declaration node, 3.7 (b), (c), (d), (e), and (f) show the different fields of the declarations of constant (c), type (t), variable (v), procedure (p), and function (f), respectively. For (c) declaration node, the information field consists of one lexical-table entry (LTE) and one integer-subfield (INT). The (t) declaration node has a lexical-table entry

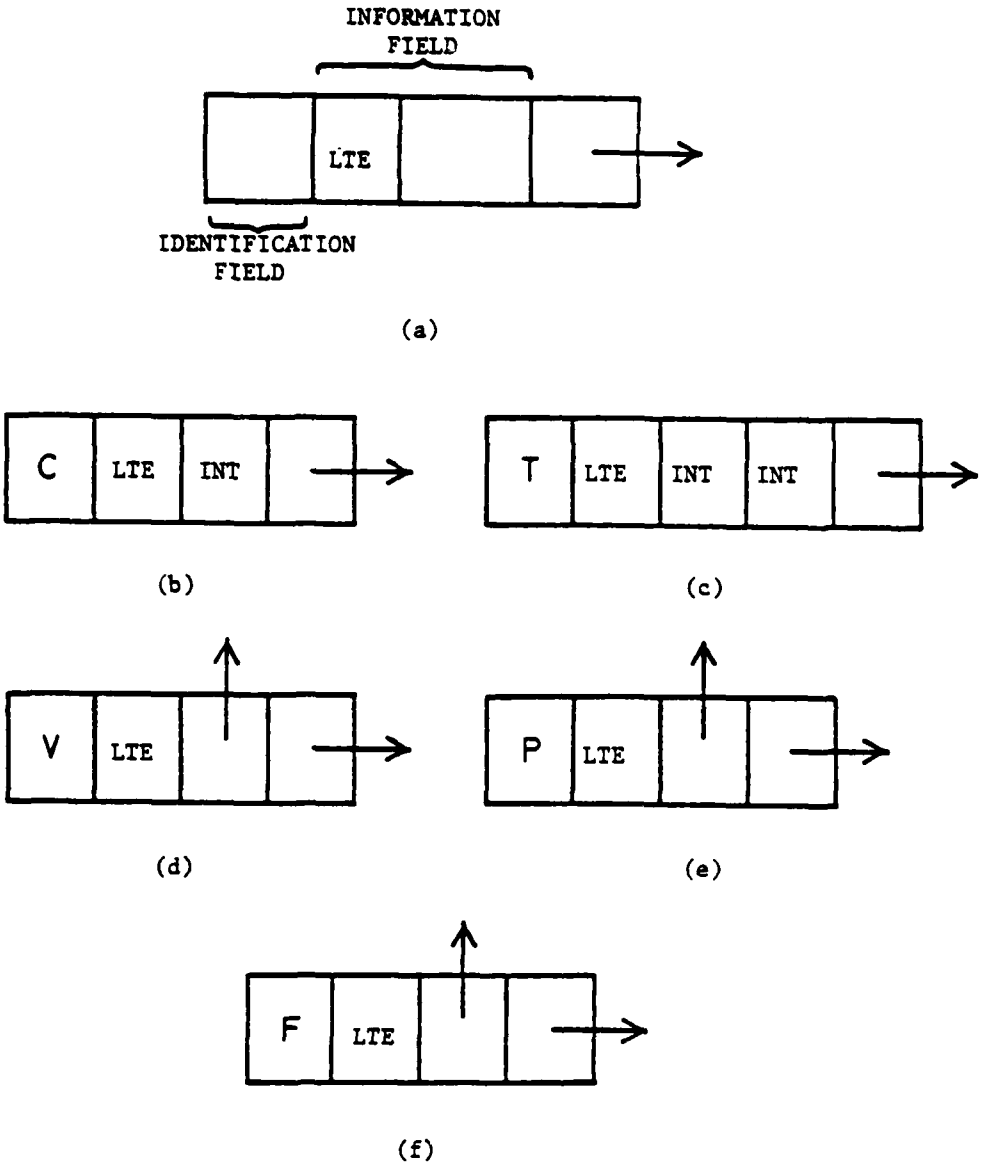



Figure 3.7: Declaration node formats

and two integer-subfields in the information field. The information field of variable, procedure, and function declarations have similar formats. They all contain lexical-table entries and a subfield for a pointer. The pointer field of variable points to a declaration node which contains the type of the variable, while the pointer field 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 involve more than one declaration node. Figure 3.8 gives different examples of declarations and their node representations.

