Fall 1988

An integrated programming environment for pseudo-code development, IPE-PC

Nurcan Coskun

Department: Computer Science

AN INTEGRATED PROGRAMMING ENVIRONMENT FOR PSEUDO-CODE DEVELOPMENT, IPE-PC

BY

NURCAN COSKUN 1957-

A DISSERTATION

Presented to the Faculty of the Graduate School of the

UNIVERSITY OF MISSOURI - ROLLA

In Partial Fulfillment of the Requirements for the Degree

DOCTOR OF PHILOSOPHY

in

COMPUTER SCIENCE

1988

Thomas J Sager
Advisor

Billy E Gillett

C. Y. Ho

John B Prater

Paul D Stigall

ABSTRACT

An Integrated Programming Environment, IPE-PC, that supports pseudo-code development has been designed and implemented. This environment is based on a Pascal-like language which is designed according to the requirements of a language-based environment. The nucleus of IPE-PC is a language-based editor which represents programs as graphs internally. The same representation is used in every mode of the environment (i.e., editing, compilation, execution, debugging and translation). The system provides facilities to take advantage of both top-down and bottom-up programming. Stepwise refinement has been supported by providing comment structures that can be transformed into procedures. Bottom-up programming is supported because it is possible to create and save program segments which can be inserted to the programs at the appropriate points.

ACKNOWLEDGEMENTS

The author wishes to express her sincere thanks to Dr. Tom J. Sager for his supervision, guidance, counsel, and help in the completion of this dissertation. She would also like to thank her committee members Dr. Pete Ho, Dr. John B. Prater, Dr. Billy E. Gillett, and Dr. Paul D. Stigall for their interest and help.

TABLE OF CONTENTS

Page

ABSTRACT ii

ACKNOWLEDGEMENTS iii

LIST OF ILLUSTRATIONS vii

I. INTRODUCTION 1

II. LITERATURE SURVEY ON PROGRAMMING IN SMALL 4

A. USER INTERFACE 4

B. LANGUAGE BASED EDITORS 6

1. Template-driven Language Based Editors 7

2. Language Based Editors without Templates 8

3. Hybrid Design 8

C. INTERNAL REPRESENTATION 9

1. Text Representation 9

2. Token Oriented Approach 9

3. Tree 10

4. Graph 10

D. EXTERNAL REPRESENTATION 11

E. LEXICAL ANALYSIS 11

F. PARSING 12

G. INTERPRETER AND COMPILER 12

1. Interpreter 12

2. Compiler 13

H. DEBUGGER 14

I. ERROR HANDLING 15

J. UNPARSER AND PRETTY-PRINTER 16

K. DOCUMENTATION AND COMMENT HANDLING 16

L. TOP-DOWN VS. BOTTOM-UP PROGRAM CONSTRUCTION .. 17

M. LANGUAGE 18
N. RELATED WORK 19
O. PROGRAMMING IN THE LARGE 23
III. DESIGN AND IMPLEMENTATION OF IPE-PC 24
A. LANGUAGE DESIGN 24
1. Motivation 24
2. Control Structures 26
a. Sequence 27
b. Comments 27
c. IF Statement 29
d. IF-ELSE IF-ELSE Structure 30
e. Iteration 32
3. Procedures 35
a. Input Parameters 35
b. Output Parameters 35
c. Access to Nonlocal Names 35
4. Views 37
a. Local Views 37
b. Global Views 38
5. Input and Output Facilities 38
B. INTERNAL REPRESENTATION 39
1. Type-List 40
2. Program-List 42
3. Mapping Pointers 44
C. LANGUAGE BASED EDITOR 45
1. Templates 45
a. Algorithm Window Templates 45
b. Variable Window Templates 58
c. Type Window Templates 58
2. User Interface 60

a. Top Window 60
b. Message Window 63
c. Menu Window 63
D. EXECUTION AND DEBUGGING 64
E. EXTERNAL REPRESENTATION 65
F. RUN TIME ENVIRONMENT 66
G. MULTIPLE LANGUAGE SUPPORT 68
H. PORTABILITY 68
IV. CONCLUSION AND FUTURE DIRECTIONS 69
BIBLIOGRAPHY 71
VITA 79

APPENDICES

A. GRAMMAR 80
B. USER MANUAL 84
C. IPE-PC LISTING FILES 93
D. PASCAL PROGRAMS 110
E. EXTERNAL REPRESENTATION 123
F. INTERACTIVE INPUT - OUTPUT FILE 160

LIST OF ILLUSTRATIONS

Figure		Page
1	A Sequence Control Structure	27
2	An IF-ELSE Logical Control Structure	29
3	An IF-ELSE IF-ELSE Logical Control Structure	31
4	A WHILE Logical Control Structure	32
5	An UNTIL Logical Control Structure	33
6	A FOR Logical Control Structure	34
7	A Start Node of Program Graph	39
8	A Type Node in the Type List	40
9	An Array Descriptor	41
10	A Field Descriptor	41
11	A Procedure or Program Node	44
12	A Tree Node	45

I. INTRODUCTION

The main purpose of programming is to automate the solution of time consuming problems. Especially if those problems have a repetitive nature, solving the problem algorithmically, will solve it, hopefully, for all possible cases.

Computers are becoming a very important necessity for not only high-tech oriented applications, but also for other type of less skill oriented applications (like word processing, file processing etc.). It seems people with a variety of skills have to use computers in some form. In other words, programming is becoming a skill as important as basic mathematics and reading skills.

Programming is a difficult problem itself. It has a very repetitive nature, all programmers go through similar steps. It is a difficult job for most people. Its difficulty and repetitiveness suggests that it is a good candidate for automation. If we can automate the programming process successfully, this would help not only pure computer scientists, but also millions of potential users.

The importance and necessity of this automation has been understood by the research community in the last 10 years[50]. Quite a few groups are doing research in this area. Although there are a few experimental systems that are used in a few universities and research labs[16,18,26,27,39,59,67,69], most of us are still living with the old fashioned tool-kit approach. In other words, we are using independently implemented editors, debuggers, compilers and interpreters managed by the operating systems. All of these tools have different interfaces, and switching from one mode to another is not always natural. The representation of programs in various modes is different : text in editors, machine code in the O.S., a tree representation in compilers etc. . These representational differences naturally create different requirements in different applications.

Recent research has shown that it is possible to use the same program representation for editing, compilation, debugging and interpretation[67]. The uniformity in the representation also brings uniformity in the implementation and user interface. It seems, when we integrate these

tools, it is possible to create a coherent environment. This is not surprising, since these tools are logically related (they are all designed for the solution of the problems in the computer).

The reason that people came up with independently implemented tools in the first place, was the difficulty of these problems (design of system programs) and the lack of understanding in their solutions. In the last 20 years, it was convenient to concentrate on these tools separately for the sake of problem simplification. At this point in the development of our science, we have enough understanding of them to be able to integrate or design them as a whole unit. As a matter of fact, recent research is a sign of this.

Programming environment research can be grouped into two major categories[10]:

1) Programming in the large.

2) Programming in the small.

The research on "programming in the large" is concerned with the efforts of groups of people to develop large software systems with long intended lifetimes. On the other hand, "programming in the small" is concerned with the efforts of individual programmers to develop moderate-sized programs (a few thousand lines) within a short period of time (a few months). Both categories require large amounts of diverse knowledge[29,36,42,70].

The ultimate goal of this kind of research, is to design a programming environment that can create a comfortable environment for different levels of skills and requirements. The environment should be designed to hide the unnecessary features (for that particular user) and suggest a simple subset when it is needed.

The main objective of this research is to explore the design and implementation issues in the "programming in the small" area. An integrated programming environment that supports structured programming was designed and implemented for this purpose. This environment supports both pseudo-code and source code development. The psuedo-code is created by using stepwise refinement. When the design of the pseudo code is complete, it can be translated into Pascal[33].

This environment was designed in an integrated way: it will edit, compile, debug, interpret and translate programs using an integrated tool. The passage from one mode to another is very natural, because all of them use the same program representation.

II. LITERATURE SURVEY ON PROGRAMMING IN SMALL

The tools and techniques which are used in the design and implementation of the recently developed integrated programming environments will be described below to set up the background in the field. The problem areas are also discussed for the different aspects of environment design.

A. USER INTERFACE

Human factors are very important issues that have to be considered in the design of integrated programming environments. The environment must be designed to provide a user-friendly interface. The features that are provided in the environment must be designed in a uniform and consistent way with each other. The major issue in the design of integrated programming environments is to provide integrated facilities for the program development activities. These integrated facilities should look like different features of a whole. In other words, the transitions between the different modes, such as editing, execution, and debugging must be smooth. The command language should be easy to use, and easy to learn.

A menu-driven approach is used to issue commands in most of the recently developed integrated programming environments. The menu is usually displayed according to the current context. Only a subset of the menu is valid for different modes and views.

Menu commands can be issued by using function, arrow and other keys of the keyboard. It is also possible to use a pointing device (i.e., a mouse) to point to the menu item that is needed. This is especially desirable if the required string for that command is more than one character.

It is also possible to use high level programming languages as a command language. In this case, it is possible to create user tailored complicated commands. In comparison to menu-driven commands, it is harder to develop such an interface, although it looks more powerful. Pascal has been used as a subset of the debug command language in the DICE[24] and MAGPIE[20] systems . Usually the high level command language is the same language that is

used to develop programs in the environment, this provides a uniform treatment of the facilities in the environment.

Interactive assistance should be provided in an integrated programming environment to provide a user-friendly interface. Even if there is a menu system, the user has to know what the menu entries mean. A "help" facility may provide additional assistance for the explanation of menu entries as in GNOME[26].

It is desirable to have a program view larger than 24 lines of text which is provided in most of the terminals that are used today. It is very difficult to use overlapping and separate windows on this type of screen because of limited space. Comments are treated as the description of refinement steps in some environments[67]. In these environments, the executable statements and/or other refinement steps that belong to a comment/refinement step are indented to show the control structure of the program. Holophrasting[75], hiding the internal parts of comments to save space and improve readibility of the whole structure of the program, has been used as in SUPPORT[75] and SED[3] to improve the user interface.

It is possible to create different views of the program to speed up the program development in an integrated programming development environment. For example, one window may show the names of the procedures, their descriptions, their interfaces and other windows may be used for variable, constant, and type declarations. The algorithmic content of the program may be displayed in the form of a flow chart as well as text. More than one window can be used to display different segments of the different procedures at the same time. During the execution, the run time stack as well as the contents of the variables can be displayed on a separate window for debugging purposes. Each of these different views provides important information to support program development activities. It is especially desirable to have more than one view on the screen at the same time. In programming environments, such as PECAN[53] and MAGPIE[20] , high resolution bitmapped displays are used to provide multiple views at the same time.

B. LANGUAGE BASED EDITORS

A language based editor, LBE, is an editor that understands the syntax and semantic rules of the corresponding language. In comparison to the traditional text editors, LBE's are harder to implement because they attempt to understand the meaning of their input. In other words, the CPU and memory requirements of LBE's are higher than those of text editors. Because of the constant decline in the hardware prices, it is feasible now to implement such editors for some programming languages.

Language based editors are designed to develop programs in the language that they support. Therefore, they are special purpose editors tailored according to the requirements of their language. Since a LBE understands the structure of the program, it can display the programs in a formatted way by using automatic indentation.

A LBE can diagnose the syntax and semantic errors immediately and inform the user about them by displaying error diagnostic messages, and corrective actions. Some LBE's even attempt to recover from those errors as in COPE[6]. Therefore, it is easier to learn programming in a language based environment. One does not have to remember all the rules of the language to write correct programs. LBE's have been used successfully in educational environments[26,67].

LBE's can support good documentation, good programming styles and reduce typing efforts. Therefore it increases the productivity of the programmers.

A LBE also forms the nucleus of an integrated set of programming tools to support all phases of program development. The usage of LBE's in the design of integrated programming environments is very popular[17,20,22,26,67].

The design of language based editors can be analyzed in three categories:

1. Template-driven Language Based Editors.

A template is a predefined, formatted pattern of keywords, punctuation marks and placeholders. Each placeholder designates the syntactic class of permissible insertions[67].

Templates resemble the productions of a grammar. In a template-driven editor, there is almost a one-to-one correspondence between templates and productions in the grammar. Templates can be used to construct programs in a top-down fashion similar to the construction of a derivation tree for a string in a language.

Each template corresponds to one production in the grammar, its expansion consists of a sequence of formatted terminal and nonterminal symbols. The terminals are keywords and punctuation symbols. Nonterminals are placeholders, which can be expanded by either other templates or terminals, i. e., identifiers.

Template driven editors enforce a top-down construction of the program. The templates can be expanded in a consistent way with the corresponding grammar rules. Therefore, it is impossible to violate syntax rules of the grammar. Since the keywords, punctuation symbols and indentation are provided automatically, typing effort is reduced.

The usage of templates seems appropriate for most of the control structures, such as IF, WHILE, and UNTIL. On the other hand, it might be very tedious for expressions. Therefore, this pure template-driven approach is relaxed to favor expression entry in the text mode in some systems[67]. It is possible to have syntax violations in the expressions in such systems.

All the edit operations in a template-driven editor are in terms of templates. The program is no longer considered as a text. Therefore, some modifications, that seem very easy to do in text editors, can be a difficult task in a syntax driven environment[67]. A typical example of this problem is to convert an existing WHILE statement to an IF statement as in the following Pascal segment.

```
WHILE (Score > = 0) DO                    IF (Score > = 0) THEN
  BEGIN                        ?            BEGIN
    NumScores: = NumScores + 1;   -----> NumScores: = NumScores + 1;
    Sum: = Sum + Score                     Sum: = Sum + Score
  END                                      END
```

In the text editor all we have to do is to replace WHILE keyword with the IF keyword and replace DO by THEN keyword after the expression. In a template-driven editor there are two possibilities:

1) Provide a special purpose command for this modification. Then we should be able to enumerate all such possibilities and design the command language accordingly.

2) Use two clip operations to save a copy of expression and statement parts, then insert a WHILE template and expand the expression and statement parts by using the clipped information. Finally, delete the IF statement.

2. Language Based Editors without Templates.

Although template driven editors work well for the novice programmers, experienced programmers may find the usage of the templates tedious. Also, strictly template-driven systems can not accept programs created outside of that environment. It is desirable to design a system that can also accept the programs in the form of text files. So, in this group of environments[46] the programs can be entered in the traditional way. However, the editor still knows the rules of the language, so the programmer is informed about the violations right away. It is harder to enforce syntax and semantic correctness in this type of environment. Incremental compilation is another issue that is harder to tackle in this type of approach.

3. Hybrid Design.

This group of environments[31] provide a hybrid form of editor. The user can freely switch between the template and nontemplate editor modes. So, it is possible to take advantage of both approaches. However, it seems that multiple entry modes may bring confusion for the novice user, and also increases the complexity of the implementation of the system.

C. INTERNAL REPRESENTATION

Internal representation refers to the data structures that are used to represent the programs when they are developed. Internal representation can be discussed in four major categories:

1. Text Representation.

In this approach, the programs are represented as a sequence of characters. This type of view provides the ability to move the cursor to anywhere on the screen without any restriction. So, it is possible to issue editor commands, such as INSERT, DELETE, REPLACE, .. etc., anywhere on the screen. This provides a uniform view of the program as a sequence of characters.

Pure text representation is not appropriate for LBE's unless the editor is designed to develop short programs. Sometimes, text representation is used to speed up display of the programs in LBE's which contains another data structure like a parse tree to represent programs. This implies a space overhead, but faster execution. Morris, et al[46] designed such an editor.

2. Token Oriented Approach.

When the programs are entered, lexical analysis is applied and tokens are formed. A linked list of tokens is the data structure that is used to represent the programs. This type of representation is more appropriate for language based editors than text representation. There is no need to reapply the lexical analysis on the program once it is scanned unless it is modified, this speeds up parsing in comparison to the pure text representation. On the other hand, it is difficult to decide when and how to reparse the program, and how to support incremental parsing.

This type of representation is not very popular among LBE designers. Brun, et al[14] implemented a token oriented editor for the PORTAL language on VAX/VMS.

3. Tree.

Syntax trees and parse trees have extensively been used by LBE designers[31,67,75].

A parse tree[1] (concrete syntax tree) shows how the start symbol of a grammar (concrete syntax) generates a string in a language. Therefore, parse trees are language dependent internal representations. The syntactic sugar (keywords, punctuation symbols, .. etc.) are on the tree as well as operands and operators. This will speed up unparsing to produce a text image of the program, but it is not space efficient.

In an abstract syntax tree[1] (syntax tree), each node represents an operator and the children of the node represents the operands. Abstract syntax trees differ from parse trees because superficial distinctions of form do not appear in syntax trees. It is easier to translate this language independent representation to produce programs in different programming languages. A text image for the tree can be generated by using tables or special print procedures.

4. Graph.

A graph data structure is necessary if we want to keep information about data flow as well as control flow. It is possible to produce optimized code from this representation. It is also useful to detect data flow anomalies and produce useful information for debugging. The programs can be interpreted directly by a data flow machine if this type of representation is used as in Ottenstain, et. al[51]. Graphs have been used as major data structures in the design of the incremental programming support environment, IPSEN[48].

D. EXTERNAL REPRESENTATION

The program can be saved either as text or in a form which helps the construction of internal representation easily. If the programs are saved as text, then the internal representation has to be constructed out of the text when the edit session starts unless the internal representation is text. This may slow down the response time of the system. On the other hand this implies that the environment is able to process programs which are created outside of the environment as well. If the program is saved in a form to speed up the construction of internal representation, the response time will be faster when the edit session starts.

Some systems save both text and coded form of internal representation if they are using text to speed up the display of the programs.

Even if the text is not required for internal manipulations, it may be desirable to be able to construct internal representations out of the text to support programs created outside of the environment.

E. LEXICAL ANALYSIS

Depending on the internal representation, the usage frequency of lexical analysis changes. If it is a strictly template driven editor, lexical analysis is only needed for the terminal tokens such as identifiers and constants, and there is no need to form tokens for keywords, punctuation symbols, and operators. If it is a partially template driven editor, then lexical analyzer may produce a series of tokens for each text entry corresponding to the expansion of expressions and assignment statements. In a nontemplate language based editor, lexical analysis has to be applied to the input program to build the parse tree.

F. PARSING

Parsing in the language based environments depends on the internal representation of the programs. In a strictly template-driven approach, parsing disappears. Because, the parse tree is constructed top-down when the templates are expanded. In a partially template-driven editor, the parser only takes care of the expressions and/or assignment statements. A simple parser is sufficient for this purpose. In the nontemplate approach, a parsing algorithm[1] suitable to the language grammar has to be used to construct the parse tree.

Some language based environments use a hybrid approach for editor design. They let the user modify segments of the program by using text editor commands. In this case, multiple entry parsers[31] can be used to parse the program segments that are modified on the text window. In the multiple entry parser, the starting symbol of the derivation is not restricted to the starting symbol of the grammar. Any nonterminal symbol of the grammar can be used as the starting symbol for the analysis of a program segment. The subtree that is produced for the text segment, can then be inserted as the expansion of a placeholder in one of the program templates. Of course, the root of the subtree must be consistent with the type of expansion required for that placeholder.

G. INTERPRETER AND COMPILER

1. Interpreter.

Most of the integrated programming environments use an interpreter[17,53,67]. Since these interpreters use the internal representation to execute the programs, the speed of the execution depends on the suitability of the internal structure. Obviously, if the program is kept as a text internally, the interpretation would be very slow. On the other hand, internal representations such as trees and graphs can be interpreted rather fast, because the program is already tokenized and the parse tree is constructed. In the pure template driven editors, a preorder traversal of the syntax tree will be the basis of the interpretation. In the partially template driven editors, the object code for the expression and assignment statements can be produced incrementally during program development, so no translation is needed before the execution starts.

Interpreters can be used easily to produce useful debugging information during the execution, because the operands in the object code are pointers to the symbol table. Therefore, the contents of the variables can be displayed easily without any overhead. Since the interpretation is done in a syntax driven way, it is also easy to stop and/or pause the execution at certain points in the program.

