Design and Implementation of Interactive Programming Systems

by

Mark William Kahrs

B.S. (University of California, San Diego) 1974

A dissertation submitted in partial satisfaction
of the requirements for the degree of

Master of Science

in

Computer Science

in the

GRADUATE DIVISION

of the

UNIVERSITY OF CALIFORNIA, BERKELEY

Committee in charge:

Michael O'Malley, Chair
Professor Robert Fabry
Dr. L. Peter Deutsch

December 1980

Abstract

Design and Implementation of Interactive Programming Systems

by

Mark William Kahrs

Master of Science in Computer Science

University of California, Berkeley

Michael O'Malley, Chair

The thesis is about the internal design and actual implementation of a system for interactive programming. First, other interactive systems are considered and how they differ from the system described in this thesis. These other systems are also examined for insights they can offer into the design of a programming system.

After examining past interactive programming systems, the design for a new system centered around the construction of a Tree Factored Interpreter/Compiler will be discussed. Each part of the Interactive Programming System will be discussed separately and in detail. The design of a formal semantic language called ISLE will also be discussed. In the conclusion, further research questions are outlined as well as the final conclusions drawn from the implementation experience with the system.

Contents

List of Figures

Acknowledgements

This thesis has been an exercise in delayed gradification. I owe a whole host of thanks (and more) to Peter Deutsch, who not only suggested the topic, but also gave me good advice and kept signing those documents which permitted me access to the facilities of Xerox PARC. Dan Swinehart also provided critical information at critical times; His readings were also superb. Mike O'Malley and Bob Fabry held up the Berkeley end and I thank them both for that. Their patience is appreciated. Rose Peet read the paper in a semi rough form and made it rougher. But if the reading is easier than it once was, it's due to her fine eye for wonderful sounding but empty phrases. Ed Smith has my thanks for printing those figures which were somehow placed in his directory. And lastly, my parents somehow waited for this to be "Signed, sealed and delivered". They've always encouraged me to finish it up and "get it out of the way". And they always wonder where the next one is too.

Chapter 1

Introduction

This thesis is about the internal design and actual implementation of a system for interactive programming. First, other interactive systems are considered and how they differ from the system described in this thesis. These other systems are also examined for insights they can offer into the design of a programming system.

After examining past interactive systems, this thesis will then lay out the plan for a new system, which is centered around the construction of a Tree Factored Interpreter. Each part of the Interactive Programming System will be discussed in detail. An extensive review or attempt to categorize previous interactive systems is not included. [Swinehart, 1974] has such a review in Chapter 2.

1.1 History

The development of the Interactive Programming System began in earnest with the invention of time-sharing. Languages such as LISP [McCarthy, et al., 1963], which was developed before time-sharing, became prominent in the world of interactive computing. Its survival today is due in part to its flexibility. LISP is one of the most flexible of interactive systems due to the ability to define functions which assist the user. It is also possible to change the interpreter easily. Later LISP systems included a compiler for functions the

user considered to be debugged. It is important to note here that the decision to compile is always made by the user. Note also that whole functions can be compiled, but partial compilation is not permitted.

JOSS [Baker, 1963] is representative of a class of interactive programming systems including BASIC and interpreted FORTRAN [Ryan, et al. 1966][Katzan, 1969]. JOSS was implemented as an interpreter which was not very flexible and not very efficient. The reason for its inefficiency was that the source text had to be rescanned and reinterpreted every time control passed through that section of text. However, it did present a new programming environment. It had the ability to trace procedure calls as well as to stop a running program, modify it, and continue. JOSS also included the ability to jump to any line in a procedure while at top level. An incremental BASIC system was written for the B5500 that combined both interpreted and compiled code [Braden and Wulf, 1968] but it did little to improve the runtime efficiency of the interpreter.

APL [Iverson, 1965] also introduced a new user interface in which program states that occur at a program break may be suspended. break. APL also allows the user to change variable values and variable types at a program break and proceed with execution. In general, APL's implementation resembles text interpreters such as JOSS. Although APL compilers are a recent development [Perlis, 1977][Guibas and Wyatt, 1978], an incremental APL does not exist.

LC^2 [Mitchell, et al, 1969] was an interactive ALGOL system. It had many desirable qualities that Mitchell integrated into his PhD thesis [Mitchell, 1970]. LC^2 included the ability to execute statements immediately as well as perform sophisticated actions on the control structure of the program. LC^2 also had a dynamic type structure that allowed a variety of types of variables during the execution of the program. There have been other attempts at creating an interactive ALGOL-like language. Both [Lock, 1965, 1968] and [Peccoud, et al, 1968] discuss systems for incremental compilation. These systems had the limitation that a line must be a statement. The system to be discussed in this thesis does not have this restriction.

SMALLTALK [Goldberg and Kay, 1976][Ingalls, 1978] is a descendant of Kay's "Reactive engine" (called Flex) which was discussed in his thesis [Kay, 1969]. It has many, if not all, of the qualities of LISP. Its present compiler implementations are not incremental. Like LISP, it is extremely flexible and not too efficient. What SMALLTALK loses in efficiency however, it makes up in the user interface. The SMALLTALK system provides multiple "windows" that permit the user to display a number of running parallel processes simultaneously. At a program break, the user is allowed to examine variables and to proceed if possible. The "spectrum of computing" runs from interpretation to compilation. The system discussed in this thesis traverses the field. If the IPS discussed in this thesis were placed on the spectrum of "flexibility" or "efficiency", then it would be found that the system can either be as efficient as compiled PASCAL or as inefficient as an interpreted version of BASIC. Its flexibility would be between LISP and APL.

1.2 Definitions of Flexibility

In his thesis, Mitchell defines flexibility to be

> "the ease with which the user of an Interactive Programming System can manipulate the objects of interest to him" (pp. 1-1A)

Mitchell's universe of "interesting objects" includes programs, data and control structures. It is difficult to formalize the notion of ease of manipulation for this huge universe. On the other hand, what operations are not easy to perform or get in the way of writing the program? An example of such an operation is editing at a program break. Suppose a program breaks because it calls a procedure that is not yet written. The user should be able to write the procedure that was called on the spot. Then, the user should be able to proceed where the procedure was called. There are many other measures of flexibility and no doubt each reader of this thesis will have his or her own.

1.3 Definitions of efficiency

Mitchell defines efficiency to be

> "the ease and grace with which the system can perform a task, alternately the amount of overhead in an Interactive Programming System is an inverse measure of efficiency" (pp. 1-1A)

The principal efficiency difference between interpreters and compilers is that interpreters must call semantic routines every time a statement is executed. In a compiler, the semantic routines are performed just once at compile time. This distinction will become important when the notion of factoring is introduced.

1.4 Goals of the thesis

Almost every programmer has tried, at one time or another, to construct a program from "whole cloth" (i.e., without any planning). [Schwartz, 1976] writes about constructing programs from "rubble" and using an optimizing compiler to produce efficient code. One reason for exploring interactive programming systems is because they offer a chance to explore "fragment programming". Because most programming systems do not allow a programmer to recover from their bugs in an easy manner, this style of programming is discouraged. It should be noted that it is not clear whether this would be a good method of programming, but it would be interesting to see how well programmers would react to an environment that encouraged, rather than prohibited such behavior. One of the experiments of this thesis is to test the idea of building programs from "fragments" by debugging and enlarging the program as more ideas occur to the author.

Of course, this mode of programming is very similar to what happens during debugging. Therefore, a system which can aid fragment programming can also make debugging easier.

The question remains: What is needed to encourage this type of programming? Here are some suggestions:

- Allow dynamic retyping of variables

- Allow the creation of new code that completes more of the program after a program break (and permit execution to continue).
- Allow direct execution of statements at any program break.
- Increase efficiency to the point where it is feasible to design, compile (interpret) and debug programs entirely within the system.