### 3.3.3 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))

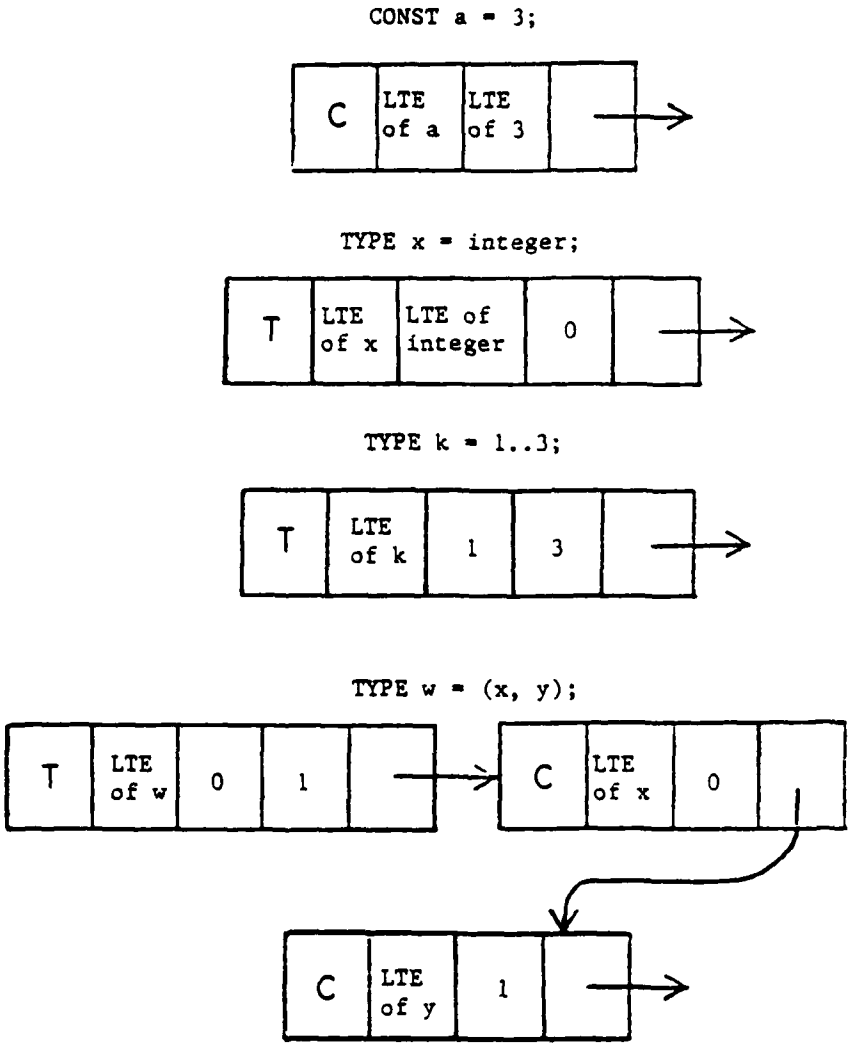



Figure 3.8: Examples of declaration nodes

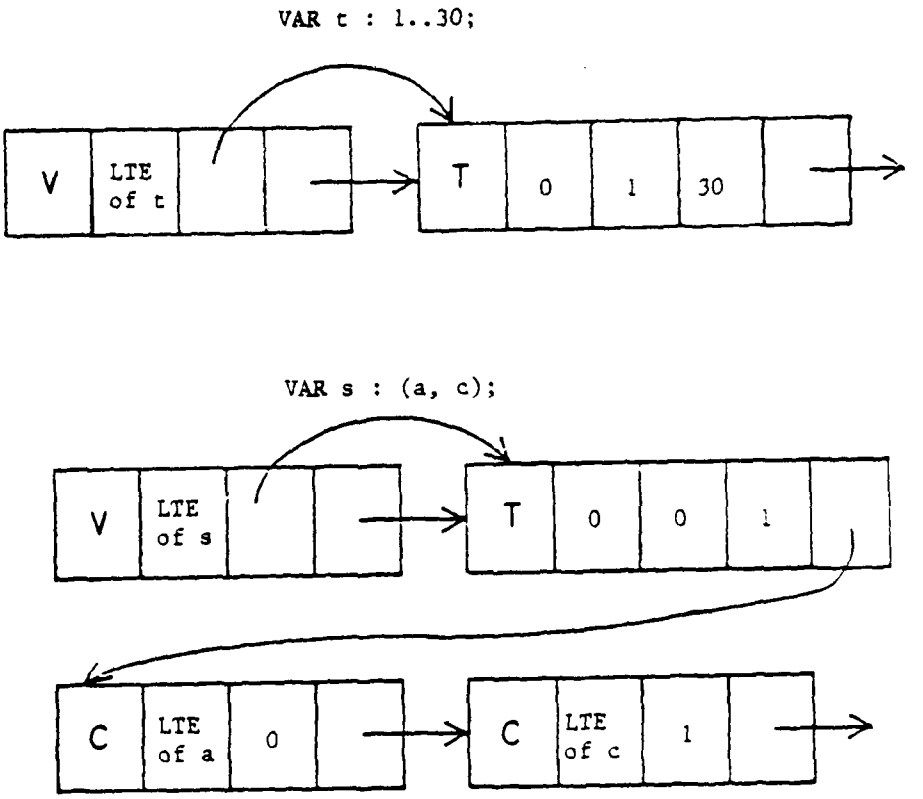



Figure 3.8: cont'd

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))
```

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
```

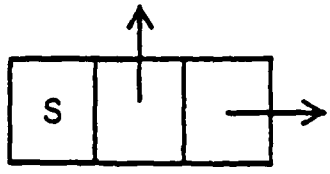
(;, (;, (:=, a, (+, x, y)), (:=, b, f)), (:=, c, k))

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 mecha-

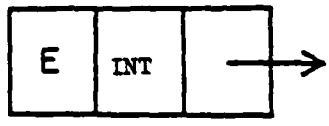
nism (by reference or by value) and one more pointer-field to indicate the parameter type (i.e., the pointer field has a pointer pointing to a declaration node which specifies the type of a parameter). The integer field of an elementary node either contains a lexical-table entry, or a keyword or operator 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 virtual processor is presented in Subsection 3.6). Figure 3.9 shows the three different kinds of nodes in a code list.