2. Compiler.

Compilers can also be used to execute programs in the language based environments[20,24]. Programs can be translated either by using the internal representation or by creating a source file which is compiled by the compilers outside of the environment. In the later case, it is harder to implement efficient debugging actions. One way is to extend the source programming language to define the debugging actions[20]. Another way is to insert "debugging segments" into the program code according to the directions of the system commands[24].

Obviously, the execution will be faster for the compiled programs, which could be a plus for the development and testing of large programs.

On the other hand, even if the program is large, the modules can be developed and tested by using interpretation in a bottom-up way. Once the modules are tested they can be compiled together to test the program as a whole. If the environment supports the development of large programs, incremental compilation (at the module level) can be used to speed up the compilation of the program.

A hybrid approach is best suited for practical purposes. During the program development, an interpreter provides enormous advantages for the debugging over the compiler. Once the program is developed and tested, it can be compiled by using optimizer compilers to produce efficient object code that can be used in the production environments.

H. DEBUGGER

The use of interpreters provides powerful debugging facilities. In most of the integrated programming environments, the same internal structure is used to support the editor, interpreter, and debugger. Therefore visual feedback about the execution flow can be provided very easily by displaying the program on the screen and by moving the cursor to the current statement that is being executed or by highlighting the corresponding area. This feature is called "tracing"[67]. The speed of the "tracing" facility can be adjusted during execution by using debugger commands. It is also possible to switch to a "single-stepping"[67] mode. In "single-stepping" mode, the user may control the execution speed manually. One statement at a time can be executed and contents of the variables can be checked. Another facility called "reverse execution"[67] is a very interesting facility used experimentally by some environment designers[67]. Reverse execution has been provided for the purposes of finding the exact location of a bug. Reverse execution is simulated by keeping the effects of the assignment statements on a stack and then restoring those overwritten values back to their locations to cancel the effect of the current assignment statement. So, the programmer can locate the exact location of the bug by interleaving the reverse and forward execution.

Although this "visual feedback" provides helpful information to the programmer, this may slow down the execution speed. Therefore, a classical interpreter mode without any visual feedback may be provided. Before the execution starts in the classical mode, the programmer may identify the stop points by using commands like "stop at procedure X", or "stop at line Y in procedure X" as in DBX[77], debugger of UNIX. When the execution flow reaches that point, execution may continue in single stepping mode by giving visual feedback.

The "tracing", "single-stepping", "reverse execution", and "classic execution" should be interleaved smoothly for the successful debugging of programs.

It is also possible to interleave the execution with the edit operations as in Cornell Synthesizer[67]. Once the execution is suspended, the program can be edited by using the usual edit operations. The program execution can be continued if and only if those edit operations did not create an inconsistency between the program state and activation stack. If an edit operation

changes the type of a variable, for example, the execution may not continue with the current activation stack.

The debugging facilities that are described above can also be implemented in the environments that use compilers. But the implementation will be much harder, and the overhead may slow down the response time of the system considerably.

I. ERROR HANDLING

Most of the language based environments produce only diagnostic messages about the errors. Syntax errors can be detected very easily by the language based editors. It is harder to implement full static semantic checking. To speed up the response time of the environment, attributed syntax trees are used for type checking. Every node in the tree carries a type attribute. When there is a change to the type of an identifier, the type attributes of the nodes that refer to this identifier (these nodes can be found easily by using reference lists in the symbol table) are reevaluated. Therefore, it is always possible to find all the statements with the syntax and/or semantic errors. These statements can be highlighted to give a visual feedback to the programmer. When the cursor is moved to an erroneous statement, the corresponding error message can be generated by using the value of the attribute on the node.

Some language based environments attempt to repair the syntax errors as well as informing about them. The repair may not always be the one which is intended by the user. Therefore it is essential to provide an "undo" mechanism to eliminate the effect of the repair. COPE[6] is an example of an integrated programming environment which includes an extensive error repair facility. The user can enter approximate forms of the source program segments. The error repair mechanism attempts to form a valid program construct out of this incomplete description. Although it may be desirable to have this type of facility, it is very difficult to implement for a reasonable sized system.

Language based environments can also detect run time errors successfully. Since most of them use interpreters, the locations of the operands are found through the symbol table during the execution. For example, it is very easy to check a reference beyond the array limits. When

there is a run time error, the execution can be suspended or stopped by giving a visual feedback about the exact location of the error.

J. UNPARSER AND PRETTY-PRINTER

Most of the language based environments represent the programs as an abstract syntax tree internally. To produce a text image out of this tree, it has to be unparsed. Abstract syntax trees can be unparsed easily by using a preorder traversal. If the nodes on the tree do not carry information about the concrete syntax (keywords, punctuation symbols, indentation), then tables or special print procedures can be used to produce this information. If the concrete syntax is kept on the nodes, it will be unparsed faster but this is not space efficient.

Template driven editors always force the programs to be pretty-printed. The format (indentation) of each template is part of its definition. So, when a placeholder is expanded, the template chosen is printed in an indented fashion.

In the nontemplate type editors, once a program segment is entered, it is reprinted by following the corresponding indentation rules. Since nontemplate editors can not enforce the syntax correctness, it is harder to pretty-print the program after a syntax error.

The indentation amount that is used by the pretty-printer can be adjusted by the programmer if the environment contains commands to adjust this parameter.

K. DOCUMENTATION AND COMMENT HANDLING

Comments can be handled in two different ways in a language based environment.

1) Comments can be used in exactly the same way the corresponding programming language uses them. In this case, the source language defines the rules about how and where to create the comments.

2) The source language can be modified slightly to treat comments as the descriptions of the refinement steps[67]. In this case, the executable statements and/or other refinement steps that belong to this refinement step are indented to show the control structure of the program. Therefore, comments are treated as special nodes in the abstract syntax tree. It has

been found that this approach not only increases the readability of the programs but also these descriptions make up the documentation about the logic of the program. The display of the refinement of a comment can be suppressed by using special commands to present more information about the program logic in a small area as in Cornell Synthesizer[67].

L. TOP-DOWN VS. BOTTOM-UP PROGRAM CONSTRUCTION

Most of the integrated programming environments support top-down programming[40,57,58,61,67]. It is also possible to support bottom-up program construction in a language based environment as in reference 32. The programmer can create program segments and name them. These segments can be kept in a library and can be copied and modified any time. Finally, these program segments can be arranged to form procedures and programs.

A hybrid form gives advantages of each approach for program construction. In programming environments like Cornell Synthesizer, the program segments can be clipped and named. Then, these segments can be inserted at the appropriate places in the program.

Libraries for common algorithmic segments can be kept globally as well as locally. In other words, the environment may provide a library that contains such segments. The programmer may borrow any of them when they are needed. It is harder to implement a global "common algorithmic segment" library. First of all, the segments have to be identified and implemented. Second, the information retrieval from this library should be fast enough to represent a comfortable interactive environment for the programmer. Also the programmer should be educated about the availability of such segments, or about how to find the segments by using an approximate description about them (rather than an exact name). It may be impossible for the programmer to remember all the names of algorithmic segments in such a library.

It is desirable to have a library that implements all the operations of abstract data structures (i. e., linked list, stack, queue, tree, graph). The system should be able to produce code for different concrete data structures. For example, it should be able to use the same search

segment to search a linked list as well as an array. Although there is an increasing interest on this area in the field, satisfactory results have not been produced yet[29,70].

A satisfactory implementation of such a global library may increase programmer productivity tremendously. The programmer does not have to reinvent the same segments again.

M. LANGUAGE

State of the art integrated programming environments have been developed for the languages with simple syntax like LISP and BASIC. Because of the rapid developments in the hardware technology, computers are becoming cheaper and more powerful everyday. Therefore, it is possible now to implement such environments for other programming languages. Prototype environments for programming languages such as ADA, PASCAL, C, FORTRAN, and PL/I have been implemented. All of these environments are tailored according to the requirements of a particular language. It is only possible to write programs in one language in these environments.

The internal structures like trees and graphs can be used to create a language independent environment. The language specific information of the environment can be isolated easily in the tables, then this information can be retrieved for the display and other language specific purposes. Some of the example environments that are designed to generate language specific environments are discussed in references 17, 22, 26, 53 and 55.

The internal structures like trees and graphs are also suitable for translation purposes. Generation of source programs in different programming languages from such internal structures is fairly easy. A few systems like ALBE/LNF[41] and Schemacode[58] experiment with this idea.

Since integrated programming environments are language specific, they have been found very useful to teach programming languages in educational environments. Because they have superior programming support facilities, they have also been designed to be used in research and development environments.

The next section summarizes some of the major programming environment research and discusses the usage of them in educational and research environments, as well as their features to support multiple programming languages.

N. RELATED WORK

Some of the state of the art integrated programming environments will be discussed below. Although most of these environments are prototypes and are being used to explore the main issues in the field, some of them have been used successfully in educational and research environments.

The Cornell Synthesizer[67] is one of the earliest integrated programming environments developed for a block-structured programming language. The programming language implemented for the Synthesizer is PL/CS, an instructional dialect of PL/I. It is operational under UNIX as well as on Terak(LSI-11) microcomputers. The Synthesizer has been adopted for elementary programming instruction at Cornell University, Rutgers University, Princeton University, and Hamilton College.

Another project[55] at Cornell University concerns the development of a language-independent system for generating Synthesizer-like systems from a grammatical specification of a given programming language. An attribute grammar is used to define the syntax, display format, and semantics of each template and phrase.

An Interactive Software Development Environment, ISDE[17], was implemented in Berkeley Pascal on UNIX. It is possible to derive an environment for a specific language as an instantiation of a language-independent meta-environment in ISDE. Environments for the subsets of ADA, PASCAL, MODULA-2, and GALILEO have been partially generated.

The SUPPORT[75] environment,The Still Unnamed Production Programming Oriented Research Tool Environment, was implemented in PASCAL for a VAX 11/780 under Berkeley UNIX. It also runs on a Sun Microsystems workstation and on an IBM PC computer. It provides an integrated program development and execution environment for a subset of PASCAL.

POE [22], Pascal Oriented Editor, is a full-screen language based editor, which is implemented as a first step in development of a comprehensive Pascal program development environment. It is operational on VAX 11's under Berkeley UNIX and on HP 9800 series personal workstations. Poe is written in Pascal, and it is designed to be readily transportable to new machines. HP 9800 version of Poe contains 27,000 source lines and requires about 270K bytes of main memory. An editor-generating system called POEGEN is also implemeted by the same group.

The Distributed Incremental Compiling Environment, DICE[24], is a highly integrated programming environment which provides programmer support in the case where the programming environment resides in a host computer and the program is running on a target computer that is connected to the host. DICE is implemeted in 20,000 lines of INTERLISP on a DEC20 computer. Its incremental compiler accepts almost full Pascal including input-output facilities.

Gandalf Novice Programming Environment, GNOME[26], is used to teach programming to undergraduates at Carnegie-Mellon University. It runs on VAX 11/780's under UNIX. GNOME Pascal editor implements standard Pascal, and uses Berkeley Pascal interpreter to generate code and do the rest of the semantic checking. GNOME environment has 3 other syntax-directed editors : a family tree editor, a Karel editor, and a Fortran editor that are used for educational purposes. ALOE editor generator has been used to develop this family of structured editors.

MAGPIE[20] is an interactive programming environment for Pascal. MAGPIE is implemeted on an engineering workstation that was developed as a research tool within the Computer Research Laboratory at Textronix. It consists of more than 40,000 lines of Pascal code and about two thousand lines of C and assembly language code.

Arcturus[62], a Prototype Advanced Ada Programming Environment, is another system that explores key programming environment issues. It offers an approach to the integrated use of compiled and interpreted Ada. Arcturus has been used with considerable success in the compiler classes at University of California, Irvine.

SYNED[25,31] is the editor component of an interactive programming environment under development at Bell Labs. It is designed to accommodate the needs of professional programmers at Bell Labs. Syned accepts almost full C language and runs under UNIX.

PECAN[53] is a program development system generator for algebraic programming languages and has been developed at Brown University. An important objective of PECAN project is to investigate ways of making full use of the computing power and graphics available on the new generation personal computers. It is implemented on APOLLO workstations.

COPE[6], a Cooperative Programming Environment, is developed at Cornell University. It implements PL/C, and provides automatic error-repair. It is written in C and runs under UNIX on a VAX.

The ALICE[68] Pascal system was developed by Looking Glass Software for the ICON computer and other 16-bit machines. It was created for the Ontario Ministry of Education and it is one of the first syntax-directed editors to go beyond the prototype stage and the full educational software market. ALICE has been tested with high school students in the Waterloo region.

An Incremental Programming Environment, IPE[45], is based on compilation technology, but provides facilities traditionally found only on interpretive systems. The IPE prototype runs under UNIX on a DEC/VAX. This environment also contains an editor generator.

PASES[61] is an interactive programming environment that supports the creation of Pascal programs.

A program development system, SIMPLE[16], supports the development of Pascal programs. SIMPLE is implemented on a PDP 11/34, under RSX-11M operating system; the output programs are compiled and run on a Univac 1100/80.

PASLAB[27] is a computer learning package to develop programs in Pascal. It has been used to teach programming courses at Worcester Polytechnic. PASLAB allows student to understand what is happening inside the computer relative to statements in Pascal programs constructed by an expert.

PMS[69], A Program to Make Learning Pascal Easier, is also developed to be used in an educational environment. PMS is organized as a collection of "minilanguages" each of which demonstrates and allows the user to experiment with, a certain category of Pascal features. PMS is written in Pascal and runs on several computers including the IBM PC. It has been used to teach programming courses at Acadia University.

ALBE/P[41] is a language-based editor for Pascal. ALBE/P has been modified as ALBE/LNF in which programs can be entered in Language-Neutral Form (LNF), a Pascal subset with language features common to C, PL/I, and ALGOL. When the program development is complete, ALBE/LNF can generate equivalent programs for the programming languages Pascal, C and Fortran. ALBE is implemented in VAX/VMS.

A Conversational Algol system, CONA[7], is an environment for Algol-60 language. It has been used in introductory programming courses at University of Sheffield. CONA was implemented in Algol 60 and runs under Maximop.

CAPS[73] is a highly interactive diagnostic compiler and interpreter that allows beginning programmers to prepare, debug, and execute fairly simple programs at a graphics display terminal. It has been used to teach Fortran, PL/I and Cobol in the programming courses at University of Illinois at Urbana-Champaign.

The Lisp Tutor[4] is designed to teach LISP at Carnegie-Mellon University. It is a large LISP program that runs under Franz LISP on VAXes, and it brings AI techniques into educational-software development. It has been observed that it can lead college students to faster, more effective learning of LISP programming.

IPSEN[48], an Incremental Programming Support Environment, is an environment in which all software documents are represented internally by graph-like high level data structures. It supports programming in large as well as programming in small.

SEE[39], A Student's Educational Environment, is designed to teach ADA programming language. It provides a toolset that includes an "Analogy" Knowledge Base, a network of language subsets and a structure oriented editor.

Bonal et al[12] developed an Informal Programming Language, IPL, which is based on programming like semantics with a Natural-Language like format. The approach relies on a system knowledge of the domain to disambiguate the IPL code. This type of interface can be used as an interface for a database system and as the front-end of a programming tutor.

Integral C[59], developed at Textronix, Inc., is an industrial grade integrated programming environment for C programming on an engineering workstation. It runs under Ultrix on a VAX station equipped with a bitmapped display.

Cedar[65] is a programming environment which was developed at Xerox Palo Alto Research Center. Cedar supports the development of programs written in a single programming language, also called Cedar. Its primary purpose is to increase the productivity of programmers whose activities include experimental programming and the development of prototype software systems for a high-performance personal computer.

O. PROGRAMMING IN THE LARGE

The integrated programming environments for programming in the large includes facilities to automate all kinds of software-development activities, such as organization and project management, requirements definition, system design, software design, implementation, programming, quality assurance, enhancement and repair, security and privacy[34].

Facilities provided for programming in the small is only a subset of the whole environment. An environment that supports all phases of the software life cycle should be designed in a uniform manner for all aspects of software development activities. In other words, the design philosophy and command language must be uniform for all the tools provided by the system. The transitions from one tool to another must be smooth. Examples of such environments can be found in references 15, 28, and 48.

III. DESIGN AND IMPLEMENTATION OF IPE-PC

An integrated programming environment that supports structured programming was designed and implemented. This environment supports both pseudo-code and source code development. A Pascal like programming language has been developed and used as the basis of the environment. The internal representation of a program is a graph and used uniformly in every mode of the environment. The environment supports the editing, compilation, interpretation, debugging, and translation of programs.

A. LANGUAGE DESIGN

A programming language is designed according to the requirements of a language based environment. A lot of features are implemented as in Pascal. Only the original features of the language will be discussed here.

1. Motivation.

Programming languages are designed according to the requirements of the environments in which they are used. In the traditional programming environments, the programs are developed by using the text editors where the programs are represented as text, a sequence of characters. This text is then processed by the compilers to produce an executable version of the same program. Therefore, some redundant features, such as keywords and punctuation symbols, are used in the language syntax as aids to the user. For example, in the following Pascal program segment;

```
WHILE (I < = N) DO
    BEGIN
        NFACT := NFACT * I ;
        I := I + 1
    END
```

the BEGIN-END pair is needed to enclose the statements that belong to the WHILE loop. Since programming languages like Pascal are free format, such keywords and/or punctuation symbols are needed to show the scope of a statement list. The compiler for such a language is blind to the format and indentation that is used by the programmer. In other words, the programmer has to learn to use keywords and punctuation symbols appropriately for the sake of

the compiler and a reasonable set of format and indentation rules for the sake of himself and other interested parties. Basically, a combination of formal and informal rules have to be used to develop programs in such languages. Since every programmer has a different taste, the programs which are created by using different styles may look different although they follow the same formal rules. Another typical example is the dangling-else problem;

```
IF (Number > = 0) THEN
      IF (Number > Largest) THEN
            Largest := Number
      ELSE IF (Number < Smallest) THEN
            Smallest := Number
ELSE
      WRITELN('Negative Number');
```

In the Pascal segment above, the intention of the programmer is clear from the program segment ; ELSE part is intended to belong to the first IF statement. On the other hand, a Pascal rule states that in a nested IF statement, each ELSE clause is matched with the nearest preceding unmatched IF, so the program segment above is interpreted by the compiler as if the intention of the programmer is as follows:

```
IF (Number > = 0) THEN
      IF (Number > Largest) THEN
            Largest := Number
      ELSE IF (Number < Smallest) THEN
            Smallest := Number
      ELSE
            WRITELN('Negative Number');
```

Even experienced programmers may find themselves in a similar situation. The problems stated above can be eliminated if the language is designed as fixed-format. If a template driven editor is used to support such a language, the correct format and indentation will be provided by the system. Keywords will only be used to differentiate instruction types. There is no need to use the punctuation symbols because the format and indentation of the statements will show their scope. This would eliminate problems caused by misspelled keywords and missing/excessive punctuation symbols. A problem in such an environment may come from the small screen area that is provided on most terminals. If the nesting level is too deep, there may not be enough space left on the screen for the indentation. This problem can be solved by providing a left-right scroll facility in the editor. Indentation amount can be adjusted by the system and/or programmer automatically according to nesting level or taste. Fortunately, deep nesting does not occur very often. Even if the left-right scroll facility is not provided, the system may inform the

user about the problem and ask the user to reduce the indentation or implement the current refinement as a procedure.

Another aspect that should be considered in the design of an integrated programming environment is its suitability to simulate the programming process. The traditional programming environments assume that the program has already been developed on a paper in the form of a pseudo-code or the programmer has a clear picture in his mind about the control structures (i.e., procedures and their interfaces) when he sits in front of the terminal. Therefore, when the programmer enters the code he does not necessarily include documentation, since typing is a time consuming process. Documentation is probably left on a scratch paper that is used when the program logic is developed. Missing documentation often becomes a problem during program maintenance which is a very costly process.

Most text books on programming encourage the programmers to design programs top-down by stepwise refinement. Top-down programming produces program documentation as a natural output of the process. Automation of program development opposed to program typing provides the following advantages:

1) The documentation will be saved automatically.

2) It will provide a more organized, cleaner and faster environment for program development.

3) Such an environment can be upgraded to provide automatic source code generation from the pseudo-code.