1.5 The implementation

This thesis describes a system written in BCPL [Richards, 1969] for an experimental personal minicomputer known as the Alto [Thacker, et al., 1980]. This small computer runs a small operating system [Lampson and Sproull, 1979] which features multiprocessing and streams [Stoy and Strachey, 1969]. An Alto hardware includes a 808 by 606 line raster scan display, a pointing device called a mouse [English, et al., 1967] and a keyboard.

The source code occupies some 40,000 words excluding the semantic routines. The text occupies approximately 4000 lines.

1.6 Thesis organization

This thesis is organized into separate chapters for each of the major parts of the system.

The second chapter discusses the line oriented display editor. The command structure for both the mouse and the keyboard is detailed. The internal structure of the editor is described in the next section.

The third chapter discusses the scanner, the parser and the symbol table handler. First, the scanner is examined. The next section details the construction of the Production Language parser and the its debugger. It also contains the details on incremental parsing. The last section discusses the symbol table handler and the effects of a block structured modular language on the organization of the symbol table.

The fourth chapter discusses the interpreter. If semantic analysis routines are separated

from code generators, then the system is called "factored". The core of the system is designed around a interpreter driven by a parse tree walk. Such a system was called by Mitchell a "Tree Factored Interpreter". The chapter introduces the notion of a Tree Factored Partial Compiler and relates it to the Tree Factored Interpreter.

The fifth chapter first discusses the writing of semantics routines. Next, the design of a formal semantics language called ISLE is given. The chapter concludes with a discussion of writing semantic routines in ISLE.

Lastly, the thesis concludes with a discussion of results and some directions for further research into interactive programming systems.

Chapter 2

The Editor

2.1 Introduction

The editor to be discussed in this chapter was designed around three concepts:

1. The mouse would be used to thumb through text
2. The mouse would be used to mark lines
3. Editing commands would only work on marked lines

These concepts are a mixture of ideas from several editors. The idea of using marked lines in an edit command was used in a version of QED for the SDS-940 [Deutsch, 1967].

Most editors today are line oriented i.e., most of the positioning commands deal with lines (generally through line numbers) rather than tokens such as words. This is not surprising, since the primary "cognitive unit" in programming is the line. In the next section, "structured editors" are discussed. These editors are classified as "structured" because the editor has some knowledge about the program structure and uses this knowledge to assist the user in editing.

2.2 History of editors past

One of the more common uses of a display is as an editor. One such early display editor was the TVEDIT system, developed at Stanford University. E [Frost, 1978] is an example of a TVEDIT derivative. This early editor used lines as the primary editing unit. The editor did not know anything about the format of the files it was editing (except, of course about the use of carriage returns as the line terminator character).

One of the first structured editors (and perhaps the most famous) is the NLS editor, developed by [Englebart et al., 1968]. This editor was used in conjunction with a block structured language known as MOL [Hay and Rulifson, 1968] NLS kept formatting information in a file which was used to maintaining the "outline" organization of the editor. Each level has a label in outline form (for example, 1A2b3). One of the problems with NLS was that it required significant time to convert from the internal form to a readable form (such as pure text). This made the transition from editor to programming system slow and inefficient. The editor for the IPS was designed with this criteria in mind: The transition from the editor to the programming system should be smooth and efficient. Therefore, a general purpose line oriented editor was implemented. This was also done intentionally to introduce problems for the Tree Factored Interpreter.

Structured editors are common in LISP systems. An example of such an editor is found in the InterLisp system [Teitelman, 1978]. The LISP editor knows the structure of LISP programs. This is made easier by the fact that most LISP programs are tree-like (and represented by trees), and therefore, the editor is mostly a tree walker.

2.3 The editor

2.3.1 How the editor works

The editor is founded on the principal of "batching" lines. "Batching" is defined as gathering text lines together and then performing some action on the collection of lines. In

order to collect ("batch") a series of lines, the user "marks" those lines with the mouse. Then when a keyboard command is given, the editor looks for those lines which are marked. It will then call the routine the user specified for each marked line. The mouse is also responsible for the position of the text in the window.

2.3.2 The editor command structure

The command structure of the editor can be divided into two parts: what the user can do with the mouse and what the user can do with the keyboard. The Alto has a tracking cursor in which the system can place small images. Upon starting the IPS an ear is shown. This is because the IPS is "listening". There are two dark windows. These are the status windows. The top one is the main status window and the bottom one is the editor status window. Like the SRI mouse, the mouse has three buttons. For future reference, the buttons will be called "up", "down" and "middle".

Mouse commands

It is impossible to depress two keys on the mouse at the same time and record the fact that the two keys were depressed simultaneously. Therefore, regardless of which keys are depressed, when a key is released, the current depressed keys are recorded. Whenever a text line is displayed, it is displayed offset from the left margin by a small amount. When the mouse is inside this margin, it is in "jump" mode. When it is to the right of the margin, it is in "mark" mode. While to the left of the text, the following modes are then possible:

Button	Action
up:	Move this line to the top of the window
down:	Move this line to the bottom of the window
middle:	Jump mode; depends on next button push

up:	Pop line from line pointer stack
down:	Push line on line pointer stack
middle:	Jump absolute

When the mouse is in "mark mode", the following possible actions are possible:

Button	Action
up:	Mark this line
down:	Remove the mark from this line
middle:	Immediately perform action on this line only

When a line is marked, the background of the line is inverted from white to black.

Commands given by the keyboard

The following commands can be typed to direct the editor to perform some action on all marked lines:

- Append Appends new text lines after the line currently pointed to.
- Delete Deletes the line currently pointed to.
- Get Accepts the name of a file, attempts to open it and read it in before the line.
- Insert Inserts new text lines before the current line.
- Jump Jump to the next line which is marked (don't jump if no lines are marked)
- Mark Mark all inserted and modified lines
- Put Accepts the name of a file, attempts to open it and then writes out all marked lines.
- Reset Marks Removes all the marks from all marked lines.
- Substitute Accepts the search string and the replacement string once the search string is found.
- Undo Insert the line (which is expected to be deleted) before the line to which it is attached.

The next commands perform a different task. They are used to control the various display modes and to toggle internal modes. Those commands which set modes can be used to reset that mode by repeating the command.

- ? status window. The user is expected to confirm with a carriage return or DEL to abort.
- Extend Extend the marking to all lines in the window.
- Find Accepts a string and marks all those lines which contains at least one occurrence of it.
- Only Cause the display to show marked lines only.
- Quit Quit from the IPS
- Top Put the top line at the top of the window
- Visible Make deleted lines visible by displaying them in boldface
- Wipe Line Stack Reset the stack used to hold line markers
- Zipping Mode Always jump the screen to the current line being operated on.

Updating the display

The editor always displays 12 lines on the screen. The user can use either the mouse or a keyboard command to position text on the screen. To speed human interaction, the editor must have a fast method of updating the screen. Because the user is mainly interested in what is visible on the screen, a "line cache" of pointers to the displayed lines is kept. The pointers are not direct pointers to the text of the line but rather pointers to a structure called "line heads". Each line head has forward and backward pointers to other line heads as well as pointers to other structures and miscellaneous flags, not to mention the body of the text. The display updating algorithm is given in Appendix A.

The first line displayed is used as a reference and is used to display the next 12 lines. Implementing the mouse actions are then very easy. The code for mouse actions is also given in Appendix A.

Note that the jump mode uses the number of lines (called `lineCount`). Keeping track of the number of lines is easy, provided the system has only one create/destroy line procedure.

The next commands which can be performed during "mark mode" (when the mouse is in the text range) are:

up:	Mark the line pointed at.
down:	Remove the mark from the line pointed at.
middle:	Immediately perform a command on the line pointed at.

Performing commands instantaneously

Often during editing, the user wants to perform a command instantaneously on a line. Rather than push the line on a stack and go back to it after the command in progress, an immediate command was added to the system. This allowed the user to perform actions on single lines spontaneously. It was implemented by marking the line pointed at and then performing the command on the line pointed at.