### 3.3.4 Diagram of Program Skeleton

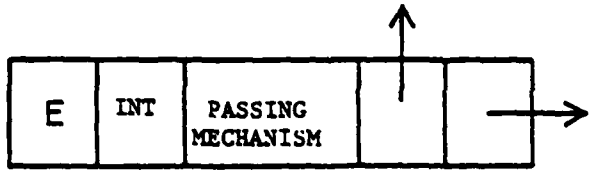
Figure 3.10 shows the program skeleton of the pseudo-Pascal program of Figure 3.3. This figure gives a general structure of the program skeleton; the detail structures can be easily figured out from the descriptions of the program contour, declaration list, and the code list. Let Pn, $n \geq 1$, represent 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 3.10, 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




(a) SUBLIST NODE




(b) ELEMENTARY NODE




(c) ELEMENTARY NODE FOR PARAMETER

Figure 3.9: Code_list node formats

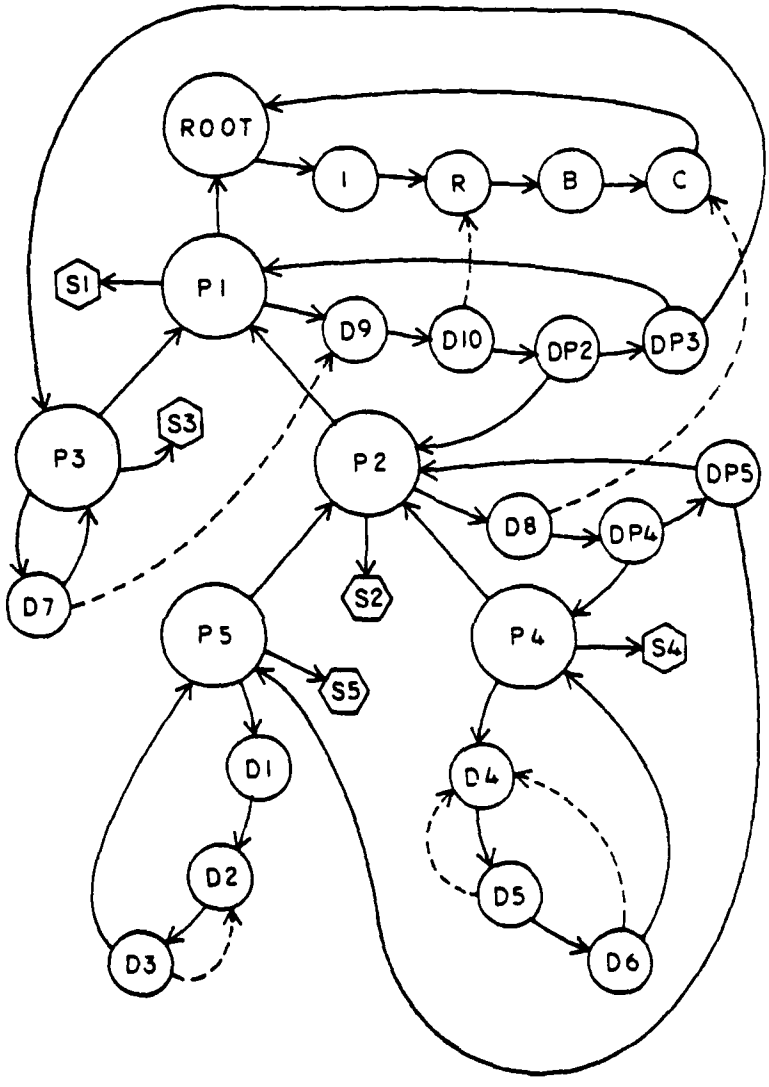



Figure 3.10: Program skeleton of Figure 3.3

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 VAR declaration nodes are using dash arrows).

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

### 3.4.1 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 Subsection 3.7). 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 descendant 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 3.11 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.

### 3.4.2 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 a variable. An association node contains the value of a variable; 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 variable. The type of the variable is stored in the type field, while the value of the variable, 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. In figure 3.12, (a) shows the general format of an association node, while (b), (c), (d), and (e) show the specific examples.