One of the important objectives of this project is to design an environment that can automate top-down programming by stepwise refinement. Such an environment can automate pseudo-code development as well as source code.

2. Control Structures.

The simplicity is one of the main objectives of this design. Only three basic logic control structures are sufficient for the expression of any program logic[34]. These basic control structures are sequence, IF-ELSE, and WHILE structures. In the current prototype, sequence, IF-ELSE

IF-ELSE, WHILE, UNTIL and FOR control structures are implemented. However, the language can be upgraded to include more sophisticated control structures like CASE.

a. Sequence.

The sequence structure is concatenation of one or more statement nodes. It is represented as a linked list internally as shown in Figure 1. It is possible to insert and delete new statements at any point in the list. All of the statements that belong to the same sequence structure have the same indentation level. Each of the statements in the list can be expanded by using the appropriate templates, such as assignment, if, while, etc..

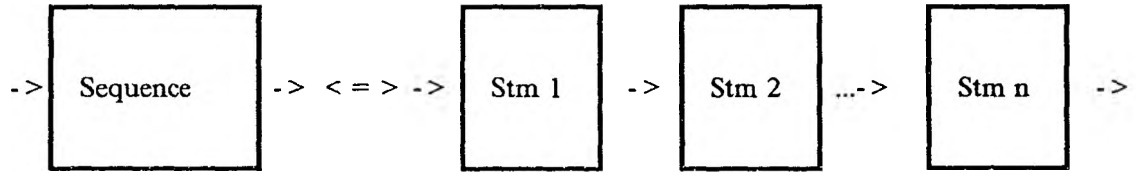


Figure 1. A Sequence Control Structure

b. Comments.

A comment is text written in pure English to explain parts of the program. It starts with a special character that is produced by the editor. The character that is used in the prototype is "-". It is preferable to use one of the special graphic characters provided on the terminals for this purpose. However, this would make the implementation terminal dependent. The editor can be modified to include commands to change this character according to the taste of individual programmers.

The difference between comment statements in this language and Pascal is not only syntax. A comment statement in this language can be best described as the description of refinement steps. The statements that are used to achieve the purpose of a comment are indented to show the logical structure of the program. So, the programmer may start programming by specifying the important tasks that should be performed for the solution of the problem. Each step will be described by using one comment statement. Then, these comments

can be refined by using other comments and/or statements which are indented properly. By using such a comment facility, stepwise refinement of top-down programming can be automated effectively. One distinguishing feature of this environment is the following: the programmer can choose to transform a refinement step to a procedure anytime. If the programmer points to a step and issues a procedure construction command, the system will ask for the name of the procedure and then it will create a procedure with this name. The body of the procedure will be the comment and its refinement. The corresponding program segment will be deleted from the main program by the system. The user can insert the appropriate call statement at that position and define the procedure interface by using the editor commands. A conversational facility can be used to help automation of the procedure interface description and procedure call insertion at this point. Procedure construction commands can be issued for the other control structures as well.

c. IF Statement.

It provides a choice between two alternatives as it is described in Figure 2.

```
IF (expression) THEN
  sequence A
ELSE
  sequence B
```

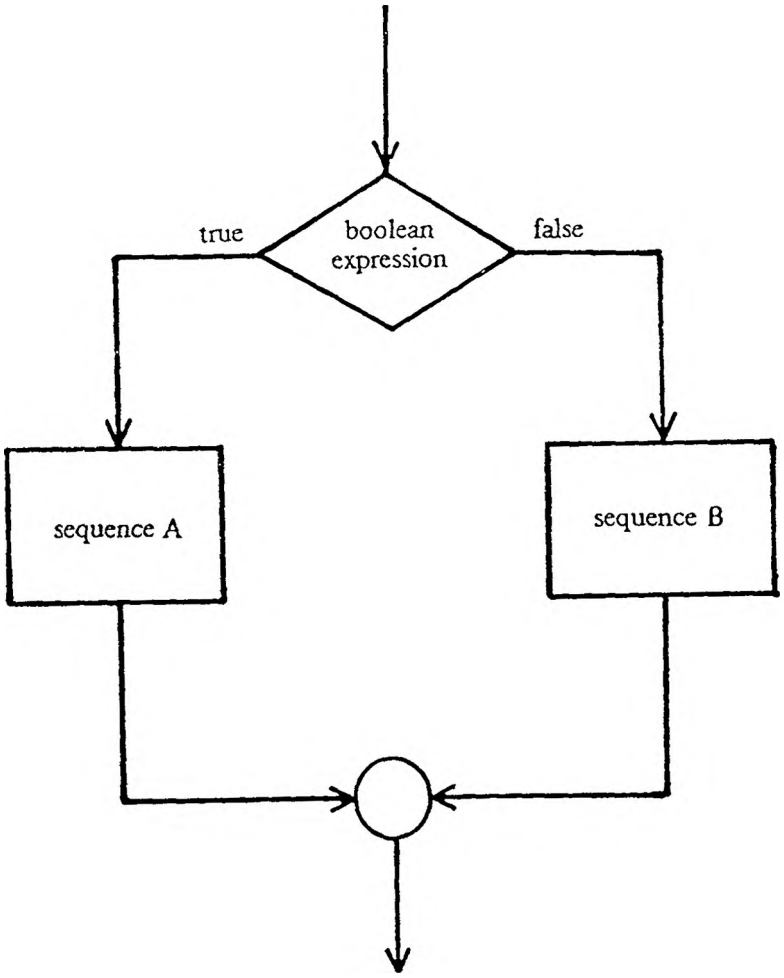


Figure 2. An IF-ELSE Logical Control Structure

d. IF-ELSE IF-ELSE Structure.

This control structure is used in the applications where the control flow depends on a series of predicates. The actions that have to be performed for each predicate is different. Although this type of logic can be implemented by using a nested IF-ELSE structure, inclusion of IF-ELSE IF-ELSE structure does not necessarily imply a redundancy in the language. It eliminates the deep nesting levels that might be a problem in a template driven editor. Another problem with the deep nesting level is poor readability.

Since CASE statement is a special case of IF-ELSE IF-ELSE structure, it is not included in the design.

IF P_0 THEN
  sequence 0
ELSE IF P_1 THEN
  sequence 1
ELSE IF P_2 THEN
  sequence 2
  .
  .
  .
ELSE IF P_n THEN
  sequence n
ELSE
  sequence n + 1

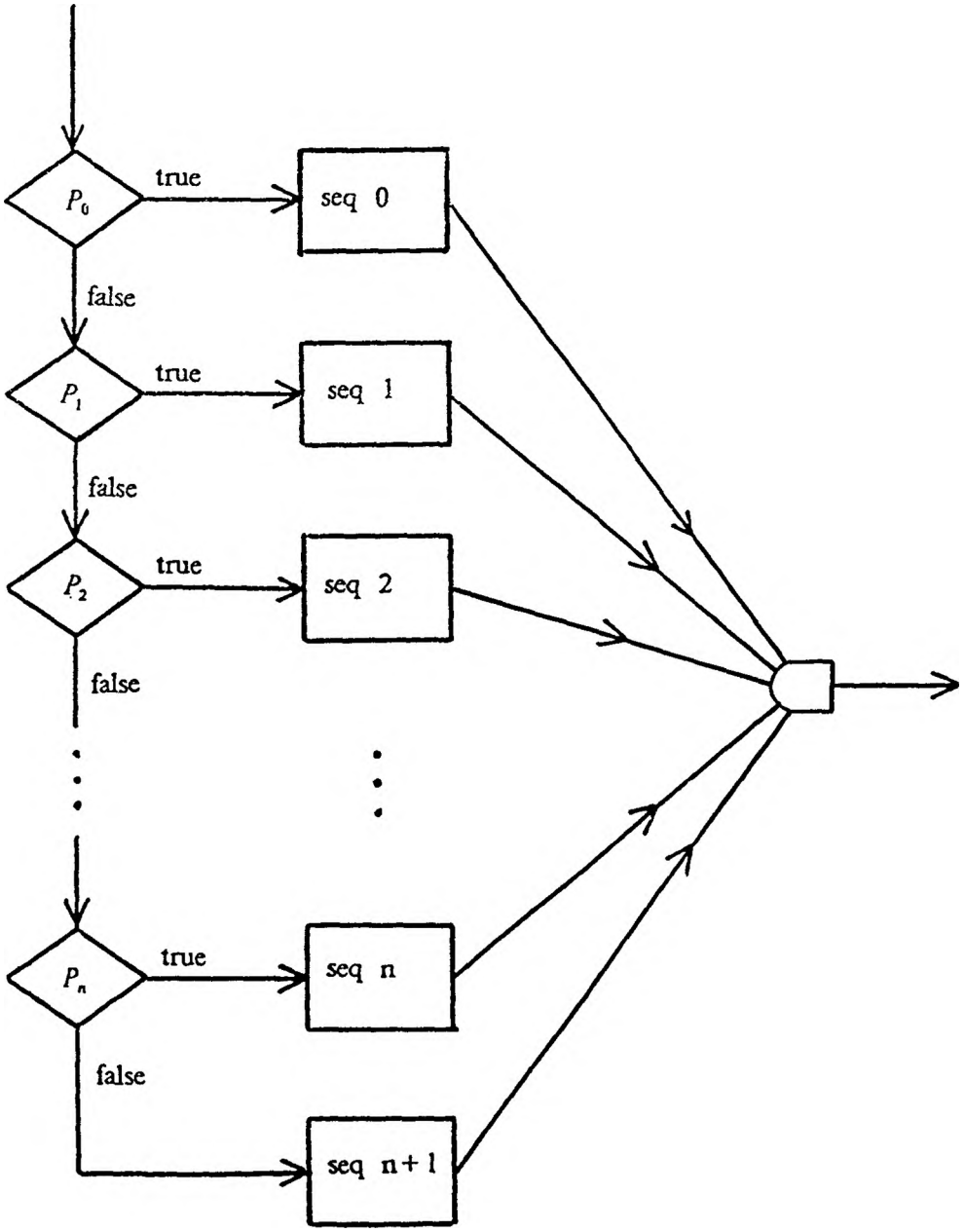


Figure 3. An IF-ELSE IF-ELSE Logical Control Structure

e. Iteration.

WHILE , UNTIL, and FOR structures are included in the language to provide basic looping capability. These iteration structures are described in Figures 4, 5, and 6.

WHILE (expression) DO
sequence

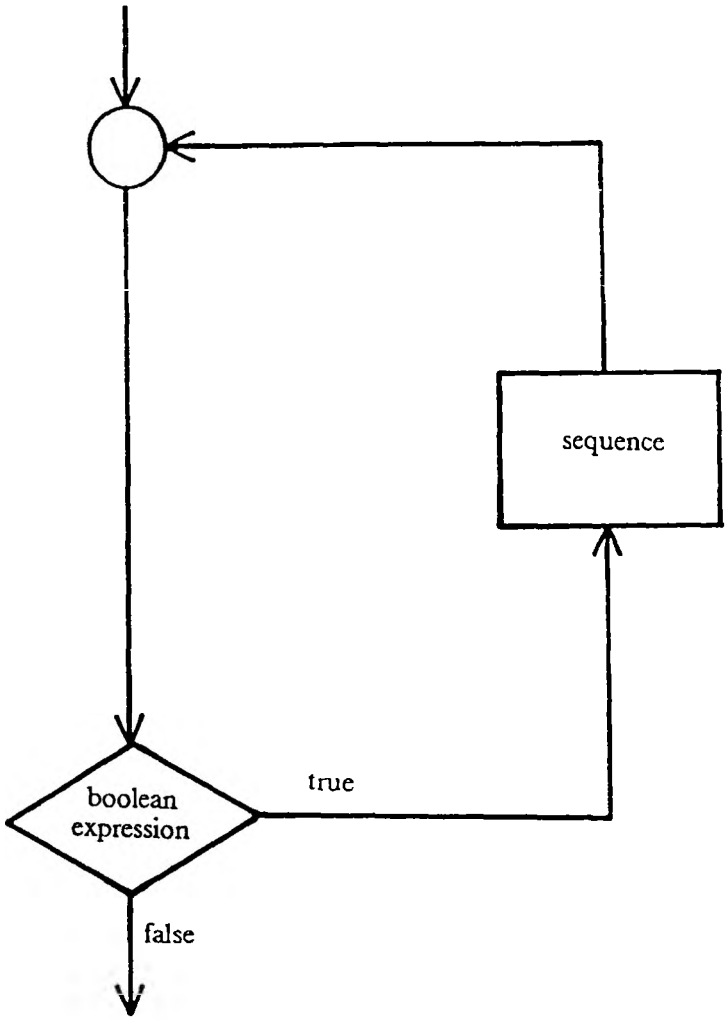


Figure 4. A WHILE Logical Control Structure

```
REPEAT
   sequence
UNTIL (expression)
```

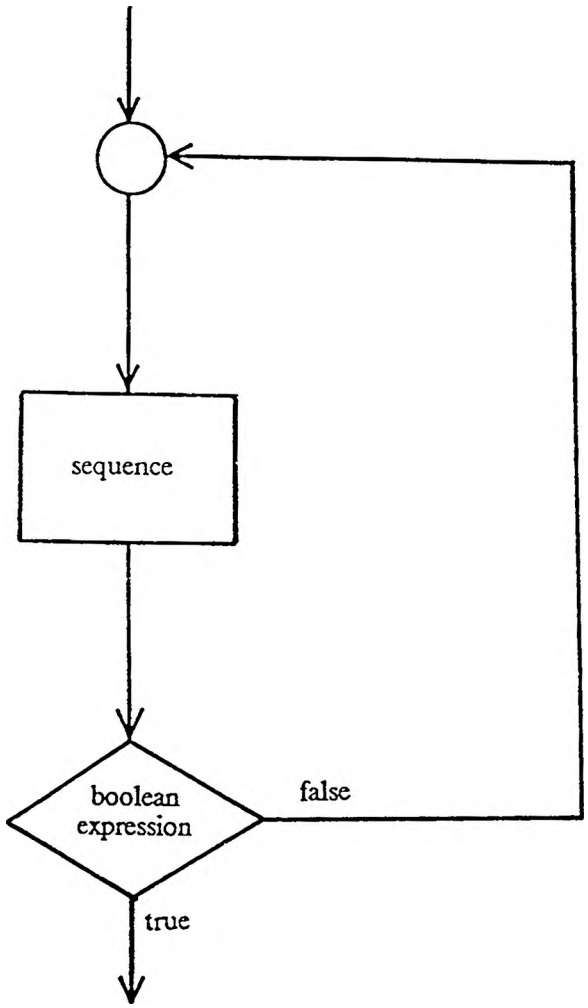


Figure 5. An UNTIL Logical Control Structure

FOR var <- initial TO final BY inc
    sequence

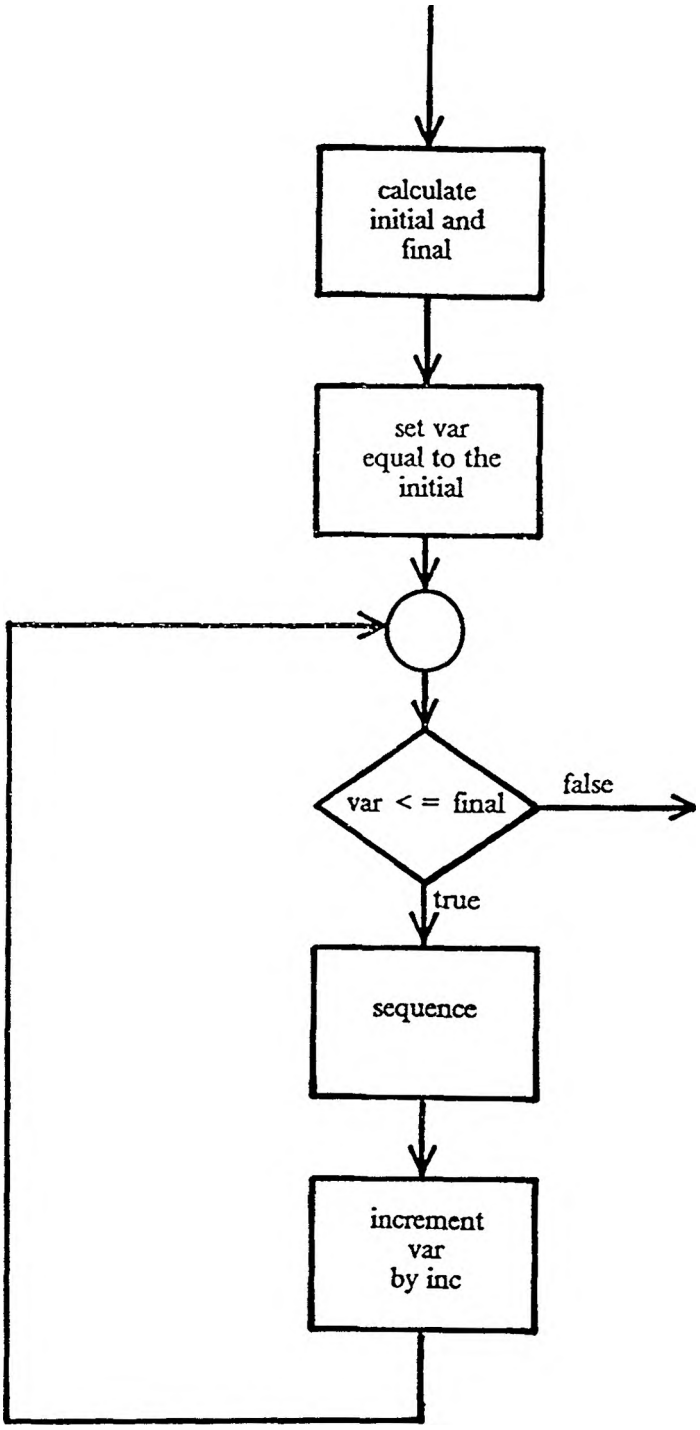


Figure 6. A FOR Logical Control Structure

3. Procedures.

Pascal uses different syntax for procedures and functions. A simpler approach, similar to C[37], is preferred in this design, a single procedure type will be used to define both procedures and functions. When the procedure call is used as a separate statement the returned value is ignored, otherwise the returned value will be used as the value of the procedure.

The procedure interfaces must be described explicitly. Procedures communicate with the caller through the parameters and global variables. There are two kinds of parameters namely input and output parameters.

a. Input Parameters.

Input parameters specify the locations for the input values which must be supplied for the execution of the corresponding procedure. They are implemented by "call by value" mechanism. So, any updates to these variables do not create any side effect.

b. Output Parameters.

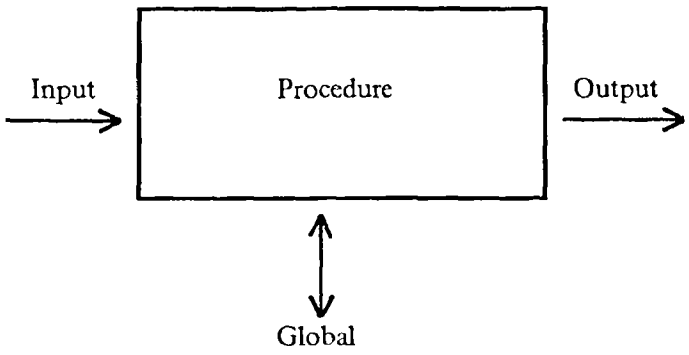
Output parameters are the values calculated by the procedure to be returned to the invoking procedure. They are implemented by "call by reference" mechanism. Any update to these locations will be reflected in the corresponding actual parameters.

c. Access to Nonlocal Names.

Any global access must be defined explicitly in the symbol table of the procedure. Procedures can only access the variables of the main program as globals. The scopes of the variables are lexical. Since no nested procedure declarations are allowed, all nonlocal names may be bound to statically allocated storage.

The syntax of a procedure call is similar to Pascal with one exception. Actual parameters corresponding to the formal input parameters precede the actual parameters corresponding to the formal output parameters. This principle simply reflects the black box view of the

procedures. It is only logical to place these two group of parameters separately to increase the readability.



4. Views.

It is possible to generate different views of a program in a language based environment. Views can be classified as local and global. Description of these views are important to explain language design as well as implementation because these views show how the program is presented to the programmer. The operations that can be performed in each view will be discussed later.

a. Local Views.