Undoing changes

Until the TFI is run, deleted lines are kept. They can be displayed with the visible command and are displayed in a bold font. Old lines are attached to the "old" field in each line head. When a line is deleted, it is attached to the chain of old lines attached to the previous line. An obvious reason for this is that if one deletes all the lines, there must be a place to attach these lines to! The EOF is the only line left. (since the user can't delete it).

Since the deleted lines are kept until the TFI is run, it is possible to have an "undo" command. The undo algorithm is found in Appendix A.

The Editor line structure

Each line has three states: modified, inserted and deleted. They are mutually exclusive. A modified line is one in which a edit has been made to the text string (currently only made by the substitute command). Figure 2.3.2 shows the editor line structure when 3 lines have been inserted, another deleted and another modified.

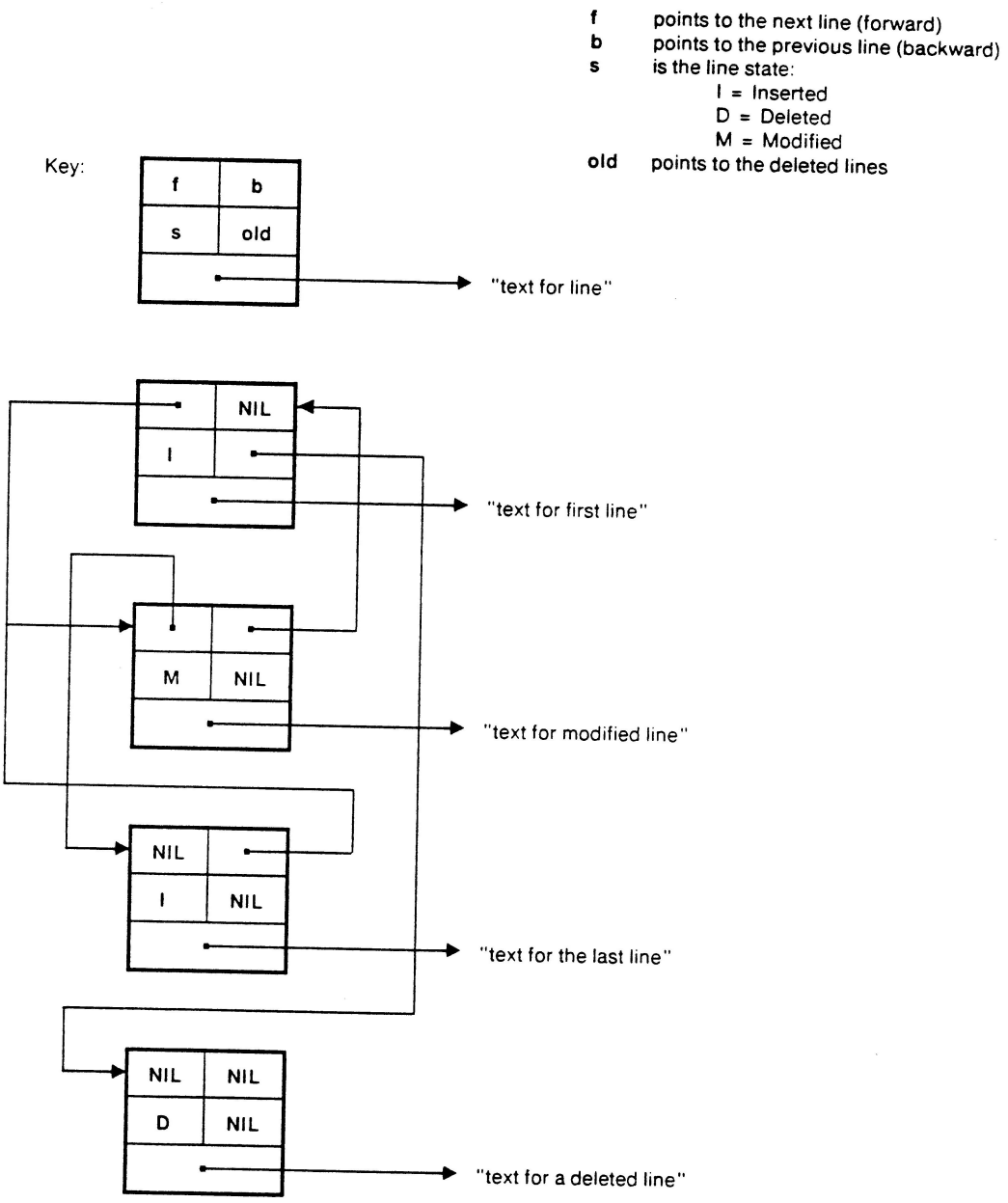

Figure 2.1. Editor Line structure

2.4 Results

The editor works surprisingly well. It has been an interesting editor to use. Not enough time has been spent with it to make any definite conclusions about the concept of marking lines and "batching" commands. An interesting side effect is the interaction with the parser in regard to syntax errors. When a syntax error is discovered, the parser can find out which line is responsible. This line can then be marked by a parse-time semantic routine which is called by the parser error routine. When the parsing process is over, all of the bad lines are marked for easy identification (the bad lines can be found by "jumping" from one bad line to another) and correction.

Another result of the experimentation with this editor is the disposition of the EOF line. The EOF line is special in that it can never be deleted. When the user marks a line, the count of marked lines is incremented. The EOF line is not counted. If the number of marks is 0, then the EOF line is marked; otherwise, the mark on the EOF line is removed. The point is that there is always at least one line with a mark. This feature evolved after preliminary trials with the editor revealed that the user was always using the EOF line for insertion of new text and referring to deleted text.

Another heuristic is that the top and bottom (EOF) line pointers are kept in special variables. Once again, this is because experience showed that these pointers were popular enough to warrant keeping them around for easy access.

In retrospect, there should have been a better method of stacking lines to which the user would like to return. One idea is to display a label (such as a letter or a number) in the left hand margin of the line pointed to by the mouse. Then, when the jump occurs, a symbolic label could be used. Another idea is to display the first 30 characters of the line as the mouse is used to position back and forth in the file. Then, when a familiar line is seen, the middle button is released and the jump occurs.

Chapter 3

The Scanner, Parser And Symbol Table Handler

3.1 Introduction

The system now has a source of text from the editor and must convert the source text into an internal representation for the Tree Factored Interpreter to use. The scanner and the symbol table handler are conventional. Because scanners tend to be static, no scanner generator was constructed [Gries, 1971, chapter 2][Johnson, et al., 1967]. For completeness, the details on the scanner and the symbol table handler are included in this chapter.

3.2 The Scanner

The scanner is described as a finite state machine. The scanner has only one unusual aspect: there does not exist a convenient way of pulling text from these lines because of the line heads. Rather than writing a coroutine, the system uses the "fast stream" mechanisms available through [1]he term "fast" is derived from the fact the other streams deal with peripheral devices; This stream uses memory as the "device". the auspices of the Alto

[1]T

Operating System. The fast stream acts as an interface between the editor lines and the characters which the scanner expects. A fast stream is a stream designed to get bytes from a buffer until the buffer is empty, at which point an "update" routine is called which refills the buffer. The exact mechanism of the fast stream interface is as follows: First, the top line is used to set up the character pointer for subsequent scans. When the text for this line is at the end (detected by causing the fast stream to have its End Of File (EOF) flag on), another line is used for the character pointer. This continues, getting new pointers (by following next links) until the EOF line is reached. Swinehart (see chapter 8, pp. 139) stored the tokens, complete with spacing information after scanning them. Then, he would reconstruct the program from this data. This method was not chosen because it was felt that the benefits derived from saving the tokens would not be worth the extra overhead in converting back and forth between text and tokens.

3.3 The Parser

3.3.1 Statement parsing and parser operation

The technology of parser construction is well understood [Aho and Ullmann, 1972, 1977]. In deciding which parser to choose, the following considerations were made:

- Easy error recovery should be possible
- The parser should be able to call semantic routines at various parser states.
- The parser should have the ability to restart the parse.
- The grammar should be easy to write and easy to debug.