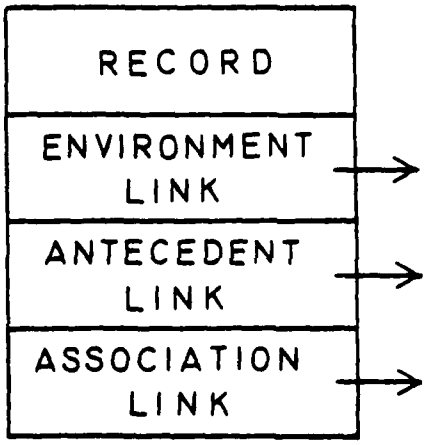



Figure 3.11: A record contour

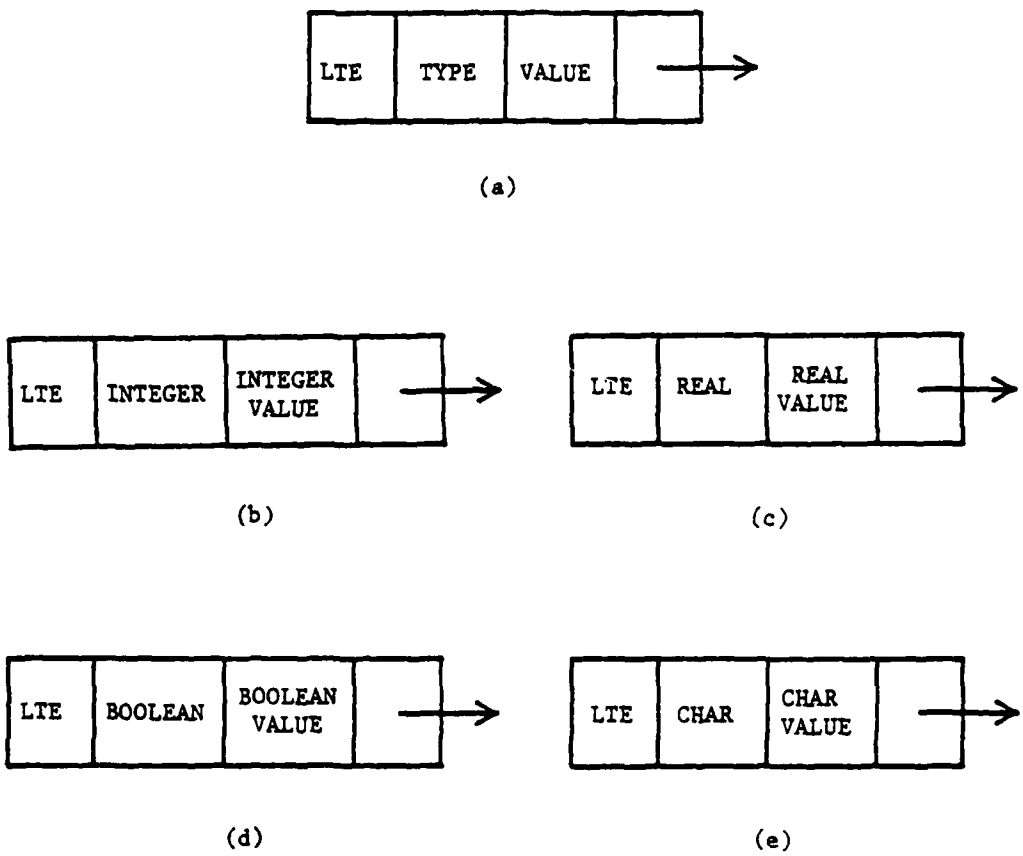



Figure 3.12: Association-node formats

### 3.4.3 Diagram of Record Skeleton

Figure 3.13 shows the structure of record skeleton and its relationship with the program skeleton. Again, let Rn, $n \geq 1$, represent the record contours; and let ADn denote the association node corresponding to the declaration node Dn in the program skeleton (refer to Figure 3.10). In Figure 3.13, the environment links of the record contours are shown with dash arrows, which are determined by the binding strategy employed.

The construction of the record skeleton is briefly described here; further details are provided in Subsection 3.8. 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.

## 3.5 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 that 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

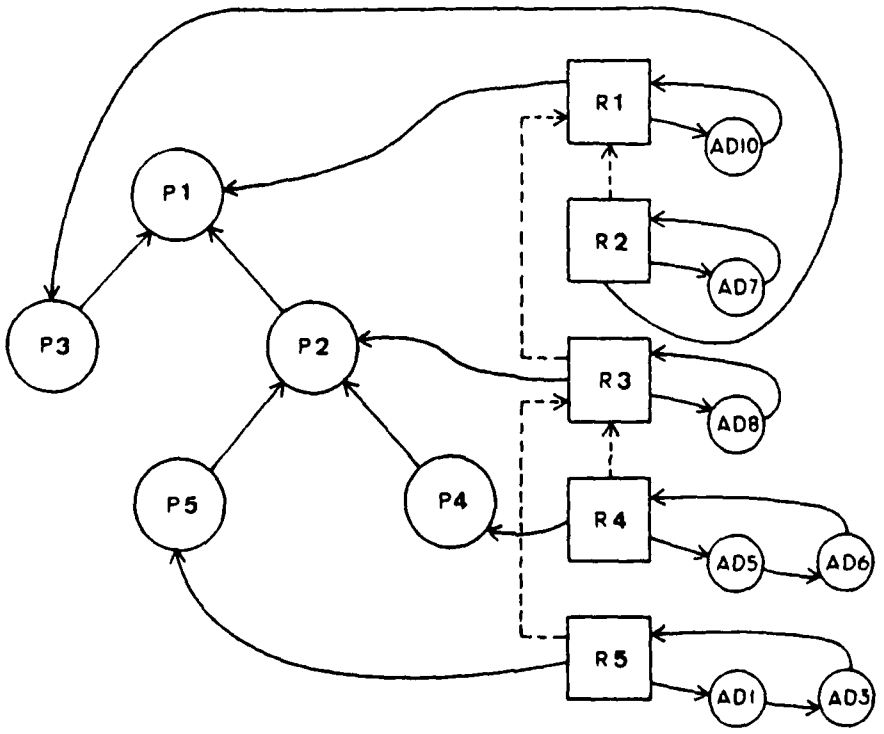



Figure 3.13: Record and its associated program skeleton

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 identifier 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.