Local views are about the specific information for each procedure and main program. There are three kinds of local view:

1) Algorithm View: This view displays the algorithmic structure of the procedure or main program.

2) Symbol Table View: It displays the declaration of identifiers that are referenced in the corresponding algorithm. Each identifier has its own definition which consists of four fields: name, type, kind, and comment. There is no need to use any separators between the fields and declaration units since the symbol table is fixed format. Declarations, with <unknown> type and kind attributes, are created automatically during the program development if the corresponding identifier is not already defined. This symbol table view corresponds to the variable declaration statements of Pascal.

Symbol name is a unique identifier whose length is up to eight characters.

Symbol type is either a basic data type (integer, character, real, and boolean) or a user defined type identifier.

Symbol kind can be input, output, global, local, or proc. Input kind is used to define formal input parameters. Output kind is used to define formal output parameters. Global kind is used for the explicit declaration of global variable access and local kind is used to define local variables. Similarly, proc kind is used to represent identifiers for the procedure names.

Symbol comment field can be used to explain the purpose of an identifier.

3) Execution View can be used to display the contents of the symbols in the current scope during the execution. Execution view is implemented by using the data structure used for the symbol table view.

b. Global Views.

Global views summarize global information for the whole program.

1) Procedure view: This view creates an index to all the procedures and its main program. Names of the procedures are displayed as a list. Operations are provided to define and/or change the interface of each procedure.

2) Type view: Definition of user defined data types is global. Since all the user defined types have to be created in this view, type definition can not be part of a variable declaration. This approach simplifies the implementation and language.

Each of these views are implemented as a separate window. In the current implementation, it is only possible to see one view at a time.

5. Input and Output Facilities.

READ and WRITE statements are implemented as procedure calls supported by the environment. A conversational style for input editing is implemented as follows: When a READ statement is interpreted, the system generates a message that gives the name of the identifier and the expected data type. If the programmer enters a wrong data type, the system produces a diagnostic message and asks again for a correct value. This conversation goes on until the programmer enters correct data or quits the execution. This simplifies the coding, because the programmer does not have to include unnecessary WRITE statements to create an informatory interactive environment. Similarly, the WRITE statement prints the name of the identifier with the corresponding data value. All of this conversation is kept in a file that can be referred to at the end or during the execution. Appendix F includes an interactive I/O file produced from the execution of an example program given in Appendix C.

B. INTERNAL REPRESENTATION

The internal representation of programs is a graph. The start node of the graph contains ten fields; one field points to a linked list called "type-list" and another points to a linked list called "program-list". Both are doubly linked lists. The other eight fields are used for mapping information about the type and program lists.

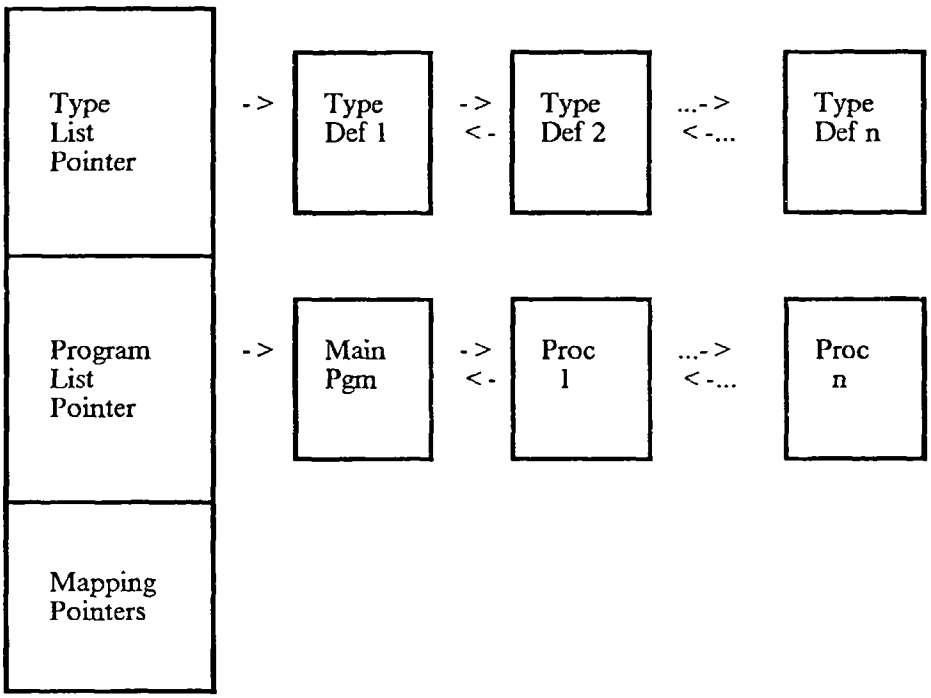


Figure 7. A Start Node of Program Graph

1. Type-List.

Type-List is used to create the type view. Every node in the list contains the following fields:

Name, a unique type identifier whose length is up to eight characters.

Code shows the kind of user defined data type. Array and structure types are implemented in the current prototype.

Size shows the storage requirements for an object with this type.

Comment explains the purpose of the type definition.

Row and Lines are used to calculate the mapping information. "Row" shows the current row for this type definition. "Lines" shows the number of necessary lines to display the whole type definition.

Id-List, a linked list of symbol table entries that use this type definition.

Type-Ptr points to the type descriptor for this aggregate data type. Different type descriptors are needed for the specification of arrays and structures.

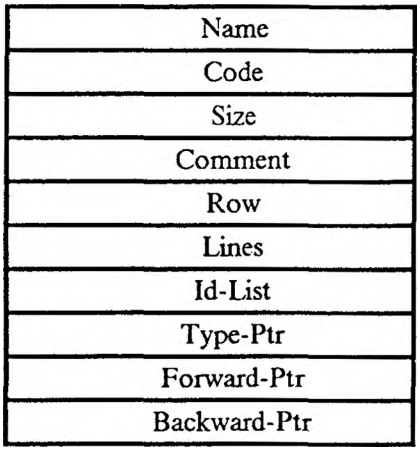


Figure 8. A Type Node in the Type List

Type Descriptor for array is a node that contains the following fields:

Information contains the text entered by the programmer for the definition of array dimensions.

Object contains information for the number of dimensions and limits for each dimension.

Code shows the data type of each element in the array. Separate integers are used to show basic data types. If the code is aggregate data type then Type-Ptr field of the descriptor points to the corresponding node in the Type-List.

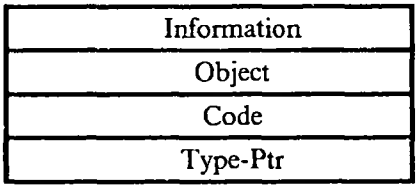


Figure 9. An Array Descriptor

Type Descriptor for structure is a doubly linked list of field definitions. Each node in the list contains the following fields for the description of a field:

Name, a unique field name whose length is up to eight characters.

Offset shows offset of this field in the structure.

Code shows the data type of this field. If the data type is aggregate, then Type-Ptr field in the node points to the corresponding type definition in the Type-List.

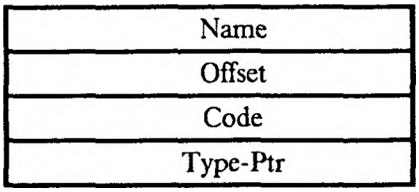


Figure 10. A Field Descriptor

2. Program-List.

Program-List is used to create other views in the environment. The first node represents the main program. The rest of the nodes are for the procedure definitions. A Program-List node contains the following fields:

Name, a unique procedure name whose length is up to eight characters. Name of the main program is "main" by default.

Symbol Table, a data structure for the symbols defined in this procedure. In the current prototype, an open hash-table of size 100 has been used as the data structure for the symbol table implementation. A hash function is applied on the symbol names to find hash position. The identifiers with the same hash value makes up a linked list at that hash position. The fields of a symbol node in the hash table are described as follows:

Name, a unique symbol name whose length is up to eight characters.

Comment field can be used to insert explanation about the symbol.

Offset of the symbol in the activation stack.

Type can be a basic data type (integer, character, real, or boolean) or an aggregate data type. Integer codes are used to differentiate between basic data type and aggregate data type. If the type is aggregate then

Type-Ptr field points to the corresponding type definition in the Type-List.

Reference List pointers are used to show the head and tail of reference list. This list contains addresses of the nodes that reference this symbol.

Root points to the syntax tree of this procedure. Abstract syntax tree for the algorithmic part of a procedure is constructed by using the tree nodes that are described below. Construction of syntax trees will be discussed later.

There are nine fields in a tree node:

Information contains the text entered by the programmer (i.e. assignment statements, expressions, procedure calls and comments).

Code is an integer code that shows the node type. Possible node types are procedure root, program root, if, else, else if, while, until, for, lb, ub, inc, variable, expression, procedure call, return, and statement.

Valid is a type attribute for the node. It shows whether there is a

syntax/semantic error in the information part of the node.

Parent, Child, Sibling, and Reverse-Sibling are pointer fields that point to the parent, child, right-sibling, and left-sibling of the node, respectively.

Object contains the object code produced for the expressions, assignment statements and procedure calls.

Object-Size shows the size of the object code.

Mapping Information About the Symbols is implemented with four pointer fields. The hash table keeps the information about the symbols unordered. The order of the symbol definitions is kept in a doubly-linked mapping list. Each node in this list contains two fields. The "Row" field shows the current row of the symbol definition in the window. The "Place" field points to the symbol node in the hash table. "Var-Chain" and "Var-Tail" show the first and last item in the variable mapping list. "Var-Start" and "Var-Finish" fields show the first and last symbols in the current window.

Mapping Information About the Algorithmic Part, Syntax Tree is implemeted with four pointer fields. The internal representation of algorithmic part of a procedure is an abstract syntax tree. This abstract tree has to be unparsed to construct a text image to be displayed in the algorithmic window. A mapping list is constructed by using a preorder traversal of the tree. Each node in the mapping list contains the following fields:

Place points to a node in the tree.

Row shows the current row of the text that corresponds to the text image of the node.

Line shows the number of lines needed to display the whole unit.

Col shows the indentation level.

"Chain" and "Tail" show the first and last nodes in the mapping list. "Start" and "Finish" show the first and last nodes of the mapping list in the current algorithm window.

Size shows the size of the activation record for the procedure.

Input, Output and Global pointer fields are used for the description of the procedure interface. Each of them points to a linked list. A node in the input list simply points to a formal input parameter in the symbol table. Output and global lists are defined similarly. A node in the

output list points to a formal output parameter and the global list is used to keep track of global variables.

Name
Symbol Table
Root (Pointer to the root of the program/proc tree)
Four mapping pointers for the Symbol Table
Four mapping pointers for the algorithmic part
Size
Pointer to the Input List
Pointer to the Output List
Pointer to the Global List

Figure 11. A Procedure or Program Node

3. Mapping Pointers.

Four pointer fields are used for the mapping information of the type list. Two of them are used to point to the first and last type nodes in the type list. The other two are used to show first and last nodes in the current window. These pointers are used to create the Type-Window.

Other four pointer fields are used to create the Procedure-Window similarly.

C. LANGUAGE BASED EDITOR

A template driven language based editor has been used as the basis of the programming environment. Templates are used to generate major control structures. Expressions, left hand side of the assignment statements, procedure calls, and comments are inserted in the text mode. Syntax trees for the program/procedure are constructed by the usage of the templates during the program development. A recursive descent parser[1] has been used for the expression parsing and translation into a postfix object code.

1. Templates.

Templates are used to develop algorithmic structure, variable declarations, and type definitions of the program as described below.

a. Algorithm Window Templates.

Each template corresponds to a subtree that can be inserted into the syntax tree as the expansion of a template/placeholder. Templates and the corresponding simplified trees are described below (the dash under the placeholders shows the current position of the cursor). Each node in the tree consists of four fields. The code field is an integer that shows the node type, information points to a text string, child points to the first of the children of this node, and sibling points to the sibling of the node. Figure 12 describes the tree node that will be used in the following trees. Only the contents of the fields will be shown in the trees. A constant identifier has been used to show different node types. If a pointer field is never used in a node, its content will be represented as 'NIL'. If it can be expanded, it will be represented as '.'.

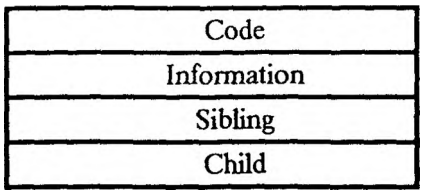


Figure 12. A Tree Node

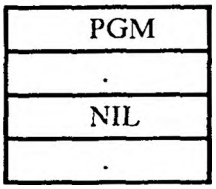
1) Main Program Template

It is generated by the editor as the root node of the main program.

Template

Main Program :

Tree

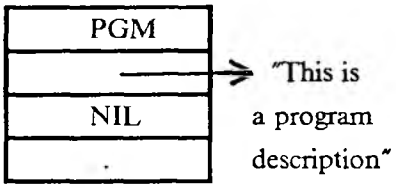


If an insert command is applied at the main program template, the cursor moves to the end of the header string, "Main Program :", and any text entered by the programmer is typed starting from this position.

Template

Main Program : This is a program description

Tree

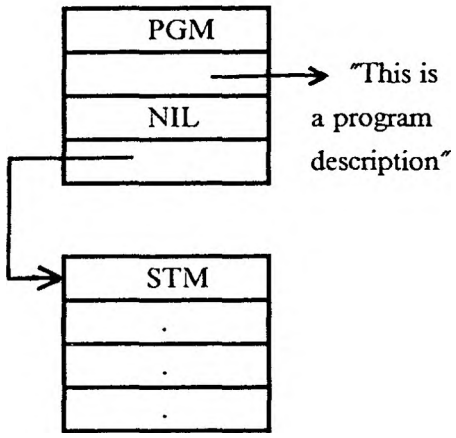


Only a refinement step, an indented statement template, can be inserted to expand a main program template/comment. After applying the refinement step command at the current position of the cursor the syntax tree is modified as follows.

Template

Main Program : This is a program description
    < statement >

Tree



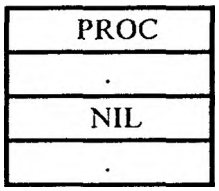
2) Procedure Template

It is generated as the root node of the syntax tree when a new procedure is constructed (for the following example, assume that the procedure name is "sort").

Template

sort procedure :

Tree



Editor operations are applied on this template in a similar way to the main program template.

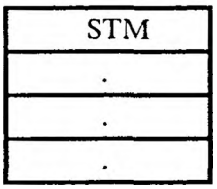
3) Statement Template

It is used to generate a statement in the program.

Template

< statement >

Tree

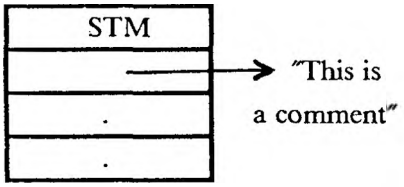


It can be expanded as a comment by inserting text at the current position of the cursor.

Comment

-This is a comment

Tree

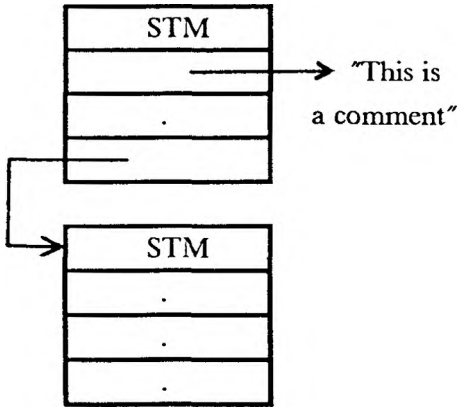


A refinement step can be inserted as the expansion of a comment template as follows.

Refinement Step

-This is a comment
< statement >

Tree



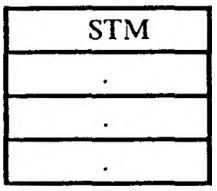
4) IF Template

It is used to generate an IF statement. A statement template can be expanded as an IF template as follows.

Template

< statement >

Tree

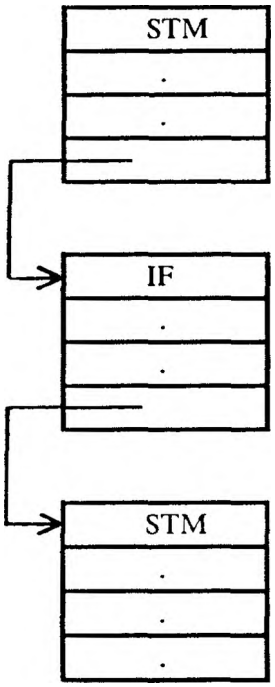


An "IF" command produces the following expansion at the current position of the cursor.

Template

IF < expression >
< statement >

Tree

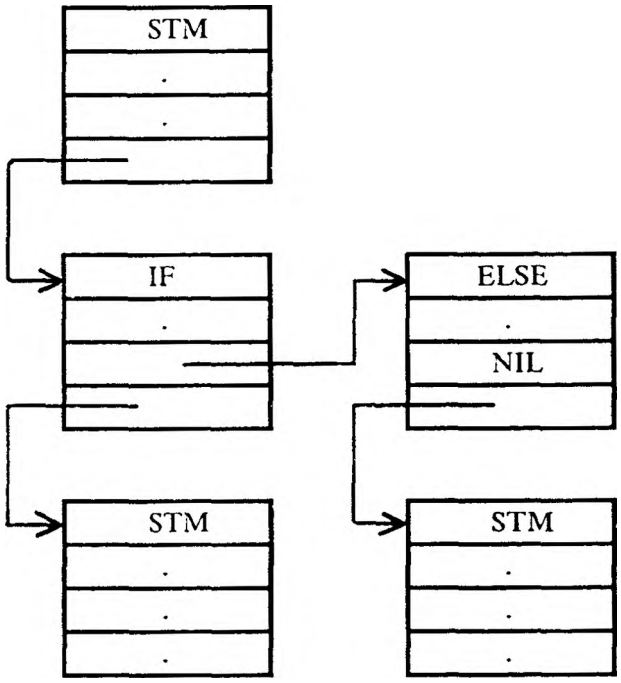


ELSE part of the IF statement is optional. An "ELSE" command at the current position of the cursor produces the following expansion.

Template

IF < expression >
    < statement >
ELSE
    < statement >

Tree



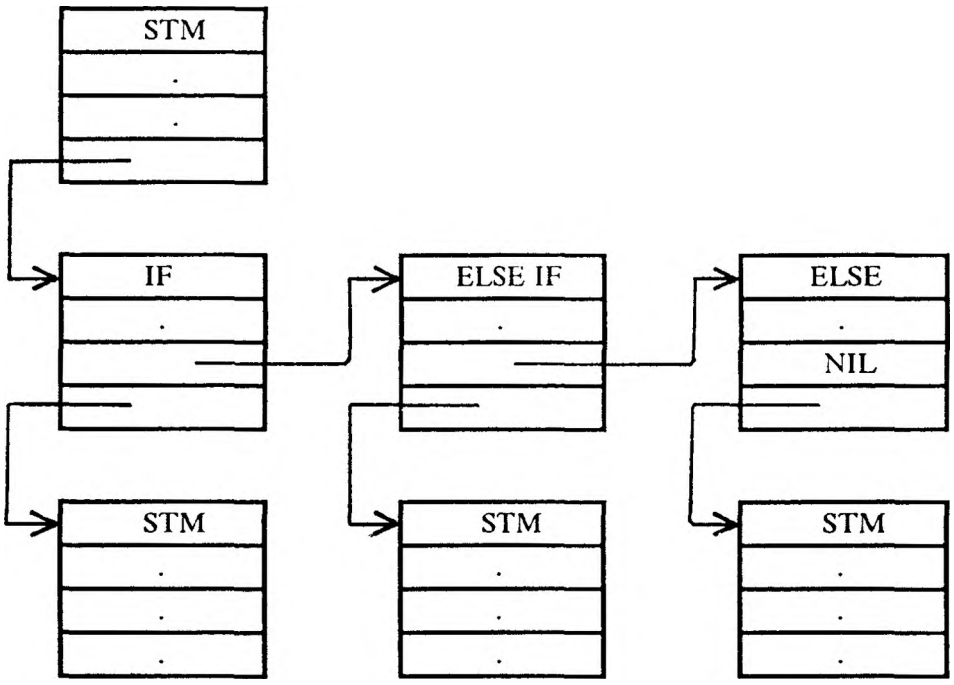
5) ELSE IF Template

It is used to generate an "ELSE IF" part in an existing IF statement. An "ELSE IF" command at the current cursor position of the program segment above produces the following expansion.