Initially, the parser was implemented in the most popular way: recursive descent implemented in a higher level language. This famous method is still being used by compiler writers everywhere. Burroughs has used it to write their ALGOL compilers since the B5000. Unfortunately, this parser proved to be inflexible and difficult to debug. Errors in the grammar were time consuming to correct because recompilation was necessary. Debugging was

made difficult by inability to "single step" the parser through the grammar. Because of this, the decision was made to rewrite the parser as a table driven Floyd-Evans [Floyd, 1961][Gries, Chapter 7] production language parser. At first, the method of implementation of the parser was as a stack machine with a small pattern matching instruction set (See [Beals, 1969]). This proved to be difficult because the limited word length of the Alto did not enable the one word instructions to have long enough addresses. The solution was to create a pattern matcher which matched symbols in the production table against symbols on the stack. This approach was used in the implementation of SAIL [Reiser, 1976] Each symbol is represented by an absolute number, beginning with the tokens (number, string, identifier), then the reserved words and finally the operators. The semantic routine code has a format very similar to that of most machines: `<opcode> <address>`. The opcode can be any one of 8 possible codes. The various opcodes are:

0 ERROR `<address>` is error number. if the error number is 0 then `HALT` with a successful parse.

1 BRANCH `<address>` is the parse table address to branch to

2 CALL `<address>` is the address of a parse table routine to call (as in a procedure call without parameters)

3 RETURN Return from the procedure

4 NODE Create a parse tree node with a name `<address>`. If the node number is 0, then this is `BASE` for an n-ary node.

5 EXEC Call semantic routine number `<address>`

6 SCAN Get a new token from the scanner

7 ROOT Create an n-ary parse tree node with a name `<address>`

When the parser was first constructed, a debugger was also coded for debugging the grammar. The debugger was modeled after the one used in the construction of SAIL. This

debugger has also been extremely valuable. In the debugger, the following actions were provided:

. Prints out the name of the current address in the parse table Prints out as a label)

↑ Set the temporary address to the address of the next production

B Set a breakpoint at this production by placing a halt in the code. Remove the previous breakpoint (Only one is allowed)

C Print out the parser call stack

D Call the BCPL debugger

E Start semantic routine stepping mode. This will halt the parser before each semantic routine.

L List the production.

M Toggle the production stepping mode. If it is on, then the parser will pause before each production.

N Toggle node tracing switch. Whenever a node is created, the name of the mode will be printed out.

P Proceed to the next instruction. If the parser has exec routine stepping on, then the parser will break again before the next instruction.

Q Quit to the Operating System

R Return from the parse routine currently active.

S Display the symbol stack

T Display the current contents of the node stack.

The debugger can be activated in one of two ways: First, after any parse error, second, during stepping mode, the parser continually breaks back to the debugger. One of the

principal reasons the debugger has proven so valuable is because the parse table generator creates symbol tables for both the symbols and the labels of the productions.

Because the number of child nodes is not known at the time a parse tree n-ary node is created, it is not possible to pre-allocate the node. In order to create a parse tree node with an arbitrary number of branches, the system must keep the newly created children nodes on a stack until the last child node is created. At that point, the child nodes can be copied into a new parse tree node. For example, in a compound statement, the parser has no idea of exactly how many statements will comprise the children of that node. So, when the parser reads a BEGIN, the current node stack pointer is pushed onto a stack called the node base stack. It is called the base stack because the pointer to the parse stack forms the base of a node. Parsing then continues and, of course, other constructs may push more pointers on the node base stack. When an END is reached, the node base pointer is popped from the stack and a node is created with (current node stack pointer-top of node base stack) nodes. This new node is then pushed on the node stack much like any other node. In order to distinquish between n-ary nodes and single nodes, there are two syntactic forms: NODE and ROOT.

Because of the fear of parse table overflow, a large address for the parsing tables (16-3 [opcode length] = 13 bits) was provided. The tradeoff is that there are now a limited number of match symbols (the same number as opcodes, 8) because of the packing of the production tables.

Although production language parsers are most often used to create bottom up parsers, a top-down parser was constructed. Because the state of the parser is contained in the control stack of the parser, it is more difficult to perform error recovery. However, it is possible to get around this problem by using a well designed grammar and by the introduction of various tricks.

3.4 Incremental Parsing

Incremental parsing has long been a thorn in the side of interactive language implementers. The questions of where to start parsing as well as where to stop are difficult to answer. [Lindstom, 1970] gives an algorithm for constructing incremental parsers for LR(K) parsers. Basically, the method used is to find the forest of possible trees at any given state and to augment the grammar by adding productions which would allow the parser to start at any state. A "two level" parser [Earley and Caizergues, 1972] was used in the implementation of VERS. A two level parser uses a second parser on top of the basic statement parser. The principal use of the top level parser is to handle statements that are split between two lines. For example, the statement

```
IF a = b THEN
C := 1
ELSE
C := 2
ENDIF
```

(from Earley and Caizergues, pp. 1041)

would be parsed as (represented line by line)

```
<if bracket> -> <if clause> CR IF <a> THEN
                <statement> <assignment statemen>
<else bracket> -> ELSE CR
                  <statement -> <assignment statement>
<endif bracket> -> ENDIF CR
```

Note that the carriage returns are represented as terminal symbols in the grammar. The left hand sides of the productions are then recognized by the production

```
Statement -> <if bracket> <statement> <else bracket>
             <statement> <endif bracket>
```

The system now requires another parser to use the second grammar to perform the syntax check of the lines using the "global grammar". It also distinquishes between multiline IFs and single line IFs. The counterargument to this is that ALGOL-60 does so also, only it used the word BEGIN.

In this system, the parser is used to help identify which lines could be used to start a parse. For example, a statement such as `IF <B> THEN ...` could be used to start parsing. A line containing the statement `<B> THEN` could not.

The parser starts by restoring the lexical environment of the statement in error. This is done by backing up through the lines detecting `BEGIN` statements. Each line with a `BEGIN/END` has a lexical level associated with it as well as a pointer to the all the identifiers declared at that level. If the block was lexically active, the pointer to the identifier list is put onto the lexical level stack and the search proceeds for the next previous level. This process terminates when all the levels have been restored. The parser stops either when there are no new lines (lines having an inserted state) or when the statement is over. Semantic routines are placed in the parse table to control the action of the parser. In the grammar a semantic routine named `StmntExec` is used to mark the line as being able to start a parse and `HaltExec` is called when a statements ends. The reparsing can stop when all of the following conditions are true:

1. No declarations have been detected
2. No new `BEGINs/ENDs` were found
3. The current line being scanned is unchanged
4. The parser is back up at statement level

After reparsing, the new node is attached to the tree at the highest level found to be incorrect.

Note that a simpler approach would be to reparse entire procedures rather than lines. This saves the implementation complexity of searching for lines which begin a statement. It could also allow intra-procedure code optimization to take place.

3.5 The Symbol Table Handler

3.5.1 Implementation of the symbol table

Mitchell divides symbol table handlers into two categories: scope-oriented and name-oriented. In a scope-oriented symbol table there is a tree whose branches are formed whenever a new lexical level is entered. A name-oriented symbol table has lists of instances which are formed for each declaration of a symbol. The LISP A-list is an example of a scope-oriented symbol tables, whereas most ALGOL compilers use a name-oriented symbol table.

Each of these symbol table structures has its own advantages. In particular, the scope-oriented structure allows enumeration of all instances in a given scope while a name-oriented structure allows the system to find an entry given its name and lexical scope.

Unless implemented carefully, the variable search time in a scope-oriented symbol table would be intolerable.

The symbol table is implemented as a name-oriented scatter table (better known as a hash table) with conflict lists. As such, the symbol table is quite conventional, with the possible exception of the module field and the ring of dependencies (making up the dependency set).

The module field is used to implement the encapsulation facilities which have come into increasingly common use in programming languages such as EUCLID [Lampson et al., 1977], CLU [Liskov et al., 1977, 1979], and MODULA [Wirth, 1977]. Each time a new module is entered (by using the MODULE statement), the current pointer to the module level is pushed on the module level stack and the current module number incremented. Variables are "imported" by copying a variable into the symbol table with a new module field. Variables are "exported" in a similar manner.