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 programmer 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 [LIND81].

Figure 3.14 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 3.14, 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 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.

```
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 3.14: A Pascal program demonstrating the scoping rules

## 3.6 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 Subsection 3.8.

## 3.7 ENVIRONMENTS

In [JOHN73], an environment is described to be either the null sequence of contours, or consists of a sequence of contours $<C_0, C_1, \ldots, C_n>$, such that $n \geq 0$, the environment link of $C_n$ is null, and for $0 \leq i < n$, 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.

### 3.7.1 Environment Binding-Strategies (EBS)

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

Let E be the record environment, such that

$$E = \langle E_0, E_1, \ldots, E_n \rangle, \text{ where } n \geq 0:$$

1. The Dynamic Environment-Binding Strategy (DYN):

   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 $0 \leq i \leq n$

   (The symbol "$\uparrow$" means "a pointer to").

2. The Static Environment-Binding Strategy (STAT):

   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:

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

   where $0 \leq i \leq n$

### 3.7.2 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, <$C_0$, $C_1$, ... , $C_n$>) --> null or $\uparrow C_j$

### 3.7.3 Complete Binding-Strategies (CBS)

Two complete binding-strategies are used in PEEP. A complete binding-strategy, which consists of a search mechanism and an environment binding strategy, determines the binding method used in a language. Thus, CBS is a 2-tuple, consists of:

(RS, EBS)

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 record-contour search-mechanism is the absolute highest method, and the EBS is the dynamic environment binding-strategy.

   DYN_CBS = (AH, DYN)

2. Static Complete Binding-Strategy (STAT CBS)

The record-contour search-mechanism is the same as DYN_CBS, but the EBS is the static environment binding-strategy.

STAT_CBS = (AH, STAT)

## 3.8 EXAMPLES

The complete binding-strategy determines the environment links of the record contours. Figure 3.14 gave an example of a Pascal program which yielded 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.

Figure 3.15 shows the program skeleton of the Pascal program. 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

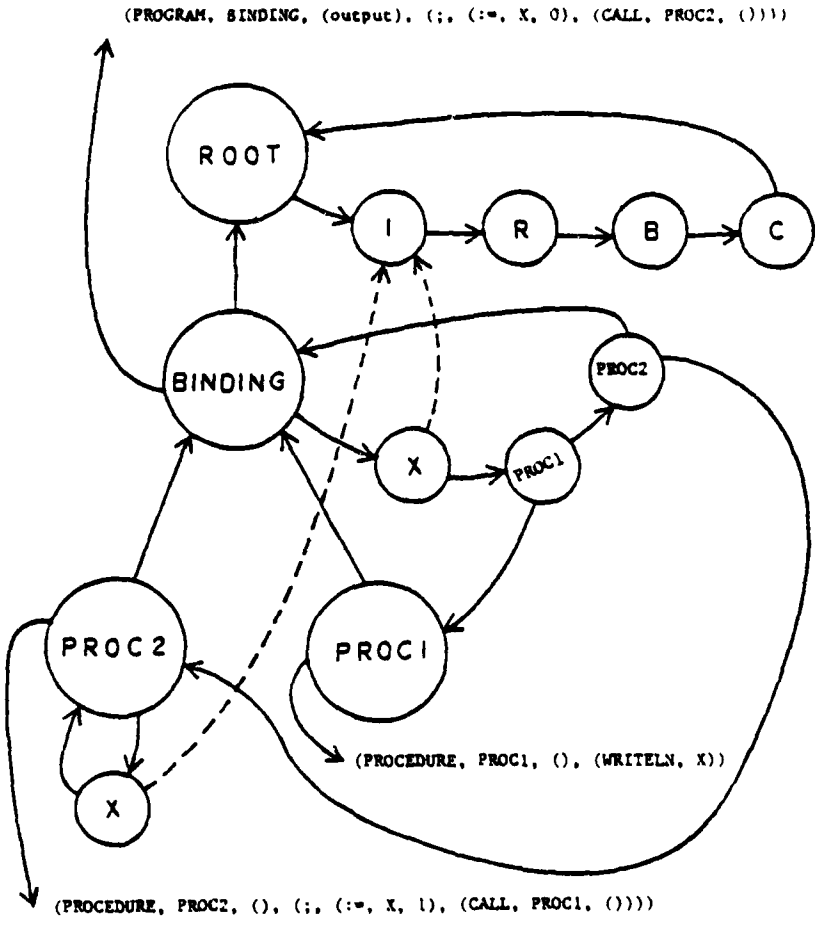



Figure 3.15: Program skeleton of Figure 3.14

to the program contour BINDING (Figure 3.16(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, and R2's antecedent link points to program-contour PROC2. Now, assume the STAT_CBS is used:

STAT_CBS = (AH, STAT)

The AH record-contour search-mechanism, with the identifier PROC2 and the current environment <R1> as the parameters, produces:

AH(PROC2, <R1>) --> ↑ 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( ↑ R1, <R1>) --> ↑ R1

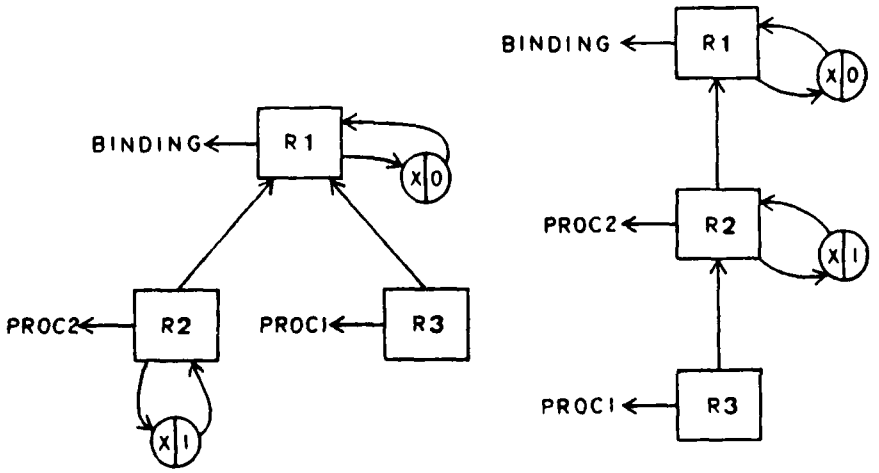



Figure 3.16: Record skeleton of Figure 3.14

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>) --> ↑R1

STAT(↑R1, <R2, R1>) --> ↑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 3.16 (b)). The following steps show why this is so.

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

AH(PROC2, <R1>) --> ↑R1

DYN( ↑R1, <R1>) --> ↑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, <R2, R1>) --> ↑R1

DYN( ↑R1, <R2, R1>) --> ↑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.

# 4. IMPLEMENTATION

The five modules described in the last section communicate with each other to constitute an interactive programming-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 mechanisms should be established among the processes. This can be accomplished by using event flags or semaphores.

The first version of PEEP used 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.

## 4.1 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}
```