Template

IF < expression >
    < statement >
ELSE IF < expression >
    < statement >
ELSE
    < statement >

Tree

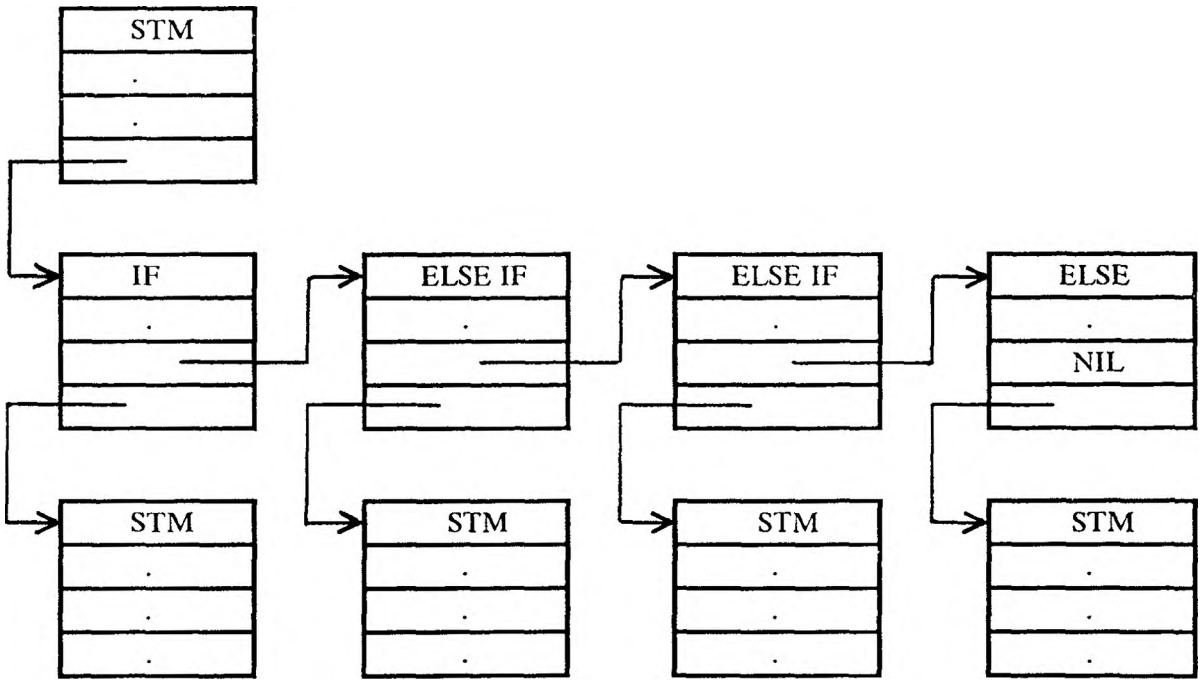


"ELSE IF" command can be applied at the beginning of an ELSE IF template as well. A second application of the command at the current position of the cursor produces the following modifications.

Template

IF < expression >
    < statement >
ELSE IF < expression >
    < statement >
ELSE IF < expression >
    < statement >
ELSE
    < statement >

Tree



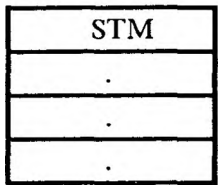
6) WHILE Statement

It is used to generate a WHILE statement. A statement template can be expanded as a WHILE statement as follows.

Template

< statement >

Tree

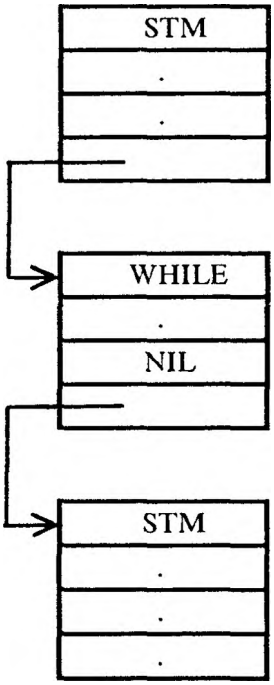


A "WHILE" command produces the following expansion at the current position of the cursor.

Template

WHILE < expression >
< statement >

Tree



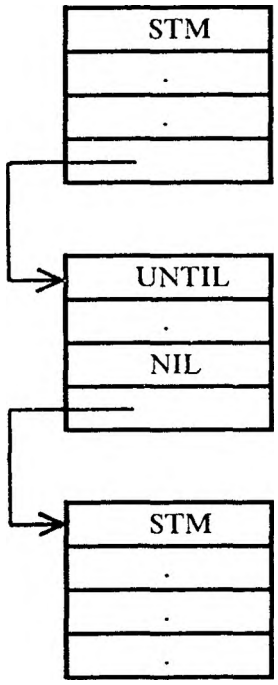
7) UNTIL Statement

An "UNTIL" command produces the following expansion when it is applied on a statement template.

Template

REPEAT
    < statement >
UNTIL < expression >

Tree



The internal representations of WHILE and UNTIL templates are the same. Therefore, implementation of commands that can change an existing WHILE statement to UNTIL or vice versa does not carry any overhead for the modification of internal structure. Only the displayed text has to be reprinted by using the appropriate keywords.

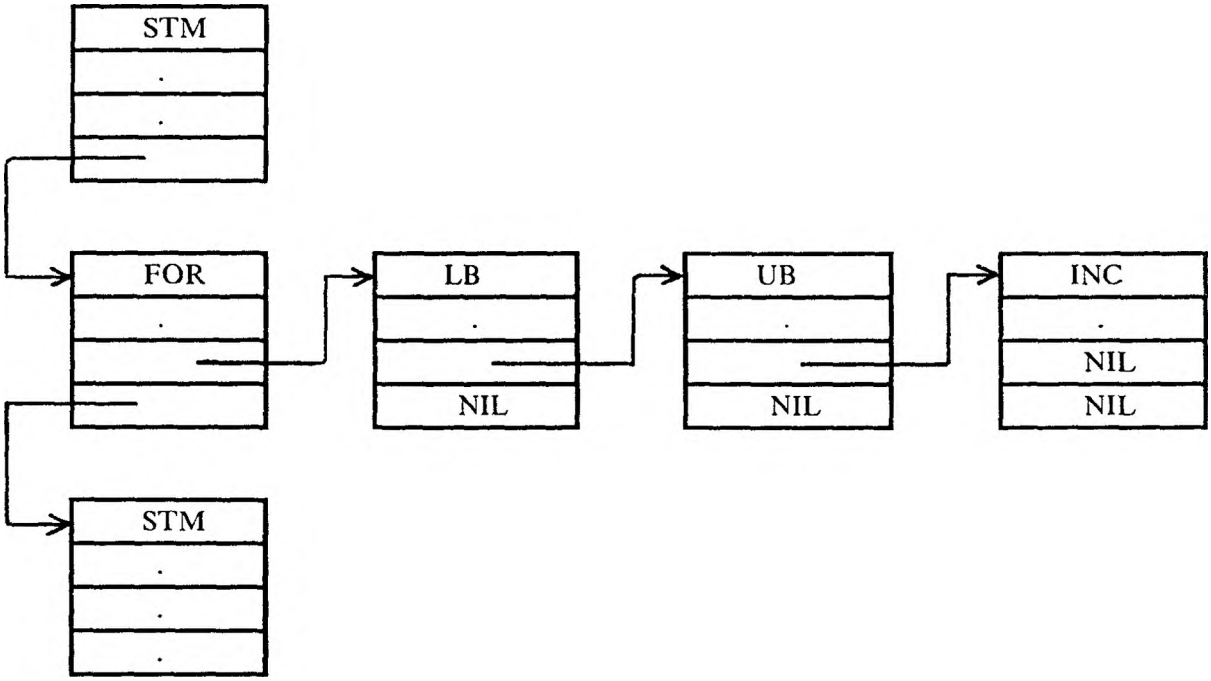
8) FOR Statement

A "FOR" command produces the following expansion when it is applied on a statement template.

Template

FOR <var> <- <lb> TO <ub> BY <inc>
<statement>

Tree



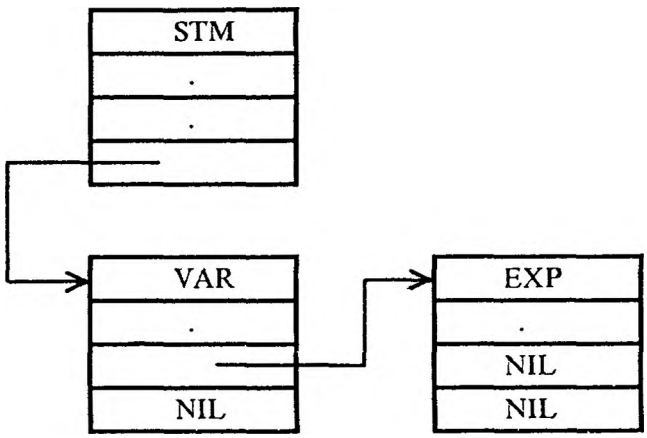
9) Assignment Template

It is used to generate an assignment statement. An "assignment" command at a statement template produces the following expansion.

Template

< var > <- < expression >

Tree



Assignment statement is the same as the assignment statement in Pascal with the following exception; it is possible to assign a real quantity to an integer location by omitting the fractional part. The assignment operator, "<-", is also different. The environment can be made flexible enough to change this operator during the edit session.

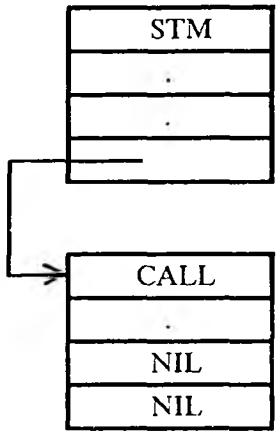
10) Procedure Call Template

It is used to generate a procedure call statement. A "procedure call" command at a statement template produces the following expansion.

Template

< Procedure Call >

Tree



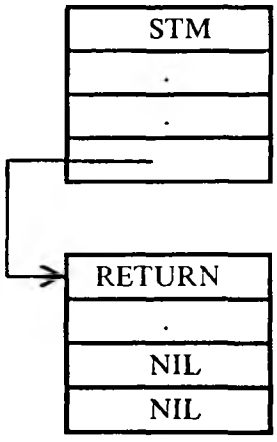
11) Return Template

It is used to generate a return statement. A "return" command at a statement template produces the following expansion.

Template

return()

Tree



Insert command can be used to insert an expression in the return template.

b. Variable Window Templates.

Variable declarations are created by using the variable template. A variable template consists of four fields.

Variable Template

< identifier > < type > < kind > < comment >

The first placeholder can be expanded by inserting a unique variable identifier, the second placeholder can be expanded by inserting a basic data type or user defined type identifier. Basic data type identifiers are int, real, char, and bool for integer, real, character, and boolean types, respectively. The kind field is expanded by inserting one of the five possible kinds; input, output, global, local, and proc for the formal input parameter, formal output parameter, global variables, local variables, and procedure names, respectively. Comment field can be expanded by inserting a text for the explanation of that variable. The internal representation of variable declarations is a combination of a variable mapping list and a hash table as was discussed previously. Each of the fields in the template has a fixed starting column in the window.

c. Type Window Templates.

The basic type template consists of three fields.

Type Template

< type identifier > < type description > < comment >

The first placeholder can be expanded by inserting a unique type identifier. The second placeholder can be expanded by using an array or structure template. After applying "array construct" command at the current cursor position of the following type template,

list < type description > < comment >

the following expansion is obtained:

```
list  array   < dimensions >
      of  < type >          < comment >
```

< dimensions > placeholder can be expanded by entering a valid dimension description, and < type > placeholder can be expanded by entering a basic data type or user defined data type identifier. After applying "structure construct" command at the current cursor position of the following type template,

```
data   < type description >   < comment >
```

the following expansion is obtained:

```
data  struct              < comment >
      < field > : < type >
```

< field > placeholder can be expanded by entering a unique field identifier. Expansion of < type > placeholder is described earlier. Any number of fields can be added to the structure by using editor commands. Internal representation of type templates is discussed in the previous section.

2. User Interface.

User interface will be described below. The details can be found in the user manual provided in Appendix B. The screen area is divided into three windows described as follows.

a. Top Window.

The first 21 lines of the screen are used as the top window. There are ten possible views supported by the system, namely: welcome, file directory, clip directory, clip display, procedure, procedure interface, algorithm, symbol table, type and execution. In the current prototype, it is only possible to see one view at a time in the top window.

Welcome View

Welcome view displays a welcome message. Programmer can use the commands displayed at the menu window to perform the following actions;

- to look at the file directory,
- to edit a program, and
- to leave the environment.

File Directory View

File directory view displays the names of the programs in the directory. The operations that can be done in this context are;

- a program file can be deleted from the file directory,
- a program file can be copied to create a new program, and
- quit command takes the programmer back to welcome window.

Clip Directory and Clip Display Views

Clip directory view displays the names of the program segments, clips, that are saved by the programmer. Clips can be created in the algorithm window by deleting or copying an existing program segment. These segments can be copied to the appropriate points as the expansion of a statement template in the algorithm window.

Unwanted clips can be deleted in the clip directory window. It is also possible to invoke the clip display view to look at the contents of a specified clip.

Procedure and Procedure Interface Views

Procedure view displays names of the procedures and main program. The following operations are possible;

- to change the order of the procedures in the window,
- to look at the procedure interface view for a specified procedure and change the order of parameters,
- to copy a procedure,
- to delete a procedure,
- to invoke the algorithm view of a specified procedure or a main program,
- to quit the view by saving the current program and return to the welcome window after issuing quit command.

Algorithm View

Algorithm view is supported by a template driven language based editor. Algorithms are constructed by using templates and inserting text at the appropriate placeholders. Diagnostic messages about syntax and semantic errors are displayed in the message window. The following operations can be performed;

- Expanding < statement > placeholder as a comment. Comments can also be inserted at the main program and procedure header templates.
- Inserting text at the < var >, < expression >, < procedure call >, < lb >, < ub >, and < inc > placeholders. When text is entered/modified at these placeholders, object code in postfix form is created and inserted into the internal node. Attributes of the node are also adjusted according to the results of syntax and semantic checks. If there is an error, a diagnostic message is displayed in the message window and the corresponding statement on the screen is highlighted.
- The text which is inserted by using the comment construction and insert operations can be modified in text editor mode.
- Expanding < statement > placeholder with IF template.
- Expanding < statement > placeholder with WHILE template.
- Expanding < statement > placeholder with UNTIL template.
- Expanding < statement > placeholder with FOR template.

- Expanding < statement > placeholder with assignment template.
- Expanding < statement > placeholder with procedure call template.
- Expanding < statement > placeholder with return template.
- To move to parent node.
- To move to sibling node.
- To find the locations of all the statements with the syntax and/or semantic error.
- A statement and its refinement can be saved in the clip directory by deleting/copying that program segment.
- An unnamed clip can be constructed by deleting a statement and its refinement
- Only the refinement part of a statement can be deleted to construct a named or unnamed clip.
- A < statement > template can be expanded by using a named or unnamed clip.
- A statement and its refinement can be transformed into a procedure.
- A statement and its refinement can be deleted.
- Execution in single-stepping mode can be started.
- Quit command takes the user back to the procedure window.
- The cursor can move to the beginning of templates and placeholders. Different commands are provided for upward, downward, left and right direction cursor movements.
- The refinement steps that belong to if, else, while, until, for, program and procedure headings, and comments are implemented as a statement list. Commands are provided to insert a statement preceding or following a current statement.
- To produce the listing file of the program.
- To produce the Pascal version of the program.
- To print out the input and output file.
- To invoke type and local symbol table views.
- To invoke the algorithm view of a specified procedure.

Symbol Table and Type Views

Symbol table view is used to create variable declarations in a template driven fashion as discussed earlier. During the program development if one of the variables that is used is undeclared, the declaration of that symbol is created automatically with <unknown> attributes.

Type view is used to define user defined data types.

Execution View

Execution view displays the contents of the variables in the current scope. It can be invoked during single stepping execution. Uninitialized locations are shown with the "<undefined>" value.

b. Message Window.

The next two lines of the screen are used as the message area. It is used for three purposes;

Input entry : Programmer enters inputs from this window.

Output display area : Outputs generated by the WRITE statements are displayed here.

Message display area : Messages generated by the system are displayed here.

c. Menu Window.

The last line on the screen is used as the menu window. It displays valid commands that can be applied at the top window.

D. EXECUTION AND DEBUGGING

Execution of the programs starts in the single-stepping[67] mode. In this mode, the cursor shows the current statement executed and the value of the current expression is displayed in the message window. The execution is controlled manually.

Some of the possible operations are;

- execute the next statement,
- switch to tracing[67] mode execution, and
- invoke the execution view to look at the contents of the variables in the current scope.

In the tracing mode execution, the current statement executed is highlighted for a short period of time, then the execution will continue with the next statement similarly.

Some of the possible operations in this mode are;

- increase the execution speed,
- decrease the execution speed, and
- switch to single-stepping mode.

Selective visual-feedback is provided to speed up the execution in both modes. When the execution flow reaches a procedure/function call, it is possible to disable the visual-feedback for that call. If the visual feedback is abled, the algorithm view of that procedure is drawn and execution continues. At the end of procedure execution the execution flow goes back to the window with the invoking algorithm view.

E. EXTERNAL REPRESENTATION

Programs are saved by creating three files. These files have the same name but different types. The file with "tree" type is created to save information about the algorithm parts of the main program and procedures. The file with "sym" type is used to save symbol table definitions and the third file with "typ" type is constructed out of the type-list. External representation closely resembles the internal representation used in the environment. Linked lists and the trees are saved in a coded form to reconstruct the data structure in a short time. Integer codes are used to represent node links, node types etc.. The strings in the files correspond to the information entered in the text mode (i.e. comments, expressions, variables, and procedure calls are saved as text). When the internal representation is constructed out of these files, the strings are reparsed to construct the object code if it is necessary.

This type of external representation speeds up the reconstruction of internal representation. Furthermore, there is no significant difference between the space requirements of this type of external representation and pure text representation. The Pascal representation of the sort program, given in Appendix D, requires 7578 bytes of disk space. On the other hand, the IPE-PC representation of the same program, given in Appendix E, requires 10142 bytes of hard disk space.

F. RUN TIME ENVIRONMENT

When the execution command is issued, type definitions are checked. If there are any unresolved type definitions, execution can not start. The storage requirements of each user defined data type is also calculated.

The second step is the calculation of activation record size for each procedure and main program. The activation size is kept in the size field of the procedure node. When the activation record size is calculated, the offsets of the variables are found and the corresponding fields in the symbol nodes are initialized. The offset fields of the global variables are initialized with their addresses since global variable locations are static. The addresses of local variables are calculated by subtracting their offsets from the top of the stack.

A run time stack is used for memory allocation. Each word in memory includes a flag for the detection of reference to uninitialized variable locations. These flags are turned off for all the words in a newly allocated activation record. When there is an assignment to a memory location, its flag is turned on. The activation record of a procedure is pushed when it is invoked. The actual input parameters are evaluated and their values are assigned to the locations of the corresponding formal input parameters. The locations that are allocated for formal output parameters are initialized with the address of the corresponding actual output parameters. Any reference to a formal output parameter location is an indirect reference. The reference to the input formal parameters and local variables are done by subtracting their offset from the top of the stack to find the address. Since a global variable offset field contains its address, there is no need for the address calculation for global variables. The procedure activation record is popped off when the execution flow returns to the caller.

Interpreter

A recursive procedure is used to interpret the program. Since expressions are translated to a postfix form, a procedure that evaluates such expressions has also been implemented. This procedure is invoked from the main interpreter when the execution flow reaches the assignment statements or expression parts of WHILE, IF, UNTIL and FOR statements.