The dependency list is used to locate statements which contain that variable. This is used to invalidate statements (and hence the code they generate) as a result of dynamic

type changes. There is another problem that has not been considered: what happens when the user inserts or deletes a `BEGIN` or `END` ?

3.5.2 Incremental changes to block structure

In order to determine which lines have `BEGIN`s and `END` s in them, the semantic routines are used once again. When either a `BEGIN` or an `END` is found, the compiler jumps to a corresponding semantic routine, `BeginExec` / `EndExec`. These routines will set either the beginBit or the endBit in the line head. Note that using a single bit for beginBit/endBit restricts lines to have only one `BEGIN/END` statement. This restriction could be removed by placing a linked list of lexical level pointers in the line head of a line with multiple `BEGIN`s/`END` s. The actions for insertion of a `BEGIN` or `END` are as follows:

- Insertion of a `BEGIN`: All declarations lexically below where the `BEGIN` was inserted must be found and the lexical levels of all declarations must be incremented by one. All variables with lexical levels lower than the inserted `BEGIN` must be checked to ensure that they refer to the right variable in the right block.

- Insertion of an `END` . Once again, it is necessary to go through the entire program after the insertion, changing all identifiers and declarations which were contained within the lexical level.

- Deletion of a `BEGIN`: The symbols declared below the line where the deletion occurred must be "moved up" by reducing the lexical level of identifiers.

- Deletion of an `END` : it is again necessary to go through all the identifiers and see if the lexical level of the identifiers found below the `END` are equal to or less than that of the `END` . If so, then this identifier is a candidate for a scope change. This would be the case if opening a lexical block would cause another variable to reference a variable with the same name in the higher block.

Chapter 4

The Tree Factored Interpreter and Compiler

4.1 Introduction

In an interpreter, the text of the program is parsed, an internal representation generated and then executed immediately. If the internal representation is the parse tree generated by parsing the text, then the interpreter becomes a Tree Interpreter or TI.

Semantic actions can be decomposed into two classes: semantic analysis and code generation (denoted S-action and X-action by Mitchell). Let a further distinction be that no S-action can perform any X-action and vice versa. That is, no semantic action can effect either program or data and no code generator can effect semantics. This separation is commonly found in compilers, but is uncommon in interpreters. An interpreter with separate S-actions and X-actions will be called a Tree Factored Interpreter or TFI. If the parse tree is used to compile a program by following all the branches (not only those on the control path), then this is called a Tree Factored Compiler or TFC. There are two classes of TFCs: If only the subtrees of the parse tree which were changed are compiled, then the TFC will be called a Tree Factored Incremental Compiler (TFIC) or Tree Factored Partial Com-

piler (TFPC). Otherwise, the TFC will be called a Tree Factored Total Compiler (TFTC). Clearly, a TFTC is another name for a "standard" compiler.

Once a statement has been interpreted, the code can be saved and used again. However, if the semantics change, then the X-actions are invalid and the S-actions must be be reinterpreted. When this technique is extended to parse trees, then the following changes must be made: when the semantics associated with a parse tree node are changed then the code that is generated by that node must be changed. Both semantic information and code "float up" from leaf nodes to their parent nodes. In a TFPC, these nodes continue to bubble information up until the root node has been reached and all the code is ready for execution. A TFI does not need to have all the code ready for execution, just that code needed at the particular point on the parse tree.

4.2 The TFI

The TFI uses semantic routines to control the walk of the parse tree. In particular, the semantic routines call the TFI recursively through the use of a routine called PERFORM. PERFORM returns true or false depending on whether the expression which was generated and evaluated by the lower nodes is true or false. This is used by the conditional statements to guide the interpreter down the proper branches. The next section discusses the use of PERFORM in control statements and expressions.

4.3 The TFPC

4.4 Introduction

This section describes two methods for constructing a TFPC. One of these was implemented. The first method described involves the interpretation of parse trees by walking the tree in preorder. The second method discussed involves marking a path from changed nodes to the root node of the parse tree.

4.5 Interpretation by tree walking

The system can walk through the tree in two ways:

1. Trace through the tree in preorder, checking each node to see if the source code has changed and recompiling that code.

2. When the editor changes a line, establish a path to the root of the parse tree by "beating a path" through the parent links from the left-most node which the changed line references. Then, when the TFI runs it is not necessary to check all the paths to find which nodes need to be translated.

There are advantages and disadvantages to these two schemes. Each of these schemes is examined and the modifications to the data structures that are necessary are described.

1. Tracing through the tree involves checking all the nodes to tell whether the text which generated the parse tree was changed. For a very large program (and a very large parse tree!) this might be intolerable. However, having a module symbol table structure restricts the size of parse trees and therefore the range of the search. The following recursive procedure implements the top down breadth-first search:

```
let TraceTree(root) be
[
// Return if a terminal node
IF root eq 0 THEN RETURN;
// Trace through, left to right
FOR branchNumber = 1 TO root >> NODE.size DO
// test to see if it's a modified terminal node
TEST root >> NODE.modified &
     root >> NODE.branch ^ branchNumber eq NIL
// If so, found a modified node
IFSO FoundRange(node)
// If not, continue tracing
IFNOT TraceTree(root >> NODE.branch ^ branchNumber);
]
```

2. Whenever a line is changed by the editor, the editor must follow the parent links of the left most node generated by that line (which is kept in the line head). Then, by

following the parent links in each node, the TFC can work its way backward through the tree, marking each node as being modified. In this way, there is a readable path starting at the node. The following procedure is used to trace such a tree:

```
let TraceBackwards(node) be
[
let currentNode = node;
until currentNode eq NIL do
    [
    // return if already marked
    if currentNode >> NODE.modified then
     RETURN;
    // mark it
    currentNode >> NODE.modified = true;
    // And continue back up
    currentNode = currentNode >> NODE.parent;
    ] // end until
]
```

4.6 Detecting changed program lines

In either the TFC or TFI, it is necessary to detect the fact that a piece of program has changed. Given the fact that the parse tree nodes, not text lines, contain information about where code is, this is a problem. This can be solved in the following manner:

1. Each parse tree node now requires a line range. To discover whether code is still valid in a node, it is necessary to check the range of lines contained in the node.

2. In order to implement the "beating a path" method, it is necessary to include pointers from the text lines to the parse tree. Each line head also has a field which points to the left-most node generated by the code in the line.

3. Each line head needs a bit which states that parsing could begin at this line. This is easily done by placing a semantic routine at the beginning of the statement parser that states `IF lineBeginning THEN SetParsable(linePointer)`

In fact, the `StmntExec` routine in the system does that. This little trick allows the system to do what few interactive systems have done: statements need not be reparsable

at the beginning of a line. Clearly, this is so because there must be at least one line which starts a statement. In order to find that line, the system traces backward through the line heads until the specific line is found. If the line can not be found, then the modified line is the top line. In this case, the solution is to reparse the text from the beginning.

When the type declaration of a symbol changes, either through dynamic typing or changing the text line which declares a symbol, the code associated with that symbol may have to be reinterpreted or recompiled. The dependency chain kept in the symbol table bucket is a ring of lines which contain that symbol. When the type of that symbol changes, all the lines on the dependency chain are marked as changed. This forces the system to re-examine every use of that symbol.

4.7 The implementation

The implementation can either perform as a TFPC or a TFI. By ignoring the calls to PERFORM, the TFI can become a TFPC with the addition of the tree walk.

The tree walking method was chosen over the "beating a path" method because its problems were well known. However sufficient information was maintained to implement the other method if it became necessary.

The hardest problem is the detection of changed lines covered by a node. When a BASE command from the parser is executed, the current line pointer is pushed on the parser control stack. When a ROOT command is executed (to create an n-ary node), that line pointer is popped off as the line beginning the code which generates the n-ary node. In the case of a NODE command both the beginning and ending line pointers are set to the current line.