## 4.2 EDITOR

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 report interested readers should see [EHRI81]. In order to use the editor SAM for this interactive programming-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}
```

## 4.3 TRANSLATOR

The components of the translator are shown in Figure 4.1. 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 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

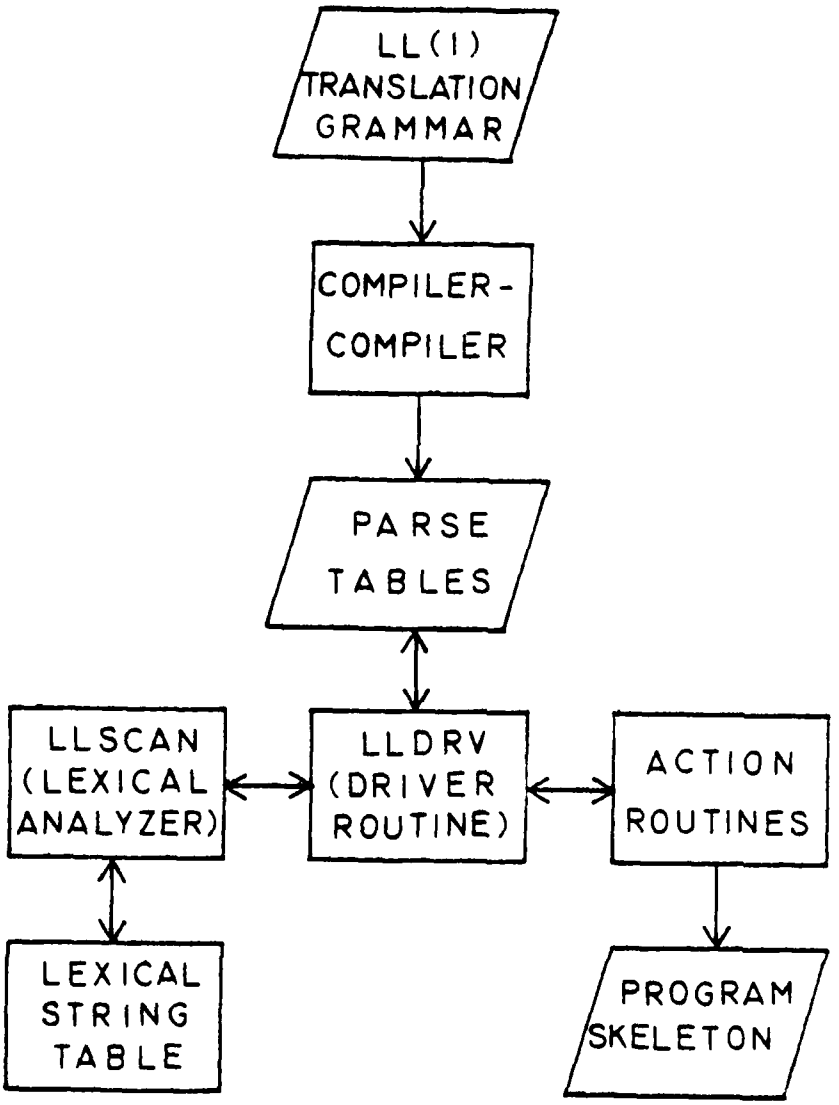



Figure 4.1: Components of the translator

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.

## 4.4 INTERPRETER

The interpreter is responsible for the building of the record skeleton. It simulates the working of the virtual processor described in Subsection 3.6. 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}
```

## 4.5 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.

## 4.6 FIRST VERSION OF PEEP

In addition to the algorithms described above, this subsection describes 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 was 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, messag-

es 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 Corporation.

```
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}
```

## 5. REFERENCES


[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 Programming Languages, Williamsburg, Virginia, January 26-28, 1981, pp.92-104.