Addressing Array Elements

Multi-dimensional arrays are stored in row-major order. The subscripts of indexes can range between 0 and an upper limit. The following formula[1] has been used to calculate the address of an array element, A[$I_1, I_2, ..., I_k$] :

$$\text{base} + ((...((I_1 N_2 + I_2)N_3 + I_3)...)N_k + I_k) \times w$$

where w : width of each array element,

base : relative address of the storage allocated for the array,

N_j : $\text{high}_j + 1$ for all j, $j = 1,k$

high_j : the upper bound of jth dimension.

G. MULTIPLE LANGUAGE SUPPORT

When the program design is completed, the program can be translated to Pascal. The implementation of the translation routines is not hard since the programs are represented as syntax trees internally. The translated program can be compiled by optimizing compilers to be used in production environments. Some example programs and their Pascal versions produced by the environment are included in the Appendices C and D.

The language used in this environment has common features with other block structured languages like PL/I, C, Pascal, and Ada. Therefore it is possible to upgrade the environment to support multiple languages by implementing translation routines for each of these languages.

H. PORTABILITY

The system is implemented under UNIX on a VAX 11/780. It can be easily transported to other systems with the UNIX environment. The implementation language was chosen as "C" because of its portability and suitability for system programming.

The screen management routines[78] that are used in the editor are provided by UNIX. To transport the system to a non-UNIX environment, these window management routines have to be implemented or the available facilities of that machine have to be used.

IV. CONCLUSION AND FUTURE DIRECTIONS

Early language-based editors appeared in the 1970's[3,21,22,25,26,31,41,46,54,62,68,75,76]. There has been quite a lot of research work in both academia and industry on this subject. Recently the work has been concentrated on integrated programming environments[6,7,16,17,20,24,45,53,61,67]. Almost all of these integrated programming environments include a structure editor which is integrated with an interpreter/compiler and debugger. More ambitious projects attempt to design integrated software development environments to support all phases of the software lifecycle[15,28,48]. Although, we would like to think that the developments in the computer science field are faster than in other fields, the reality is different. We are still programming by using independently designed tools rather than a coherent, uniform environment. The general consensus is the urgency and importance of such software[50]. The need for programming environments can be considered at three levels. The simple starting point is to design an integrated programming environment for small one person projects. The second step is of course to use this knowledge to design software development environments for team work. The ultimate goal should be to create an intelligent system that can save the information that is created during the software development. The system should contain a database to keep the reusable code and provide this as necessary in an efficient way. A knowledge database can be used to increase productivity in terms of both reusable code and easy access to requirements, specifications etc.[29,36,42,70].

The first step to develop such an environment was taken with this project. An Integrated Programming Environment, IPE-PC, that supports pseudo-code development has been designed and implemented. This environment is based on a Pascal-like language which is designed according to the requirements of a language-based environment. The nucleus of IPE-PC is a language-based editor which represents programs as graphs internally. The same representation is used in every mode of the environment(i.e. editing, compilation, execution, debugging and translation). The system provides facilities to take advantage of both top-down and bottom-up programming. Stepwise refinement has been supported by providing comment structures that can be transformed into procedures. Bottom-up programming is supported because it is possible to create and save program segments which can be inserted into programs at the appropriate points. When the program design is completed, the program can be translated into Pascal. It is

fairly easy to implement translation routines for different languages because of the tree like program representation.

Future Directions :

The prototype should be upgraded to include file I/O facilities and more sophisticated control structures. Top-down and bottom-up testing of procedures should also be supported by automating the inclusion of drivers and stubs.

Another important improvement can be made by transporting the system to a microcomputer with an advanced graphics terminal. The window management system should be modified to display more than one view at the same time on the screen.

Translation of IPE-PC programs to the different languages can be automated by using a table-driven approach. This approach reduces translator design for a particular high level language to the table design. Table design can also be automated by designing a table generator from an attribute grammar specification of the target language.

BIBLIOGRAPHY

1. Aho, A.V., Sethi, R., and Ullman, J.D., Compilers Principles,Techniques and Tools, Addison-Wesley Publishing Company, 1986.

2. Albizuri-Romero, M.B., "Internal Representation of Programs in Grase", SIGPLAN Notices, 1985, Vol. 20, No. 8, pp. 41-50.

3. Allison, L., "Syntax Directed Program Editing", Software Practice and Experience, 1983, Vol. 13, pp. 453-465.

4. Anderson, J.R., and Reiser B.J., "The Lisp Tutor", BYTE, 1985, No. 4, pp. 159-75.

5. Arango, G., and Freeman, P., "Modeling Knowledge for Software Development", IEEE, Third International Workshop on Software Specification and Design, 1985, pp. 63-66.

6. Archer, Jr.J., and Conway, R., "COPE: A Cooperative Programming Environment", Technical Report, Dept. of Computer Science, Cornell University, 1981.

7. Atkinson, L.V., and McGregor, J.J., "CONA-A Conversational Algol System", Software Practice and Experience, 1978, Vol. 8, pp. 699-708.

8. Atkinson, L.V., McGregor, J.J., and North, S.D., "Context Sensitive Editing as an Approach to Incremental Compilation", The Computer Journal, 1981, Vol. 24, No. 3, pp. 222-229.

9. Balzer, R. "Transformational Implementation", IEEE Trans. on SE., 1981, Vol. SE-7, No.1, pp. 3-14.

10. Barstow, D., "Artifical Intelligence and Software Engineering", Proceedings of the 9th International Conference on Software Eng., 1987, pp. 200-11.

11. Bhujade, M.R., "Visual Specification of Blocks in Programming Languages", SIGPLAN Notices, 1987, Vol. 22, No. 8, pp. 24-6.

12. Bonar, J., and Well, W., "An Informal Programming Language", Expert Systems in Government Symposium, 1985, pp. 136-44.

13. Boute, R.T., "Building a Uniform Programming Environment Based on Data Abstraction", Proceedings of the 6th ACM European Regional Conference, 1981, pp. 415-24.

14. Brun, G., "The Token-Oriented Approach to Program Editing", Sigplan Notices, 1985, Vol. 20, No. 2, pp. 17-20.

15. Campbell, R.H., and Kirslis, P.A., " The SAGA Project: A System for Software Development", ACM Software Eng. Notes, 1984, Vol. 9, No. 3, pp. 73-80.

16. Celentano, A., Vigna, P.D., and Ghezzi, C., "SIMPLE: A Program Development System", Computer Languages, 1980, Vol. 5, pp. 103-114.

17. Chesi, M., Dameri, E., Franceschi, M.P., Gatti, M.G., and Simonelli, C., "ISDE: An Interactive Software Development Environment", ACM Software Eng. Notes, 1984, Vol. 9, No. 3, pp. 81-88.

18. Christensen, L.C., Stokes, G.E., Hays, B., and Coons, E.D., "TEACH: A Knowledge-Driven Lab Assistant for a Computer-Based Instruction System", Expert Systems in Government Symposium, 1985, pp. 588-95.

19. Cohen, E., "Text-Oriented Structure Commands for Structure Editors", SIGPLAN Notices, 1982, Vol. 17, No. 11, pp. 45-49.

20. Delisle, N.M., Menicosy D.E., and Schwartz, M.D., "Viewing a Programming Environment as a Single Tool", ACM Software Eng. Notes, 1984, Vol. 9, No. 3, pp. 49-56.

21. Donzeau-Gouge, V., Huet, G., Kahn, G., Lang, B., and Levy, J.J., "A Structure-oriented Program Editor: A First Step Towards Computer Assisted Programming", International Computing Symposium, 1975, North Holland Publishing Company, pp. 113-120.

22. Fischer, C.N., Pal, A., and Stock, D.L., "The POE Language-Based Editor Project", ACM Software Eng. Notes, 1984, Vol. 9, No. 3, pp. 21-29.

23. Foisseau, J., Jacquart, R., Lemaitre, M., Lemoine, M., Vignat, J.C., and Zanon, G., "Program Development with or without Coding", IFIP, 1980, pp. 327-330.

24. Fritzson, P., "Preliminary Experience from the DICE System, A Distributed Incremental Compiling Environment", ACM Software Eng. Notes, 1984, Vol. 9, No. 3, pp. 113-123.

25. Ganser, E.R., Horgan, J.R., Moore, D.J., Surko, P.T., Swartwout, D.E., and Reppy, J.H., "SYNED -- A Language-Based Editor for an Interactive Programming Environment", Digest of Papers Spring COMPCON 83, 1983, pp. 406-10.

26. Garlan, D.B., and Miller, P.L., "GNOME: An Introductory Programming Environment Based on a Family of Structure Editors", ACM Software Eng. Notes, 1984, Vol. 9, No. 3, pp. 65-72.

27. Goodwin, L., and Sanati, M., "Learning Computer Programming Through Dynamic Representation of Computer Functioning: evaluation of a new learning package for Pascal", Int. J. Man-Machine Studies, 1986, Vol. 25, pp. 327-341.

28. Habermann, A.N., and Notkin, D., "Gandalf: Software Development Environments", IEEE Trans. on SE, 1986, Vol. SE-12, No.12, pp. 1117-1127.

29. Harandi, M.T., and Young, F.H., "Template Based Specification and Design", IEEE, Third International Workshop on Software Specification and Design, 1985, pp. 94-97.

30. Hausen, H.L., and Mullerburg, M., "Architecture of Software Systems in the Context of Software Engineering Environments", Proceedings of the 6th ACM European Regional Conference, 1981, pp. 147-57.

31. Horgan, J.R., and Moore, D.J., "Techniques for Improving Language Based Editors", ACM Software Eng. Notes, 1984, Vol. 9, No. 3, pp. 7-14.

32. Hunter, R.B., "A System for Writing Pascal Programs Interactively", SIGPLAN Notices, 1984, Vol. 19, No. 1, pp. 46-56.

33. Jensen, K., and Wirth, N., Pascal User Manual and Report, Springer-Verlag, 2nd Edition, 1978

34. Jensen, R.W., and Tonies, C.C., Software Engineering, Prentice-Hall, Inc., 1979.

35. Jonsson, D., "Pancode and Boxcharts: Structured Programming Revised", SIGPLAN Notices, 1987, Vol. 22, No. 8, pp. 89-98.

36. Kaiser, G., and Feiler, P.H., "An Architecture for Intelligent Assistance in Software Development", Proceedings of the 9th International Conf. on Software Eng., 1987, pp. 180-8.

37. Kernighan, B.W., and Ritchie, M.D., The C Programming Language, Prentice-Hall, Inc., 1978.

38. Kramer, B., and Schmidt, H.W., "Interactive Software Development by Stepwise Formalisation", Proceedings of the 6th ACM European Regional Conference, 1981, pp. 134-43.

39. Latour, L.J., "A Programming Environment for Learning SEE: A Student's Educational Environment", IEEE, Ada Application and Environment Conference, 1986, pp. 127-134.

40. Leer, V., "Top-down development using a program design language", IBM SYST J, 1976, No. 2, pp. 155-170.

41. Lewis, J.W., "Beyond ALBE/P: Language Neutral Form", 5th International Conference of Software Eng., 1981, pp. 422-9.

42. Madhauji, N.H., and Choudhury, S., "Beyond a Program Synthesizer", IEEE, Third International Workshop on Software Specification and Design, 1985, pp. 143-6.

43. Mander, K.C., "The Software Developer's Note Pad", IEEE, Third International Workshop on Software Specification and Design, 1985, pp.147-50.

44. Marcus, M., and Sattley K., "DAPSE: A Distributed Ada Programming Support Environment", IEEE Ada Application and Environments Conference, 1986, pp. 115-25.

45. Mora, R.M., and Feiler, P.H., "An Incremental Programming Environment", IEEE trans. on SE, 1981, Vol. SE-7, No.5, pp. 472-482.

46. Morris, J.M., and Schwartz, M.D., "The design of a language-directed editor for block-structured languages", SIGPLAN Notices, 1981, Vol. 16, No. 6, pp. 228-33.

47. Muldner, T., "A CAI Implementation of Pascal", SIGPLAN Notices, 1985, Vol. 20, No. 4, pp. 88-95.

48. Nagl, M., "An Incremental Programming Support Environment", Computer Physics Communications 38, 1985, pp. 245-276.

49. Nakajima, R., Yuasa, T., and Kojima, K., "The Programming System- A Support System for Hierarchical and Modular Programming", Proceedings of the IFIP Congress, 1980, pp. 299-304.

50. Osterweil, L., "Software Environment Research: Directions for the Next Five Years", IEEE, Computer, April 1981, pp. 35-43.

51. Ottenstein, K.J., and Ottenstein, L.M., "The Program Dependence Graph in a Software Development Environment", SIGPLAN Notices, Vol. 19, No. 5, 1984, pp. 177-184.

52. Parker, J., "Towards More Intelligent Programming Environments", ACM SIGSOFT SE Notes, 1985, Vol. 10, No. 3, pp. 28-32.

53. Reiss, S.P., "Graphical Program Development with PECAN Program", ACM Software Eng. Notes, 1984, Vol. 9 , No. 3, pp. 30-41.

54. Reps, T., "Static-Semantic Analysis in Language-Based Editors", Digest of Papers Spring COMPCON 83, 1983, pp. 411-14.

55. Reps, T., and Teitelbaum T., "The Synthesizer Generator", SIGPLAN Notices, Vol. 19, No. 5, 1984, pp. 42-8.

56. Rising, L., "A Syntax-Directed Editor, World-Builder and Simulator for the Language of Karel the Robot", SIGPLAN Notices, 1984, Vol. 19, No. 11, pp. 18-21.

57. Robillard, P.N., "A Software Tool and a Schematic Notation that Improve the Use of Programming Languages", SOFTFAIR II, 1985, pp. 149-58.

58. Robillard, P.N., "Schematic Pseudocode for Program Constructs ant Its Computer Automaton by Schemacode", ACM Communications, 1986, Vol. 29, No. 11, pp. 1072-89.

59. Ross, G., "Integral C-A Practical Programming Environment", SIGPLAN Notices, 1987, Vol. 22, No. 1, pp. 42-48.

60. Rubin, L.F., "Syntax-Directed Pretty Printing: A First Step Towards a Syntax-Directed Editor", COMPSAC 81, 1981, pp. 418-27.

61. Shapiro, E., Collins, G., Johnson, L., and Ruttenberg, J., "PASES: a Programming Environment for PASCAL", SIGPLAN Notices, Vol. 16, No. 8, 1981, pp. 50-7.

62. Standish, T.A., and Taylor, R.N., "Arcturus: A Prototype Advanced Ada Programming Environment", ACM Software Eng. Notes, 1984, Vol. 9, No. 3, pp. 57-64.

63. Steensgaard-Madson, "Module Trees and Software Design", IEEE, Third International Workshop on Software Specification and Design, 1985, pp. 216-17.

64. Steier, D., and Kant, E., "Symbolic Execution as an Aid for Algorithm Design", IEEE, Third International Workshop on Software Specification and Design, 1985, pp. 218-222.

65. Swinehard, D., Zellweger, P., Beach, R., and Hugmann, R., "A Structural View of the Cedar Programming Environment", ACM Transactions on Programming and Systems, 1986, Vol. 8, No. 4, pp. 419-90.

66. Taylor, R.N., Clarke, L., Osterweil, L.J., Wileden, J.C., and Young, M., "Arcadia: A Software Development Environment Research Project", IEEE, Ada Application and Environment Conference, 1986, pp.137-49.

67. Teitelbaum, T., and Reps, T., "The Cornell program synthesizer: a syntax-directed programming environment", Comm. ACM, 1981, Vol. 24, No. 9, pp. 563-573.

68. Templeton, B., and Gardner, J. "The ALICE Programming Education System", ECOO/AEDS Conference Proceedings on 'Computer Knows no Borders', 1985, pp. 85-8.

69. Tomek, I., Muldner, T., and Khan, S., "PMS - A Program to Make Learning Pascal Easier", Comput. Educ. 1985, Vol. 9, No. 4, pp. 205-211.

70. Waters, R.C., "The Programmer's Apprentice: Knowledge Based Program Editing", IEEE Trans. on SE, 1982, Vol. SE-8, No.1, pp. 1-12.

71. Waters, R.C., "Program Editors Should Not Abandon Text Oriented Commands", pp. 39-46. SIGPLAN Notices, Vol. 17, No. 7, 1982, pp. 39-46.

72. Whitelaw, M.W., "Some Ramifications of the Exit Statement in Loop Control", SIGPLAN Notices, 1985, Vol. 20, No. 8, pp. 99-106.

73. Wilcox, T.R., Davis, A.M., and Tindall, M.H., "The design and implementation of a table driven, interactive diagnostic programming system.", Comm. ACM, 1976, Vol. 19, No. 11, pp. 609-616.

74. Zavodnik, R.J., and Middleton, M.D., "YALE the Design of Yet Another Language-Base Editor", SIGPLAN Notices, 1986, Vol. 21, No. 6, pp. 70-77.

75. Zelkowitz, M.V., "A Small Contribution to Editing with a Syntax Directed Editor", ACM Software Eng. Notes, 1984, Vol. 9, No. 3, pp. 1-6.

76. Zhoholev, E.A., "Syntax Directed Program Construction", Trans in: Program. and Comput. Software (USA), 1979, Vol. 5, No. 6, pp. 373-7.

77. Unix User's Manual, Reference Guide, 4.2 Berkeley Software Distribution, Virtual VAX-11 Version, March 1984.

78. Unix Programmer's Manual, Reference Guide, 4.2 Berkeley Software Distribution, Virtual VAX-11 Version, March 1984.

VITA

Nurcan Coskun was born on September 17, 1957 in Malatya, Turkey. She received her elementary, junior high, and high school education in Ankara, Turkey. She received a Bachelor of Science degree in Industrial Engineering from Middle East Technical University (METU) - Ankara, Turkey in November 1979.

She enrolled in the Graduate School of the University of Missouri-Rolla in May 1982 and received a Master of Science degree in Computer Science in May 1983.

APPENDIX A

GRAMMAR

The following table summarizes the meta symbols that are used in the grammar.

Meta Symbol	Meaning
::=	is defined as
< >	nonterminal symbol
{ }	repeated item
[ ]	optional item
*	including the empty element (0 or more)
+	not including the empty element (1 or more)
C	an element of the set of all existing characters or symbols
|	or
inc	increments the indentation level of the following production
dec	decrements the indentation level of the following production
eol	starts a new line with the current indentation.

< program > ::= Main Program : [< text >] [**eol inc** < refinement >]

< procedure > ::= **id** procedure : [< text >] [**eol inc** < refinement >]

< comment > ::= **comsym** < text > [**eol inc** < refinement >]

< text > ::= *C**

< refinement > ::= { < statement > **eol**}+

< statement > ::= < comment > | < IF > | < WHILE > | < UNTIL > | < FOR > |
< asg > | < procedure call > | < return >

< IF > ::= IF < expression > **eol inc**
< refinement > [**dec** < elsepart >]

< elsepart > ::= ELSE **eol inc** < refinement > |
ELSE IF < expression > **eol inc**
< refinement > [**dec** < elsepart >]

< WHILE > ::= WHILE < expression > **eol inc** < refinement >

< asg > ::= < variable > **assignop** < expression >

< UNTIL > ::= REPEAT **eol inc** < refinement >
dec UNTIL < expression >

< FOR > ::= FOR **id assignop** < lb > TO < ub > BY < inc > **eol inc**
< refinement >

< lb > ::= < simple expression >

< ub > ::= < simple expression >

< inc > ::= < simple expression >

< RETURN > ::= return([< expression >])

< procedure call > ::= **id**([< expression-list >])

< expression-list > ::= < expression > | < expression-list > , < expression >

< expression > ::= < simple expression > |
< simple expression > **relop** < simple expression >

< simple expression > ::= < term > | < sign > < term > |
< simple expression > **addop** < term >

< term > ::= < factor > | < term > **mulop** < factor >

< factor > ::= < variable > | < procedure call > | **num** |
< expression > | not < factor >

< sign > ::= + | -

< variable > ::= < entire variable > | < component variable >

< entire variable > ::= < var identifier >

< var identifier > ::= **id**

< component var > ::= < indexed variable > | < field designator >

< indexed variable > ::= < array variable > [< expression-list >]

< array variable > ::= < variable >

< field designer > ::= < record variable > . < field identifier >

< record variable > ::= < variable >

< field identifier > ::= **id**

Lexical Conventions

1) Terminal **id** for identifiers matches a letter followed by letters or digits

letter ::= A|B|...|Z|a|b|...|z

digit ::= 0|1|...|9

id ::= letter (letter|digit)*

Only the first eight characters of the identifiers has been used to distinguish between the identifiers.