4.7.1 How the TFI works

As the tree walk proceeds, each node will generate code. The system can store this and save the pointer to the code area (called a "code buffer" by Mitchell) in the node. This

code is then executed immediately before walking the rest of the tree. In the interpreter, the code is not compiled if control fails to take that branch. When all branches of a node have been interpreted, then the code can be "copied" from all the branches into the node. If all the code was really copied, most of memory would be filled with duplicate code from lower branches. To solve this, pointers to code are passed. So, when the tree walk generates code (by calling semantic routines) for a node, the tree walker will save the address of the pointer to the code buffer and the size of the code buffer as well.

4.7.2 How the TFPC works

The TFI is part compiler. As explained in the previous section, the distinction between compiling and interpreting is that the interpreter will execute the code immediately whereas the compiler will insure that code is generated for all possible branches of control. To guarantee that a program is complete and compiled, the system walks the tree and skips nodes that already have code attached to them. Of course the code that is already attached to these nodes is gathered along with the code generated on the "untouched" nodes. Execution is trivial. Since all the code is valid and complete, the code pointers at the root node should also be valid. Therefore, to execute the program, the system executes starting at the code address held in the root node.

It should also be noted that at this point the parse tree is no longer needed for interpretation. The system could choose to release the tree to some lower level of storage such as disk. It should also be noted that the compiler can perform code optimization on the code generated by the interpreter since many parts of the program would be considered static.

4.8 An example of the operation of the TFI/TFPC

To illustrate the operation of an IPS using the TFI/C algorithm, a factorial program with two bugs introduced is used as an example programshown in figure 4.1 Text lines are denoted by [n], where n is the line number. Note that line 8 has been split across two lines.

```
[1]  procedure Factorial
[2] (n: integer);
[3] begin
[4]       var result,i: integer;
[5]       while i$>$ 0 do
[6]       begin
[7]             result := result*i;
[8]             x := x
[9]             + 1;
[10]       end;
[11]       Return(result);
[12] end;
```

Figure 4.1. The sample factorial program

The parse tree constructed from this program is shown in figure 4.8. The pointers to the lines are shown as well.

Since x will increase, clearly the program as constructed will never terminate. To correct this, the user will change statement [9] from + 1 to - 1. When the editor modifies line [9], the line is marked invalid. The parse tree if left in its present state until an attempt is made to run the program. Under partial compilation, the parse tree is walked in end-order, checking the range of lines associated with each statement node. If a line has been modified the algorithm checks whether the modified line can be used to restart the parse. Line [9] in the example program cannot start a statement. The previous line is then checked. In this case, the TFI need not look any further; the parse can be started from line [8]. A new tree is constructed and is grafted onto the existing tree.

However, another bug still lurks. The variable i is never initialized to n. The user would insert a line (Call it [4.5]) that sets i to be n. This new tree is also grafted in and then the tree would be as shown in figure 4.3 (the new branches are shown in heavy lines and the new symbols in italics). The path of the code generator is shown in figure 4.4.

The code can now be executed and the walk of the parse tree continues.

In the TFI system, execution would proceed differently. Since interpretation follows the path of execution, interpretation would first follow the flow shown in figure 4.5 below.