[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.

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

[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.

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

[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.

[LIND81] 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.

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

[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.

[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.

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

OFFICE OF NAVAL RESEARCH

Code 442EP

TECHNICAL REPORTS DISTRIBUTION LIST

OSD

CAPT Paul R. Chatelier
Office of the Deputy Under Secretary
of Defense
OUSDRE (E&LS)
Pentagon, Room 3D129
Washington, DC 20301

Department of the Navy

Engineering Psychology Programs
Code 442
Office of Naval Research
800 North Quincy Street
Arlington, VA 22217

Communication & Computer Technology
Programs
Code 240
Office of Naval Research
800 North Quincy Street
Arlington, VA 22217

Manpower, Personnel and Training
Programs
Code 270
Office of Naval Research
800 North Quincy Street
Arlington, VA 22217

Information Sciences Division
Code 433
Office of Naval Research
800 North Quincy Street
Arlington, VA 22217

Physiology & Neuro Biology Programs
Code 441B
Office of Naval Research
Arlington, VA 22217

Special Assistant for Marine
Corps Matters
Code 100M
Office of Naval Research
800 North Quincy Street
Arlington, VA 22217

Department of the Navy

Commanding Officer
ONREAST Office
ATTN: Dr. J. Lester
Barnes Building
495 Summer Street
Boston, MA 02210

Commanding Officer
ONRWEST Office
ATTN: Dr. E. Gloye
1030 East Green Street
Pasadena, CA 91106

Office of Naval Research
Scientific Liaison Group
American Embassy, Room A-407
APO San Francisco, CA 96503

Director
Naval Research Laboratory
Technical Information Division
Code 2627
Washington, DC 20375

Dr. Michael Melich
Communications Sciences Division
Code 7500
Naval Research Laboratory
Washington, DC 20375

Dr. Louis Chmura
Code 7592
Naval Research Laboratory
Washington, DC 20375

Dr. Robert G. Smith
Office of the Chief of Naval
Operations, OP987H
Personnel Logistics Plans
Washington, DC 20350

Naval Training Equipment Center
ATTN: Technical Library
Orlando, FL 32813

Department of the Navy

Human Factors Department
Code N-71
Naval Training Equipment Center
Orlando, FL 32813

Dr. Alfred F. Smode
Training Analysis and Evaluation Group
Naval Training Equipment Center
Code TAEG
Orlando, FL 32813

CDR Norman F. Lane
Code N-7A
Naval Training Equipment Center
Orlando, FL 32813

Dr. A. L. Slafkosky
Scientific Advisor
Commandant of the Marine Corps.
Code RD-1
Washington, DC 20380

Naval Material Command
NAVMAT 0722 - Rm. 508
800 North Quincy Street
Arlington, VA 22217

Commander
Naval Air Systems Command
Human Factors Programs
NAVAIR 340F
Washington, DC 20361

Commander
Naval Air Systems Command
Crew Station Design,
NAVAIR 5313
Washington, DC 20361

Mr. Phillip Andrews
Naval Sea Systems Command
NAVSEA 0341
Washington, DC 20362

Commander
Naval Electronics Systems Command
Human Factors Engineering Branch
Code 81323
Washington, DC 20360

Department of the Navy

Dr. George Moeller
Human Factors Engineering Branch
Submarine Medical Research Lab.
Naval Submarine Base
Groton, CT 06340

Head
Aerospace Psychology Department
Code 15
Naval Aerospace Medical Research Lab.
Pensacola, FL 32508

Commanding Officer
Naval Health Research Center
San Diego, CA 92152

Dr. James McGrath
CINCLANT FLT HQS
Code 04E1
Norfolk, VA 23511

Navy Personnel Research and Development Center
Planning & Appraisal Division
San Diego, CA 92152

Dr. Robert Blanchard
Navy Personnel Research and Development Center
Command and Support Systems
San Diego, CA 92152

Mr. Stephen Merriman
Human Factors Engineering Division
Naval Air Development Center
Warminster, PA 18974

Mr. Jeffrey Grossman
Human Factors Branch
Code 3152
Naval Weapons Center
China Lake, CA 93555

Human Factors Engineering Branch
Code 1226
Pacific Missile Test Center
Point Mugu, CA 93042

Mr. J. Williams
Department of Environmental Sciences
U.S. Naval Academy
Annapolis, MD 21402