2) Terminal **num** matches unsigned integers

num ::= digit (digit)*

3) The relation operators (**relop**'s) are:

'=', '< >', '<', '< =', '>', and '> =' .

4) The **addop**'s are '+', '-', and 'or'.

5) The **mulop**'s are '*', '/', 'div', 'mod', and 'and'.

6) The lexeme for the token **assignop** is : '< -' .

7) The comment symbol, **comsym**, is : '-' .

APPENDIX B

USER MANUAL

Welcome View

The IPE-PC is loaded in UNIX under VAX 11/780. It can be invoked by entering "pgm" command. The Welcome View welcomes the user to the environment. Available commands and their functions are:

(e) Starts an edit session.

First, the system will ask the name of the program. At this point, the user has to enter a program name. If the program exists its internal representation is constructed out of the external representation. Otherwise, a new program structure is constructed. Once the internal representation is built the procedure view comes to the screen.

(d) Brings the directory view to the screen.

(q) Leaves the environment.

(H) Displays help information for the Welcome View.

Algorithm View Commands

The commands that are used often has only one character long name. Names were chosen according to the name of the corresponding template. For example 'w','u', and 'f' commands are used for the while, until and for loop templates respectively. Less frequent commands are two character long. Two character commands are grouped (i.e. members of the same group start with the same character and second character reminds the specifics of the corresponding command). For example, in the 'vt' command, first character 'v' shows that this command belongs to view group and second character 't' stands for type. When the first character of a two character command is typed, menu window displays the members of that group to remind the user of the exact name.

(s) Creates a statement following the current statement.

(S) Creates a statement preceding the current statement.

(w) Expands the current statement as a WHILE loop.

(u) Expands the current statement as an UNTIL loop.

(f) Expands the current statement as a FOR loop.

(i) If the cursor is on a < statement > template, it is expanded as an IF statement.

If the cursor is on < var >, < expression >, < lb >, < ub >, and < inc > placeholders, they are expanded by inserting text. When text is inserted < Del > and < Bs > keys on the keyboard can be used to move the cursor left. < Return > shows the end of insertion.

(E) Adds an ELSE IF part to an existing IF statement.

(e) Adds an ELSE part to an existing IF statement.

(c) Expands the current statement as a comment statement.

(a) Expands the current statement as an assignment statement.

(p) Expands the current statement as a procedure call.

(d) Deletes the current statement and its refinement.

(r) If the cursor is on a comment, a refinement step is created.

If the cursor is on a < statement > template, it is expanded as a return statement.

(j) Expands(joins) a < statement > template by using a clip as follows :

First, it asks for the name of program segment. The tree for that segment is constructed from its external representation and it is inserted to the current position.

(n) Moves the cursor to the beginning of a template following the current line.

(b) Moves the cursor to the beginning of a template preceding the current line.

(h) Moves the cursor to the beginning of a placeholder at the right of the current cursor position.

(g) Moves the cursor to the beginning of a placeholder/template at the left of the current cursor position.

(t) It is used to start a text mode for the modification of an already inserted text. Available commands in the text mode are:

- (n) Moves the cursor to the beginning of a next line, if the text is longer than one line.
- (b) Moves the cursor to the beginning of a preceding line, if the text is longer than one line.
- (h) Moves the cursor one character position to the left.
- (g) Moves the cursor one character position to the right.
- (i) Inserts a new text starting at the current cursor position.
- (d) Deletes the current character.
- (?) Displays the rest of the menu options in the text editor view.

  Since the menu window consists of only one line, it is impossible to display all of the menu items and their short descriptions in one line. By using this command the rest of the menu items are displayed in the menu window. This command is available in every view.
- (q) Ends the text mode.
- (H) Displays help information about the text mode.

(x) Starts single-step execution of the program.

(q) Brings the procedure view to the screen.

(?) Displays the rest of the menu options in the Algorithm View.

(H) Displays help information for the Algorithm View.

(.p) Moves the cursor to the parent.

(.s) Moves the cursor to the sibling.

(.e) Takes the cursor to the next statement with an error.

(mp) Deletes the current statement and its refinement and constructs a procedure out of this deleted segment. The user has to enter a valid procedure name following this command.

(md) Deletes the current statement and its refinement as a clip. The system asks a clip name and saves the segment with that name.

(mc) Copies the current statement and its refinement as a clip.

(mr) Deletes the refinement of the current statement as a clip.

(op) Produces Pascal code for the current program.

(ol) Produces a listing file for the current program.

(oi) Sends the interactive input and output file to the printer.

(vv) Brings the variable view of the current procedure/main program.

(vt) Brings the type window to the screen.

(va) Brings the algorithm view of an existing procedure. The user has to enter a valid procedure name following this command.

(vc) Brings the clip view.

Directory View Commands

(d) Deletes an existing program from the directory. (Name of the program has to be entered by the user following this command).

(c) Copies an existing program to a new file. Names of the existing program and new program are entered by user in a conversational style.

(q) Brings the welcome view to the screen.

(H) Displays help information for the Directory View.

Clip View Commands

This view displays the names of the program segments, clips, to the user.

(d) Deletes an existing clip.

(s) Shows the contents of a clip by bringing the Display View to the screen.

(q) Brings the Algorithm View.

(H) Displays help information for the Clip View.

Clip Display View Commands

(p) If the segment is too long to be displayed on 20 lines available in the window, the rest of the program can be displayed by using this command.

(q) Takes the user to the Clip View.

Execution View Commands

In the single stepping execution mode there are three possible commands:

(Return) Moves the cursor to the next executable statement. The value of the interpreted expression is displayed in the message window.

(v) Brings the Execution View which displays the contents of the variables in the current scope.

(i) Displays the contents of the interactive I/O file.

(t) Switches to the tracing mode execution.

The available operations in this mode are:

(+) Speeds up the execution.

(-) Slows down the execution.

(Return) Switches back to the single-stepping mode execution.

(q) Takes the user to the Algorithm View.

(H) Displays help information for the Execution View.

Symbol Table View Commands

It is used to insert/modify variable declarations of the current procedure.

(n) Moves the cursor to the beginning of the next declaration.

(b) Moves the cursor to the beginning of the preceding declaration.

(g) Moves the cursor right on the current declaration.

(h) Moves the cursor left on the current declaration.

(v) Inserts a new variable definition after the current position.

(V) Inserts a new variable definition before the current position.

(d) Deletes the current variable definition.

(i) Expands the current placeholder by inserting a text.

(I) If the current placeholder is < type >, it is expanded as 'int'.

If the current placeholder is < kind >, it is expanded as 'input'.

(R) If the current placeholder is < type >, it is expanded as 'real'.

(C) If the current placeholder is < type >, it is expanded as 'char'.

(B) If the current placeholder is < type >, it is expanded as 'bool'.

(O) If the current placeholder is < kind >, it is expanded as 'output'.

(L) If the current placeholder is < kind >, it is expanded as 'local'.

(G) If the current placeholder is < kind >, it is expanded as 'global'.

(P) If the current placeholder is < kind >, it is expanded as 'proc'.

(t) Brings the Type View to the screen.

(q) Ends the Variable View and returns to the point of invocation.

(H) Displays help information for the Variable View.

Procedure View Commands

(t) Brings the Type View to the screen.

(v) Brings the Variable View for the current procedure.

(a) Brings the Algorithm view to the screen for a procedure/main program pointed by the cursor.

(n) Moves the cursor to the next procedure name in the window.

(d) Removes the existing procedure.

(c) Copies an existing procedure to build a new one.

(u) Changes(updates) the name of an existing procedure.

(m) Moves the existing procedure to the end of the procedure list.

(i) Displays the interface information for a specified procedure.

(q) Ends the Procedure View and returns the Welcome View.

(H) Displays help information for the Procedure View.

Type View Commands

(a) Expands the current type description placeholder as an array definition.

(s) Expands the current type description placeholder as a structure definition.

(n) Moves the cursor to the beginning of the next type definition.

(b) Moves the cursor to the beginning of the preceding type definition.

(g) Moves the cursor right on the screen.

(h) Moves the cursor left on the screen.

(t) Inserts a new type definition after the current position.

(T) Inserts a new type definition before the current position.

(f) Creates a new field for the current record following the current field position.

(F) Creates a new field for the current record preceding the current field position.

(d) If the cursor is at a field definition, it deletes that field from the structure.

If the cursor is at the first line of a type definition, it deletes that type def..

(i) Expands the current placeholder by starting an insert operation.

(I) If the current placeholder is < type >, it is expanded as 'int'.

(R) If the current placeholder is < type >, it is expanded as 'real'.

(C) If the current placeholder is < type >, it is expanded as 'char'.

(B) If the current placeholder is < type >, it is expanded as 'bool'.

(q) Ends the type view and returns to the invocation point.

(H) Displays help information for the Type View.

APPENDIX C

IPE-PC LISTING FILES

This appendix includes the listing file produced for an example program.

sort.listing:

Type Definitions :

< type id >	< type decription >	< comment >

---

arrtype	array[10]	
	of int	< comment >

Main Program :

Variable Declarations :

< identifier >	< type >	< kind >	< comment >
main	int	proc	< comment >
sorttype	int	local	< comment >
select	int	proc	< comment >
location	int	local	< comment >
bubble1	int	proc	< comment >
qsort	int	proc	< comment >
bubble2	int	proc	< comment >
I	int	local	< comment >
binarysr	int	proc	< comment >
N	int	local	< comment >
DATA	arrtype	local	< comment >
inssort	int	proc	< comment >
test	int	local	used to test binary search

Algorithm:

```
-This pgm implement various sort procedures
 -read the size of data
  read(N)
 -read N integers to sort
  I <- 1
  WHILE I < = N
   read(DATA[I])
   I <- I+1
 -read a flag to choose one of the sort algorithms
  read(sorttype)
 -sort DATA by using the sort algorithm identified by the flag
  IF sorttype = 0
   select(N,DATA)
  ELSE
   IF sorttype = 1
    qsort(N,1,DATA)
   ELSE
    IF sorttype = 2
     bubble1(N,DATA)
    ELSE
     IF sorttype = 3
      bubble2(N,DATA)
     ELSE
      inssort(N,DATA)
 -check if it is sorted
  I <- 1
  WHILE I < = N
   write(DATA[I])
```

```
    I <- I+1
  -Test binary search procedure
    read(test)
    binarysr(N,test,DATA,location)
    write(location)
```

Procedure swap:

Variable Declarations :

< identifier >	< type >	< kind >	< comment >
swap	int	proc	< comment >
MINDEX	int	input	< comment >
I	int	input	< comment >
DATA	arrtype	output	< comment >
temp	int	local	< comment >

Algorithm:

-swaps the contents of I,and MINDEX positions in DATA array

```
temp <- DATA[I]
DATA[I] <- DATA[MINDEX]
DATA[MINDEX] <- temp
```

Procedure select:

Variable Declarations :

<identifier>	<type>	<kind>	<comment>
select	int	proc	<comment>
N	int	input	<comment>
DATA	arrtype	output	<comment>
swap	int	proc	<comment>
MINDEX	int	local	<comment>
I	int	local	<comment>
J	int	local	<comment>

Algorithm:

```
-Sorts array DATA from index 1 through N in order of increasing value
 I <- 1
 -Loop through whole array
  WHILE I <= N
   -Initilize
    MINDEX <- I
    J <- I+1
   -Find index of minimum unsorted element
    WHILE J <= N
     IF DATA[J] < DATA[MINDEX]
      MINDEX <- J
     J <- J+1
   -Swap first unsorted element with minimum unsorted element
    IF MINDEX <> I
     swap(MINDEX,I,DATA)
```

```
I <- I+1
```

Procedure binarysr:

Variable Declarations :

< identifier >	< type >	< kind >	< comment >
binarysr	int	proc	binary search routine
num	int	input	< comment >
keyval	int	input	< comment >
DATA	arrtype	input	< comment >
location	int	output	< comment >
last	int	local	the last position
found	boolean	local	< comment >
first	int	local	< comment >
midpoint	int	local	< comment >

Algorithm:

```
-This procedure does a binary search for the record containing 'keyval'.
 If it is found, its index is returned in 'location' if not, 0 is
 returned
  -Initialize
    found <- false
    first <- 1
    last <- num
  -Search until element found or there are no more elements
    WHILE ((first <= last) and (not found))
      -Compare middle element in search area to the key value
        midpoint <- (first + last) div 2
        IF DATA[midpoint] = keyval
          found <- true
```

```
    ELSE
      -Move first half indexes to cut search area in half
       IF DATA[midpoint] > keyval
         last <- midpoint - 1
       ELSE
         first <- midpoint + 1
-Sets value of 'location'
  IF found
    location <- midpoint
  ELSE
    location <- 0
```

Procedure inssort:

Variable Declarations :

<identifier>	<type>	<kind>	<comment>
inssort	int	proc	<comment>
N	int	input	<comment>
DATA	arrtype	output	<comment>
swap	int	proc	<comment>
placefou	boolean	local	<comment>
I	int	local	<comment>
J	int	local	<comment>

Algorithm:

```
-Sorts the DATA array by using the insertion sort algorithm,it calls
 the swap proc when it is necessary
 I <- 2
 WHILE I <= N
   -Put DATA[I] in its proper place relative to DATA[1]..DATA[I-1]
    placefou <- false
    J <- I
    WHILE ((J > 1) and ( not placefou))
      IF DATA[J] < DATA[J - 1]
        -swap and decrement J
          swap(J,(J-1),DATA)
          J <- J - 1
      ELSE
        placefou <- true
```

I <- I + 1

_____

_____

Procedure split:

Variable Declarations :

< identifier >	< type >	< kind >	< comment >
split	int	proc	< comment >
last	int	input	< comment >
first	int	input	< comment >
split1	int	output	< comment >
split2	int	output	< comment >
DATA	arrtype	output	< comment >
left	int	local	< comment >
more	boolean	local	< comment >
right	int	local	< comment >
swap	int	proc	< comment >
V	int	local	< comment >

Algorithm:

```
-Chooses a splitting value V and arranges DATA so that
 DATA[first]..DATA[sp lit2] < = V and DATA[split1 + 1]..DATA[last] > V
  -Let V be the middle value
    V <- DATA[(first + last) div 2]
    right <- first
    left <- last
  -Rearrange the array
    more <- true
    WHILE more
      WHILE DATA[right] < V
        right <- right + 1
```

```
    WHILE DATA[left] > V
      left <- left - 1
    IF right <= left
      swap(left,right,DATA)
      right <- right + 1
      left <- left - 1
    more <- right <= left
-Set up split1 and split2
  split1 <- right
  split2 <- left
```

Procedure qsort:

Variable Declarations :

<identifier>	<type>	<kind>	<comment>
qsort	int	proc	<comment>
last	int	input	<comment>
first	int	input	<comment>
DATA	arrtype	output	<comment>
split1	int	local	<comment>
split2	int	local	<comment>
split	int	proc	<comment>

Algorithm:

-A recursive procedure for Quicksort

```
IF first < last
  -call split to find split value V and rearrange the DATA array
   accordingly
    split(last,first,split1,split2,DATA) .
  IF split1 < last
    qsort(last,split1,DATA)
  IF first < split2
    qsort(split2,first,DATA)
```

Procedure bubble1:

Variable Declarations :

<identifier>	<type>	<kind>	<comment>
bubble1	int	proc	<comment>
N	int	input	<comment>
DATA	arrtype	output	<comment>
swap	int	proc	<comment>
I	int	local	<comment>
J	int	local	<comment>

Algorithm:

```
-Sorts DATA from index 1 through N in ascending order
 I <- 1
 WHILE I < N
  -Bubble up the smallest unsorted value
   J <- N
   WHILE J > I
    -If the bottom value is smaller than its predecssor, swap them
     IF DATA[J] < DATA[J - 1]
      swap(J, J - 1, DATA)
    J <- J - 1
  I <- I + 1
```

Procedure bubble2:

Variable Declarations :

<identifier>	<type>	<kind>	<comment>
bubble2	int	proc	<comment>
N	int	input	<comment>
DATA	arrtype	output	<comment>
swap	int	proc	<comment>
swapped	boolean	local	<comment>
I	int	local	<comment>
J	int	local	<comment>

Algorithm:

```
-Sorts DATA from index 1 through N in ascending order; stopssorting
 when the array is sorted
 I <- 1
 swapped <- true
 -Loop through array; stop when sorted. The array is sorted when there
  are no values swapped in the inner loop
   WHILE (I < N) and swapped
     -Initialize
       J <- N
       swapped <- false
     -Bubble up the smallest unsorted value
       WHILE J > I
         -If the bottom value is smaller than its predecessor, swap
          them. Note that the swap took place by setting boolean
          flag swapped.
```

```
        IF DATA[J] < DATA[J - 1]
          swapped <- true
          swap(J, J - 1, DATA)
      J <- J - 1
  I <- I + 1
```