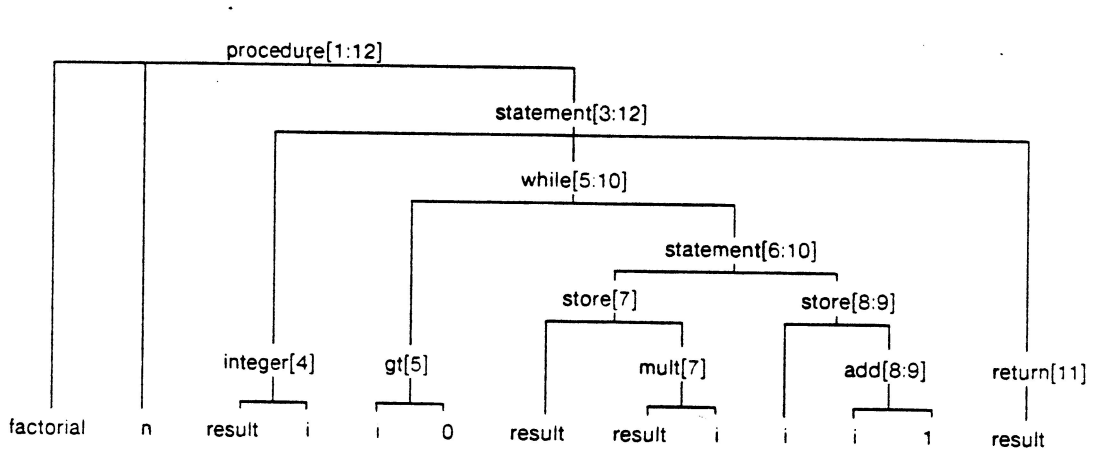

Figure 4.2. Parse tree of the sample program

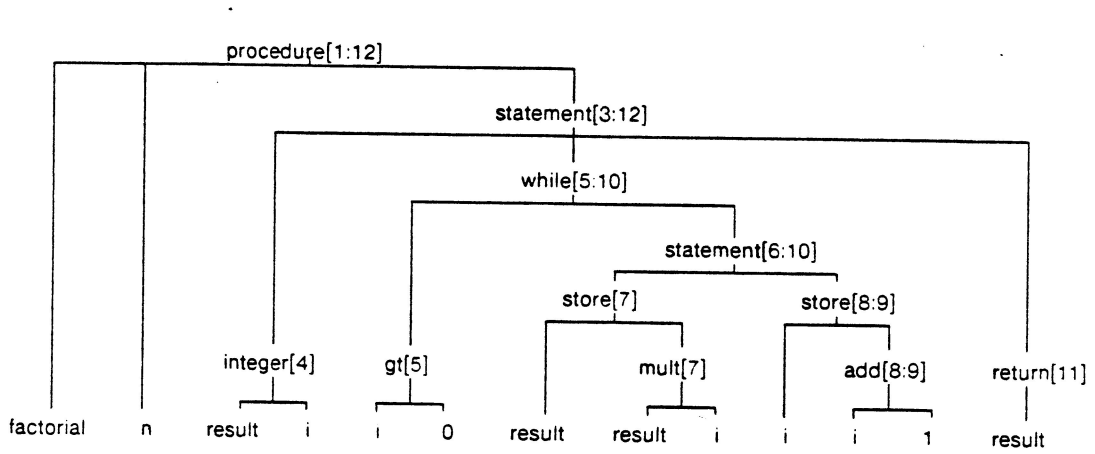

Figure 4.3. Changed parse tree of the sample program

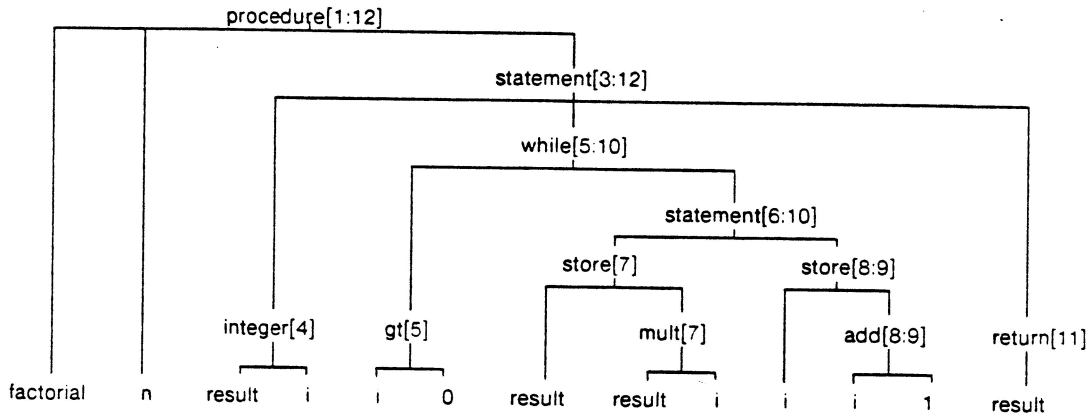

Figure 4.4. Code generator path for a TFPC

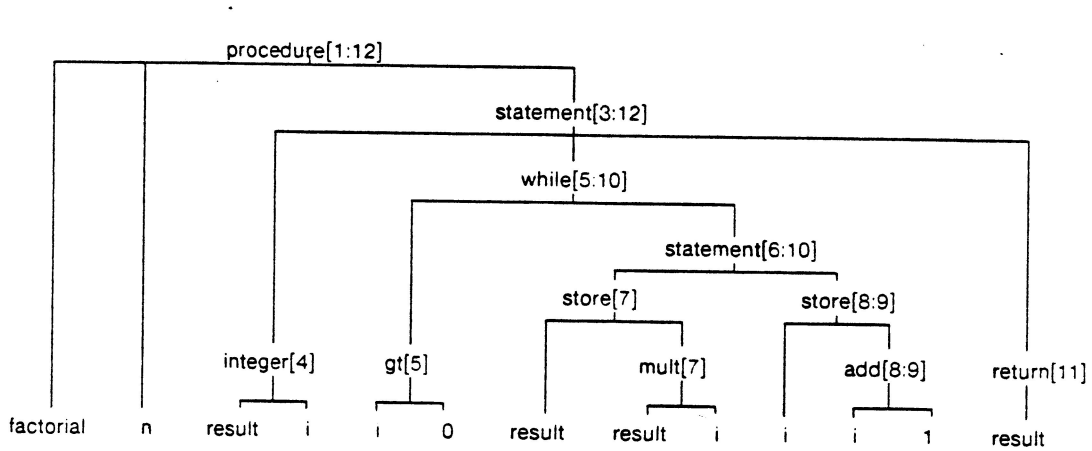

Figure 4.5. Interpretation under a TFI (one bug)

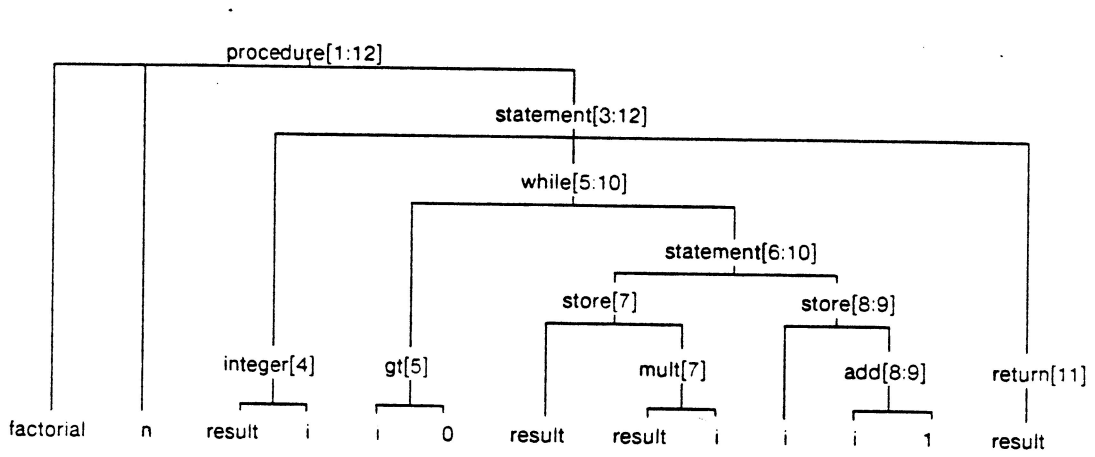

Figure 4.6. Interpretation under a TFI (no bugs)

Notice how most of the tree remains uninterpreted. After both bugs are corrected, the resulting interpretation path would be as shown in figure reffig:nobugs.

4.9 From compilation to interpretation

Now, the question is how does the system make the transition back from compilation to interpretation. Note that while the code generated for a node is valid and complete, there is no need to generate new code. If the dependencies associated with a variable change, then the code which uses that variable must change as well. Also, if a line of a program is modified, then the code which was generated for that line must be generated again. The TFC and the TFI have two different strategies for handling this event.

The TFC must ensure that all the code is correct before executing the code pointed to by the code pointer in the root node. This is done by walking the tree in end-order checking each node for bad code. If bad code is found (or perhaps a new text line), then code is generated for that node. This continues until the entire tree again has valid code.

The TFI, however, can execute code even when parts of the parse tree point to invalid

code. Since the TFI only executes code that is on the path of execution, invalid code may be by-passed during execution. However, in a TFC, it is possible that invalid code may be executed unless the changed nodes have their code re-compiled.

Mitchell makes the distinction between what he calls the "interpretation point" (I-point) and the "execution point" (X-point). The interpretation point is the point at which the interpreter would be if the parse tree was being interpreted (instead of the compiled code being executed). The execution point is the point where the program counter for the program is located. Note that Mitchell only allows the program counter to stop at three places: the end of a statement, the end of a loop, and before a function (procedure) call. Depending on whether the TFI is at a terminal or non-terminal node in the interpretation, the TFI decides whether or not to execute the semantic routines based on the following table:

Valid Code?	Terminal?	I-Point in range	Action
False	False	False	1
False	False	True	1
False	True	False	1
False	True	True	2
True	False	False	3
True	False	True	4
True	True	False	3
True	True	True	4

Chapter 5

Semantic analysis

5.1 Introduction

In this section, the writing of semantic routines are discussed. How to write semantic routines is discussed first. Next, the paper design for a language known as ISLE (for Interpretive Semantics LanguagE) for writing semantic routines is described. The language is used for the description of semantic routines.

5.2 Writing Semantic routines

There are special considerations that must be made when writing semantic routines for a TFI/C. If the system is a partial compiler, then it is possible that the compiler may skip over some nodes on its way down the parse tree to the node that has the changed lines. For this reason, the semantic routines must not depend on "inherited" attributes [Knuth, 1969] or must save the results so that the compiler can pick them up on its way down the tree. On the other hand, if the system is a TFI, then saving semantics is not necessary because the interpreter will have full state at any given node. Of course, "synthesized" attributes are acceptable to either the TFI or TFC.

After calling PERFORM (which is really a recursive call to the interpreter) for the

lower branches, a TFI semantic routine must then coerce the stack data to the proper type. For example, a production which creates a multiply will need to know the types of its two operands before generating the code for the multiply. After coercing data types, the next step is to generate the code. Lastly, the code is attached to the parse tree node that generated it. This is done by storing pointers to the starting and ending addresses of the block of code in the parse tree node.

FSL was designed to run in an environment with a syntax and semantics stack. Each element of the semantics stack would hold the semantics associated with its corresponding symbol on the syntax stack. Recall that the IPS operates on trees, not on semantics associated with symbols at parse time. Note that a crucial distinction has been made here. The distinction is being made between semantic routines that can be called at compile time and routines that are called at runtime. As mentioned earlier, Mitchell distinquishes between them and calls them S-actions and X-actions. In this case of the IPS, the actions operate on different data structures. Actions performed at parse time will only be performed once; actions at run time may be performed any number of times.

This problem is solved in a straight forward way. One can think of the symbols on the syntax stack as being represented by the nodes in a parse tree. Each node has a preset arrangement of branches; for example, an IF statement would have a boolean expression, a statement for the THEN clause, and possibly a statement for the ELSE clause. Since the parsing algorithm is scans tokens from left to right, (counting from 0) the boolean expression would be branch 0 of the THEN statement branch 1 and so forth. This assumes that the translator writer knows where various branches are attached. In fact, the translator writer must know because interpretation depends on the ordering of the branches of the tree!

5.3 ISLE

5.3.1 Introduction to ISLE

Languages for writing semantics routines can save the compiler writer considerable time and effort. ISLE is an extension of FSL [Feldman, 1964] which includes modular decomposition [Parnas, 1972] and abstract data types modeled after the Alphard language [Wulf, et al., 1977] [Shaw, et al., 1977].