Department of the Navy

Dean of the Academic Departments
U.S. Naval Academy
Annapolis, MD 21402

CDR C. Hutchins
Code 55
Naval Postgraduate School
Monterey, CA 93940

Office of the Chief of Naval
Operations (OP-115)
Washington, DC 20350

Department of the Army

Mr. J. Barber
HQS, Department of the Army
DAPE-MBR
Washington, DC 20310

Technical Director
U.S. Army Research Institute
5001 Eisenhower Avenue
Alexandria, VA 22333

Director, Organizations and
Systems Research Laboratory
U.S. Army Research Institute
5001 Eisenhower Avenue
Alexandria, VA 22333

Technical Director
U.S. Army Human Engineering Labs.
Aberdeen Proving Ground, MD 21005

Department of the Air Force

U.S. Air Force Office of Scientific
Research
Life Sciences Directorate, NL
Bolling Air Force Base
Washington, DC 20332

Chief, Systems Engineering Branch
Human Engineering Division
USAF AMRL/HES
Wright-Patterson AFB, OH 45433

Dr. Earl Alluisi
Chief Scientist
AFHRL/CCN
Brooks AFB, TX 78235

Foreign Addressees

Dr. Kenneth Gardner
Applied Psychology Unit
Admiralty Marine Technology
Establishment
Teddington, Middlesex TW11 0LN
England

Director, Human Factors Wing
Defence & Civil Institute of
Environmental Medicine
Post Office Box 2000
Downsview, Ontario M3M 3B9
Canada

Dr. A. D. Baddeley
Director, Applied Psychology Unit
Medical Research Council
15 Chaucer Road
Cambridge, MA CB2 2EF
England

Prof. Brian Shackel
Department of Human Science
Loughborough University
Loughborough, Leics, LE11 3TU
England

Other Government Agencies

Defense Technical Information Center
Cameron Station, Bldg. 5
Alexandria, VA 22314 (12 copies)

Dr. Craig Fields
Director, System Sciences Office
Defense Advanced Research Projects
Agency
1400 Wilson Blvd.
Arlington, VA 22209

Other Organizations

Dr. Jesse Orlansky
Institute for Defense Analyses
1801 N. Beauregard Street
Alexandria, VA 22311

Dr. Robert T. Hennessy
NAS - National Research Council (COHF)
2101 Constitution Ave., N.W.
Washington, DC 20418

Other Organizations

Dr. Deborah Boehm-Davis
General Electric Company
Information Systems Programs
1755 Jefferson Davis Highway
Arlington, VA 22202

Mr. Edward M. Connelly
Performance Measurement
Associates, Inc.
410 Pine Street, S.E.
Suite 300
Vienna, VA 22180

Dr. Richard Pew
Bolt Beranek & Newman, Inc.
50 Moulton Street
Cambridge, MA 02238

Mr. Richard Main
ONR Resident Representative
George Washington University
2110 G. Street, N.W.
Washington, DC 20037

LCDR Stephen Harris, USN
HF Engineering Division
Naval Air Development Center
Warminster, PA 18974

Dr. J. Hopson
HF Engineering Division
Naval Air Development Center
Warminster, PA 18974

Dr. A. Meyrowitz
Code 433
Office of Naval Research
800 N. Quincy Street
Arlington, VA 22217

Dr. Thomas McAndrew
Code 32
Naval Undersea Systems Center
New London, CT 06320

Mr. Walter P. Warner
Code K0Z
Strategic Systems Department
Naval Surface Weapons Center
Dahlgren, VA 22448

Other Organizations

Mr. John Impagliazzo
Code 101
Newport Laboratory
Naval Underwater Systems Center
Newport, RI 02840

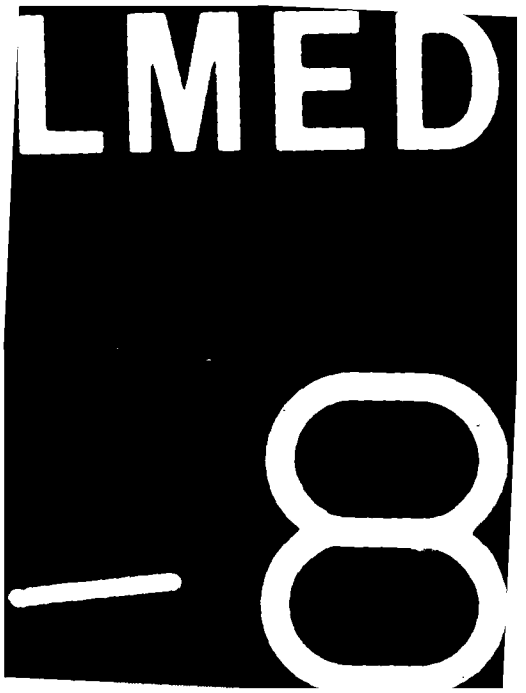
Dr. Mel C. Moy
Code 302
Naval Personnel R&D Center
San Diego, CA 92152

Dr. Richard Neetz
Pacific Missile Test Center
Code 1226
Pt. Mugu, CA 93042

Mr. Larry Olmstead
NSWC
Code N32
Dahlgren, VA 22448

Mr. Rick Miller
NSWC
Code N32
Dahlgren, VA 22448

Dr. Arthur Fisk
ATT Long Lines
12th Floor
229 W. Seventh St.
Cincinnati, OH 45202


LMED
8