APPENDIX D

PASCAL PROGRAMS

This appendix includes Pascal translation of the example program given in Appendix C. This Pascal program is generated by the environment.

```
program sort (input, output) ;
type arrtype = array [0..10] of integer ;

var sorttype : integer ;
    location : integer ;
    I : integer ;
    N : integer ;
    DATA : arrtype ;
    test : integer ;

{-----------------------------------------------------------------}
procedure swap(MINDEX : integer ; I : integer ; var DATA : arrtype) ;
            var temp : integer ;

begin
{swaps the contents of I,and MINDEX positions in DATA array}
  temp := DATA[I] ;
  DATA[I] := DATA[MINDEX] ;
  DATA[MINDEX] := temp
end ;
{-----------------------------------------------------------------}
```

```
procedure select(N :  integer  ; var DATA : arrtype) ;
var MINDEX :  integer  ;
   I :  integer  ;
   J :  integer  ;

begin
{Sorts array DATA from index 1 through N in order of increasing value}
 I := 1 ;
 {Loop through whole array}
  while I < = N do  begin
   {Initilize}
    MINDEX := I ;
    J := I+1 ;
   {Find index of minimum unsorted element}
    while J < = N do  begin
     if DATA[J] < DATA[MINDEX] then  begin
      MINDEX := J
      end ;
     J := J+1
     end ;
   {Swap first unsorted element with minimum unsorted element}
    if MINDEX < > I then  begin
     swap(MINDEX,I,DATA)
     end ;
    I := I+1
   end
end ;
{-----------------------------------------------------------------}
```

```
procedure binarysr(num : integer ; keyval : integer ; DATA : arrtype;
                   var location : integer ) ;
var last : integer ;
    found : boolean ;
    first : integer ;
    midpoint : integer ;

begin
{This procedure does a binary search for the record containing 'keyval'.
 If it is found, its index is returned in 'location' if not, 0 is
 returned}
  {Initialize}
    found := false ;
    first := 1 ;
    last := num ;
  {Search until element found or there are no more elements}
    while ((first < = last) and (not found)) do begin
      {Compare middle element in search area to the key value}
       midpoint := (first + last) div 2 ;
       if DATA[midpoint] = keyval then begin
         found := true
         end
       else begin
         {Move first half indexes to cut search area in half}
          if DATA[midpoint] > keyval then begin
            last := midpoint - 1
            end
          else begin
            first := midpoint + 1
            end ;
```

```
        end ;
      end ;
  {Sets value of 'location'}
    if found then  begin
      location := midpoint
      end
    else  begin
      location := 0
      end ;
end ;
{-------------------------------------------------------------------}
```

```
procedure inssort(N :  integer  ; var DATA : arrtype) ;
var placefou :  boolean  ;
   I :  integer  ;
   J :  integer  ;

begin
{Sorts the DATA array by using the insertion sort algorithm,it calls the
 swap proc when it is necessary}
 I := 2 ;
 while I < = N do  begin
   {Put DATA[I] in its proper place relative to DATA[1]..DATA[I-1]}
    placefou := false ;
    J := I ;
    while ((J  >  1) and ( not placefou)) do  begin
      if DATA[J] < DATA[J - 1] then  begin
        {swap and decrement J}
          swap(J,(J-1),DATA) ;
          J := J - 1
        end
      else  begin
        placefou := true
        end ;
      end ;
   I := I + 1
   end
end ;
{-----------------------------------------------------------------}
```

```
procedure split(last : integer ; first : integer ;
                var split1 : integer ;
                var split2 : integer ; var DATA : arrtype) ;
var left : integer ;
    more : boolean ;
    right : integer ;
    V : integer ;

begin
{Chooses a splitting value V and arranges DATA so that
 DATA[first]..DATA[split2] < = V and DATA[split1 + 1]..DATA[last] > V}
  {Let V be the middle value}
    V := DATA[(first + last) div 2] ;
    right := first ;
    left := last ;
  {Rearrange the array}
    more := true ;
    while more do begin
      while DATA[right] < V do begin
        right := right + 1
        end ;
      while DATA[left] > V do begin
        left := left - 1
        end ;
      if right < = left then begin
        swap(left,right,DATA) ;
        right := right + 1 ;
        left := left - 1
        end ;
      more := right < = left
```

```
      end ;
  {Set up split1 and split2}
    split1 := right ;
    split2 := left
end ;
{-----------------------------------------------------------------}
procedure qsort(last : integer ; first : integer; var DATA : arrtype);
var split1 : integer ;
    split2 : integer ;

begin
{A recursive procedure for Quicksort}
  if first < last then begin
    {call split to find split value V and rearrange the DATA array
     accordingly}
      split(last,first,split1,split2,DATA) ;
    if split1 < last then begin
      qsort(last,split1,DATA)
      end ;
    if first < split2 then begin
      qsort(split2,first,DATA)
      end
    end
end ;
{-----------------------------------------------------------------}
```

```
procedure bubble1(N :  integer  ; var DATA : arrtype) ;
var I :  integer  ;
   J :  integer  ;

begin
{Sorts DATA from index 1 through N in ascending order}
 I := 1 ;
 while I < N do  begin
   {Bubble up the smallest unsorted value}
    J := N ;
    while J > I do  begin
      {If the bottom value is smaller than its predecssor, swap them}
       if DATA[J] < DATA[J - 1] then  begin
         swap(J, J - 1, DATA)
         end ;
      J := J - 1
      end ;
   I := I + 1
   end
end ;
{-----------------------------------------------------------------}
```

```
procedure bubble2(N :  integer  ; var DATA : arrtype) ;
var swapped :  boolean  ;
   I :  integer  ;
   J :  integer  ;

begin
{Sorts DATA from index 1 through N in ascending order; stopssorting when
 the array is sorted}
 I := 1 ;
 swapped := true ;
 {Loop through array; stop when sorted. The array is sorted when there
  are no values swapped in the inner loop}
  while (I < N) and swapped do  begin
    {Initialize}
     J := N ;
     swapped := false ;
    {Bubble up the smallest unsorted value}
     while J > I do  begin
       {If the bottom value is smaller than its predecessor, swap
        them. Note that the swap took place by setting boolean
        flag swapped.}
        if DATA[J] < DATA[J - 1] then  begin
          swapped := true ;
          swap(J, J - 1, DATA)
          end ;
       J := J - 1
       end ;
   I := I + 1
```

```
    end
end ;
{-----------------------------------------------------------------}
```

```
begin
{This pgm implement various sort procedures}
 {read the size of data}
  read(N) ;
 {read N integers to sort}
  I := 1 ;
  while I < = N do begin
   read(DATA[I]) ;
   I := I+1
   end ;
 {read a flag to choose one of the sort algorithms}
  read(sorttype) ;
 {sort DATA by using the sort algorithm identified by the flag}
  if sorttype = 0 then begin
   select(N,DATA)
   end
  else begin
   if sorttype = 1 then begin
    qsort(N,1,DATA)
    end
   else begin
    if sorttype = 2 then begin
     bubble1(N,DATA)
     end
    else begin
     if sorttype = 3 then begin
      bubble2(N,DATA)
      end
     else begin
      inssort(N,DATA)
```

```
            end ;
          end ;
        end ;
      end ;
  {check if it is sorted}
    I := 1 ;
    while I < = N do begin
      write(DATA[I]) ;
      I := I + 1
      end ;
  {Test binary search procedure}
    read(test) ;
    binarysr(N,test,DATA,location) ;
    write(location)
end.
```

APPENDIX E

EXTERNAL REPRESENTATION

This appendix includes the external representation of the example program given in Appendix C. Each program is represented by using three files. The file with 'typ' type contains the external representation of the user defined data types. The file with 'sym' type contains the external representation of the variable declarations. The file with 'tree' type contains the external representation of the algorithm parts of the main program and procedures.

TYPE FILE :

For each data type definition the following information is printed:

- an integer that shows the start of a definition,
- the name of the user defined data type,
- an integer that shows if the data type is an array or a structure definition,
- the number of lines necessary to print this type definition in the type view.
- a flag that shows if this type definition has a comment, and the comment if it exists,
- If the data type is array, then print the dimension and type information for the array elements.
  If the data type for the array elements is also aggregate print its name.
- if the data type is structure, print the field name and type information for each field. Use integer codes to show the beginning and the end of the field definitions.

Print an integer code to show the end of the type definitions when the whole type list is printed.

sort.typ file for the sort program:

```
58
arrtype
 461 2 57
10
41
38
```

SYMBOL FILE :

For each symbol in the symbol table the following information is printed:

- a code that shows the symbol type,
- a code that shows the symbol kind,
- a flag that shows if the symbol has a comment,
- the symbol identifier,
- the comment if it exists,
- if the data type is aggregate, the name of the user defined data type.

Print an integer code to show the end of the file when all the symbols are printed.

sort.sym file for the sort program:

41 44 57

main

41 51 57

sorttype

41 44 57

select

41 51 57

location

41 44 57

bubble1

41 44 57

qsort

41 44 57

bubble2

41 51 57

I

41 44 57

binarysr

41 51 57

N

411 51 57

DATA

arrtype

41 44 57

inssort

41 51 56

test

used to test binary search

58

41 53 57

MINDEX

41 53 57

I

411 54 57

DATA

arrtype

41 44 57

swap

41 51 57

temp

58

41 53 57

N

411 54 57

DATA

arrtype

41 44 57

select

41 44 57

swap

41 51 57

MINDEX

41 51 57

I

41 51 57

J

58

41 53 57

num

41 53 57

keyval

411 53 57

DATA

arrtype

41 54 57

location

41 44 56

binarysr

binary search routine

41 51 56

last

the last position

46 51 57

found

41 51 57

first

41 51 57

midpoint

58

41 53 57

N

411 54 57

DATA

arrtype

41 44 57

inssort

41 44 57

swap

46 51 57

placefou

41 51 57

I

41 51 57

J

58

41 53 57

last

41 53 57

first

41 54 57

split1

41 54 57

split2

411 54 57

DATA

arrtype

41 44 57

split

41 51 57

left

46 51 57

more

41 51 57

right

41 44 57

swap

41 51 57

V

58

41 53 57

last

41 53 57

first

411 54 57

DATA

arrtype

41 44 57

qsort

41 51 57

split1

41 51 57

split2

41 44 57

split

58

41 53 57

N

411 54 57

DATA

arrtype

41 44 57

bubble1

41 44 57

swap

41 51 57

I

41 51 57

J

58

41 53 57

N

411 54 57

DATA

arrtype

41 44 57

bubble2

41 44 57

swap

46 51 57

swapped

41 51 57

I

41 51 57

J

58

TREE FILE :

During the preorder traversal of the abstract syntax tree, print the following information for each node:

- an integer code to show the beginning of a node information.
- the node type, the length of the string in the information field of the node, the contents of the information field.
- if the node has a child, then print an integer code to show the beginning of the child information.
- if the node does not have a child, then if it has a sibling, print an integer code to show the beginning of the sibling information.
- if the node has neither a child nor a sibling, print an integer code to show the end of the children information for the parent.

Print an integer code to show the end of the abstract tree information for the main program or a procedure. The abstract tree information for a procedure starts with its name followed by the node information for each node in the tree.

sort.tree file for the sort program:

```
37 0 42 This pgm implement various sort procedures
34
37 1 21 read the size of data
34
37 1 0
34
37 77 7 read(N)
36
36
35
```

```
37 2 23 read N integers to sort
34
37 1 0
34
37 6 1 I
34
37 7 1 I
36
36
35
37 2 0
34
37 3 6 I < = N
34
37 1 0
34
37 77 13 read(DATA[I])
36
35
37 2 0
34
37 6 1 I
34
37 7 3 I + 1
36
36
36
36
36
35
```

37 2 48 read a flag to choose one of the sort algorithms

34

37 1 0

34

37 77 14 read(sorttype)

36

36

35

37 2 60 sort DATA by using the sort algorithm identified by the flag

34

37 1 0

34

37 4 12 sorttype = 0

34

37 1 0

34

37 77 14 select(N,DATA)

36

36

35

37 5 0

34

37 1 0

34

37 4 12 sorttype = 1

34

37 1 0

34

37 77 15 qsort(N,1,DATA)

36

```
36
35
37 5 0
34
37 1 0
34
37 4 12 sorttype = 2
34
37 1 0
34
37 77 15 bubble1(N,DATA)
36
36
35
37 5 0
34
37 1 0
34
37 4 12 sorttype = 3
34
37 1 0
34
37 77 15 bubble2(N,DATA)
36
36
35
37 5 0
34
37 1 0
34
```

37 77 15 inssort(N,DATA)

36

36

36

36

36

36

36

36

36

36

35

37 2 21 check if it is sorted

34

37 1 0

34

37 6 1 I

34

37 7 1 1

36

36

35

37 2 0

34

37 3 6 I < = N

34

37 1 0

34

37 77 14 write(DATA[I])

36

35

37 2 0

34

37 6 1 I

34

37 7 3 I + 1

36

36

36

36

36

35

37 2 28 Test binary search procedure

34

37 1 0

34

37 77 10 read(test)

36

35

37 2 0

34

37 77 30 binarysr(N,test,DATA,location)

36

35

37 2 0

34

37 77 15 write(location)

36

36

36

```
 38
 39
swap
 37 66 58 swaps the contents of I,and MINDEX positions in DATA array
 34
 37 1 0
 34
 37 6 4 temp
 34
 37 7 7 DATA[I]
 36
 36
 35
 37 2 0
 34
 37 6 7 DATA[I]
 34
 37 7 12 DATA[MINDEX]
 36
 36
 35
 37 2 0
 34
 37 6 12 DATA[MINDEX]
 34
 37 7 4 temp
 36
 36
 36
 38
```

```
 39
select
 37 66 68 Sorts array DATA from index 1 through N in order of increasing
          value
 34
 37 1 0
 34
 37 6 1 I
 34
 37 7 1 1
 36
 36
 35
 37 2 24 Loop through whole array
 34
 37 1 0
 34
 37 3 6 I < = N
 34
 37 1 9 Initilize
 34
 37 1 0
 34
 37 6 6 MINDEX
 34
 37 7 1 I
 36
 36
 35
 37 2 0
```

```
34
37 6 1 J
34
37 7 3 I + 1
36
36
36
35
37 2 38 Find index of minimum unsorted element
34
37 1 0
34
37 3 6 J < = N
34
37 1 0
34
37 4 22 DATA[J] < DATA[MINDEX]
34
37 1 0
34
37 6 6 MINDEX
34
37 7 1 J
36
36
36
36
35
37 2 0
34
```

```
37 6 1 J
34
37 7 3 J + 1
36
36
36
36
36
35
37 2 57 Swap first unsorted element with minimum unsorted element
34
37 1 0
34
37 4 11 MINDEX < > I
34
37 1 0
34
37 77 19 swap(MINDEX,I,DATA)
36
36
36
35
37 2 0
34
37 6 1 I
34
37 7 3 I + 1
36
36
36
```

```
 36
 36
 36
 36
 38
 39
binarysr
 37 66 145 This procedure does a binary search for the record containing
          'keyval'. If it is found, its index is returned in 'locatio
 34
 37 1 10 Initialize
 34
 37 1 0
 34
 37 6 5 found
 34
 37 7 5 false
 36
 36
 35
 37 2 0
 34
 37 6 5 first
 34
 37 7 1 1
 36
 36
 35
 37 2 0
 34
```

37 6 4 last

34

37 7 3 num

36

36

36

35

37 2 56 Search until element found or there are no more elements

34

37 1 0

34

37 3 33 ((first < = last) and (not found))

34

37 1 54 Compare middle element in search area to the key value

34

37 1 0

34

37 6 8 midpoint

34

37 7 20 (first + last) div 2

36

36

35

37 2 0

34

37 4 23 DATA[midpoint] = keyval

34

37 1 0

34

37 6 5 found

```
34
37 7 4 true
36
36
36
35
37 5 0
34
37 1 50 Move first half indexes to cut search area in half
34
37 1 0
34
37 4 23 DATA[midpoint] > keyval
34
37 1 0
34
37 6 4 last
34
37 7 12 midpoint - 1
36
36
36
35
37 5 0
34
37 1 0
34
37 6 5 first
34
37 7 12 midpoint + 1
```

36

36

36

36

36

36

36

36

36

36

36

35

37 2 24 Sets value of 'location'

34

37 1 0

34

37 4 5 found

34

37 1 0

34

37 6 8 location

34

37 7 8 midpoint

36

36

36

35

37 5 0

34

37 1 0

```
34
37 6 8 location
34
37 7 1 0
36
36
36
36
36
36
38
39
inssort
37 66 102 Sorts the DATA array by using the insertion sort algorithm,
        it calls the swap proc when it is necessary
34
37 1 0
34
37 6 1 I
34
37 7 1 2
36
36
35
37 2 0
34
37 3 6 I < = N
34
37 1 62 Put DATA[I] in its proper place relative to DATA[1]..DATA[I-1]
34
```

```
37 1 0
34
37 6 8 placefou
34
37 7 5 false
36
36
35
37 2 0
34
37 6 1 J
34
37 7 1 I
36
36
35
37 2 0
34
37 3 29 ((J > 1) and ( not placefou))
34
37 1 0
34
37 4 21 DATA[J] < DATA[J - 1]
34
37 1 20 swap and decrement J
34
37 1 0
34
37 77 18 swap(J,(J-1),DATA)
36
```

35

37 2 0

34

37 6 1 J

34

37 7 5 J - 1

36

36

36

36

35

37 5 0

34

37 1 0

34

37 6 8 placefou

34

37 7 4 true

36

36

36

36

36

36

36

35

37 2 0

34

37 6 1 I

34

```
37 7 5 I + 1
36
36
36
36
36
38
39
split
37 66 124 Chooses a splitting value V and arranges DATA so that
        DATA[first]..DATA[split2] < = V and DATA[split1 + 1]..DATA[last]
34
37 1 25 Let V be the middle value
34
37 1 0
34
37 6 1 V
34
37 7 26 DATA[(first + last) div 2]
36
36
35
37 2 0
34
37'6 5 right
34
37 7 5 first
36
36
35
```

37 2 0

34

37 6 4 left

34

37 7 4 last

36

36

36

35

37 2 19 Rearrange the array

34

37 1 0

34

37 6 4 more

34

37 7 4 true

36

36

35

37 2 0

34

37 3 4 more

34

37 1 0

34

37 3 15 DATA[right] < V

34

37 1 0

34

37 6 5 right

```
34
37 7 9 right + 1
36
36
36
36
35
37 2 0
34
37 3 14 DATA[left] > V
34
37 1 0
34
37 6 4 left
34
37 7 8 left - 1
36
36
36
36
35
37 2 0
34
37 4 13 right < = left
34
37 1 0
34
37 77 21 swap(left,right,DATA)
36
35
```

```
37 2 0
34
37 6 5 right
34
37 7 9 right + 1
36
36
35
37 2 0
34
37 6 4 left
34
37 7 8 left - 1
36
36
36
36
35
37 2 0
34
37 6 4 more
34
37 7 13 right < = left
36
36
36
36
36
35
37 2 24 Set up split1 and split2
```

```
 34
 37 1 0
 34
 37 6 6 split1
 34
 37 7 5 right
 36
 36
 35
 37 2 0
 34
 37 6 6 split2
 34
 37 7 4 left
 36
 36
 36
 36
 38
 39
qsort
 37 66 35 A recursive procedure for Quicksort
 34
 37 1 0
 34
 37 4 12 first  <  last
 34
 37 1 73 call split to find split value V and rearrange the DATA array
         accordingly
 34
```

```
37 1 0
34
37 77 36 split(last,first,split1,split2,DATA)
36
36
35
37 2 0
34
37 4 13 split1 < last
34
37 1 0
34
37 77 23 qsort(last,split1,DATA)
36
36
36
35
37 2 0
34
37 4 14 first < split2
34
37 1 0
34
37 77 24 qsort(split2,first,DATA)
36
36
36
36
36
36
```

```
38
39
bubble1
37 66 52 Sorts DATA from index 1 through N in ascending order
34
37 1 0
34
37 6 1 I
34
37 7 1 1
36
36
35
37 2 0
34
37 3 5 I < N
34
37 1 37 Bubble up the smallest unsorted value
34
37 1 0
34
37 6 1 J
34
37 7 1 N
36
36
35
37 2 0
34
37 3 5 J > I
```

```
34
37 1 61 If the bottom value is smaller than its predecssor, swap them
34
37 1 0
34
37 4 21 DATA[J] < DATA[J - 1]
34
37 1 0
34
37 77 20 swap(J, J - 1, DATA)
36
36
36
36
35
37 2 0
34
37 6 1 J
34
37 7 5 J - 1
36
36
36
36
36
35
37 2 0
34
37 6 1 I
34
```

```
37 7 5 I + 1
36
36
36
36
36
38
39
bubble2
37 66 91 Sorts DATA from index 1 through N in ascending order;
        stops sorting when the array is sorted
34
37 1 0
34
37 6 1 I
34
37 7 1 1
36
36
35
37 2 0
34
37 6 7 swapped
34
37 7 4 true
36
36
35
37 2 108 Loop through array; stop when sorted. The array is sorted when
        there are no values swapped in the inner loop
```

```
34
37 1 0
34
37 3 19 (I < N) and swapped
34
37 1 10 Initialize
34
37 1 0
34
37 6 1 J
34
37 7 1 N
36
36
35
37 2 0
34
37 6 7 swapped
34
37 7 5 false
36
36
36
35
37 2 37 Bubble up the smallest unsorted value
34
37 1 0
34
37 3 5 J > I
34
```

37 1 126 If the bottom value is smaller than its predecessor, swap them.

Note that the swap took place by setting boolean flag swap

34

37 1 0

34

37 4 21 DATA[J] < DATA[J - 1]

34

37 1 0

34

37 6 7 swapped

34

37 7 4 true

36

36

35

37 2 0

34

37 77 20 swap(J, J - 1, DATA)

36

36

36

36

35

37 2 0

34

37 6 1 J

34

37 7 5 J - 1

36

36

36

36

36

35

37 2 0

34

37 6 1 I

34

37 7 5 I + 1

36

36

36

36

36

36

36

38

38

APPENDIX F

INTERACTIVE INPUT - OUTPUT FILE

This appendix includes the contents of an interactive I/O file which is produced during the execution the sort program given in the Appendix C.

Enter an integer value for variable 'N' = = > 10

Enter an integer value for variable 'DATA[1]' = = > 6

Enter an integer value for variable 'DATA[2]' = = > 1

Enter an integer value for variable 'DATA[3]' = = > 3

Enter an integer value for variable 'DATA[4]' = = > 4

Enter an integer value for variable 'DATA[5]' = = > 11

Enter an integer value for variable 'DATA[6]' = = > 25

Enter an integer value for variable 'DATA[7]' = = > 5

Enter an integer value for variable 'DATA[8]' = = > 30

Enter an integer value for variable 'DATA[9]' = = > 40

Enter an integer value for variable 'DATA[10]' = = > 33

Enter an integer value for variable 'sorttype' = = > 0

DATA[1] = = > 1

DATA[2] = = > 3

DATA[3] = = > 4

DATA[4] = = > 5

DATA[5] = = > 6

DATA[6] = = > 11

DATA[7] = = > 25

DATA[8] = = > 30

DATA[9] = = > 33

DATA[10] = = > 40

Enter an integer value for variable 'test' = = > 11

location = = > 6