Earlier in this thesis, the distinction was made between semantic routines (S-actions) and code generation routines (X-actions). As Mitchell points out, these are distinquished in FSL by use of the so called "code brackets". The clear distinction between S-actions and X-actions is one of the benefits of using a language with code brackets. Therefore, this feature was kept in the design of ISLE.

5.3.2 Type definition facilities

FSL limited the translator implementer by restricting the realm of types the user was able to use. In the case of linked lists, Feldman agrees:

> "The other difficulty in (the implementation of) LISP and COMIT was the absence of fixed FSL primitives to act on linked lists. These were not put into the system because, as yet, there is no universally accepted way of representing and operating upon such lists." (pp. 99)

In ISLE, this problem is solved by creating a new form for linked lists.

On the other hand, the opposite opinion is to keep the primitives simple and abstract. One response to this is that it is both possible and, in part, necessary to express the semantics of the translator in the type structure of the language.

ISLE uses the "division of labor" mechanisms of Alphard to draw the line between the implementation and the abstract representation of a type. For an example, the definition of a queue type in ISLE would be declared as:

```
form Queue(item: form);
```

```
 functions
  Enqueue(q: Queue; x: item);
  Dequeue(q: Queue);
  Length(q: Queue) returns Integer;
 end functions;
 implementation
  let Enqueue(x) be ...
 end implementation;
end form;
```

5.3.3 Organization of ISLE programs

The overall organization of an ISLE program is similar to an ALGOL program: first, the definitions of new types, then the ISLE statements themselves. Several types are already declared within ISLE. They are integers, stacks, booleans, words and the FSL Table. In FSL, Feldman often had types which were found in runtime as well as compile time. For example, a stack may be used to hold jump labels during compile time and values for the interpreter at runtime. This notion is now extended to cover all types. The ability to declare a runtime type is done by preceding the declaration of a type instance by the reserved word RUNTIME. For example, the user could declare an instance of a runtime queue by the declaration:

```
polishQueue: RUNTIME Queue;
```

The system calls functions associated with a given type by specifying the function and then the type. Again, for queues:

```
Enqueue.polishQueue[x];
```

ISLE has the following predefined types and functions:

```
Stack:
 Push; Pop; Loc
Boolean:
 Test; Set; Loc
Table:
 Enter; Loc
Cell:
 Loc
```

5.4 Optimization of ISLE generated code

It is true that at some level some knowledge of the source machine is necessary. For example, the compilation of a code bracket requires knowledge of how to generate machine opcodes as well as optimization specific details. However, for optimization such as peephole optimization [McKeeman, 1965], it would not be necessary to have the optimizer as part of the translator.

5.4.1 Writing semantic routines in ISLE

In ISLE, the task of writing the semantics is divided into three parts: first, the user defines the new forms; second, the user declares all the variables needed using either the predefined classes listed above or the new forms declared in step one; last, write the semantic routine in the following form:

```
routineName: statement; .. ; statement END
```

where routineName is either the name of the node (an X-action) or the name of a semantic routine called by an EXEC action (an S-action). A statement in ISLE is very much like a statement in FSL, except that the semantic routine writer uses the declared forms to call the operations on data types. The symbols used by ISLE are slightly different from those in FSL. For example, instead of LEFT1, ..., LEFT5 and RIGHT1, ..., RIGHT5 as in FSL, ILSE calls them Left[0],...Left[7] and Right[0],... Right[7]. The semantics associated with children nodes are referenced as BRANCH[i].

In the first section, the problem of coercing the result of operations was discussed. In ISLE this mechanism can easily be made available to the user by placing the type specific coercer in the definition of the type. For example, coercion with a real and an integer will generally float the integer into a real. The code for such a coercer would be:

```
RealCoercer(arg1,arg2) =
IF arg1.type = Integer and arg2.type = Real THEN
 Float(arg1)
ELSE
```

```
IF arg1.type = Real and arg2.type = Integer THEN
 Float(arg2)
End;
```

After that code is generated through the use of "code brackets" and finally, the completed code is passed to the parent node. Before being passed up, it is executed.

An example of an ISLE routine is shown below for the end of an IF statement:

```
markIf:
 Push.FloatingAddress[FA,0]; (* Store a dummy 0 *)
 Code(IF Branch[0] THEN Jump[FA[0]]); (* Code bracket *)
END markIf;
```

The interpretation for this piece of code is as follows: When an IF statement is encountered, a 0 is pushed on the variable FA (of type FloatingAddress stack). If BRANCH[0] is false, then a jump is generated to the label on the top of the stack (i.e., offset 0). The purpose for this is to generate the jump over the succeeding statements should the boolean expression prove to be false. The FA (called a FLAD by Feldman) is fixed up by the ASSIGN operator. The FA is a stack in case the IFs are nested.

It is possible for a ISLE routine to access the symbol table through the normal ISLE type mechanism. For example, to enter a new symbol with the type `Real`, the routine `SymbolTable.Enter[LEFT[2], Real]` would be called. Note that `LEFT[2]` would hold the string name of the symbol in this case.

5.5 Conclusions

This thesis has described the design and the implementation of a programming system which incrementally compiles statements for an ALGOL like language. This section considers some of the doubts that were present while implementing the system as well as further directions for this study.

5.6 Parallel Processing

In the IPS just described, the editor and the TFI were implemented as separate cooperating processes. Swinehart makes a big case for non-preemption throughout his thesis. He states that:

> A request for one's services is not always granted instantly. In fact, it is sometimes not granted at all. At any rate, having noticed such a request, one may respond to it immediately, queue it temporarily until some other task is complete, or ignore it entirely. He [the user] is not automatically preempted by a "service request"; he can continue what he is doing, or go on to something else entirely; nor must he take care of things in a fixed order. (pp. 9)

While it was useful at times, the overall usefulness of separating the editor from the TFI is not clear. If the TFI and the editor are separated, it allows the user to make changes to the program while the program is running. The question remains: how many times will the user be performing a task in the TFI that would hang up long enough for the user to get restless enough to want to do editing. Note that LISP systems have editors which can be called at a program break. After modifying a function or variable, the user can then proceed.

5.7 Efficiency issues

The most substantial cost of the IPS is space. In the implemented system, each parse tree node occupies 5 words including the code pointers. Each text line head occupies 7 words. And, of course, each text line must be stored in memory as well. Each link in the dependency chain takes up 3 words. For a program of any size, all of these records add up to take substantial portions of memory.

The compile-time overhead for the TFPC/TFI is less than the TFTC. The TFPC and TFI both compile code faster because they are compiling smaller pieces of code than the TFTC which must compile the entire program whenever a change is made. The question is: What is the runtime efficiency of these systems? Although intuition would say that the TFI must be the slower than a TFC, detailed measurements should be performed to determine

exactly how slow it runs compared to a TFC. It would also be interesting to compare the performance of a TFI with a parse tree driven interpreter (i.e., a "TI").

5.8 Helping the debugger

If a debugger for a higher level language is to be informative and useful to the user, it must offer the user something more than machine code output [Satterthwaite, 1974]. How can the TFI or the editor help in the debugging of programs? As mentioned before, the editor-parser interaction enabled the user to know which line caused the error. The TFI can offer a similar mechanism at runtime. When an error occurs, the parse tree can be walked to find the range of code (as stored in the parse nodes) that the error occurred in. Note that under full interpretation, the tree walk is not necessary. However, when most of the code is compiled, it is necessary to find what statement created that code. Unfortunately, it is somewhat time consuming to search the parse tree. There is a possible kludge that might prove useful: At the beginning of all statements, insert an instruction sequence which stores the current line head pointer in a location in memory. Then, at the runtime error, this location will contain the pointer to the line that contains the error.

In Satterthwaite's system two interpreters were used to control tracing and maintain frequency counts. An additional interpreter was used by the editor. The X interpreter (not to be confused with X actions) executed machine code, the R interpreter simulated machine code execution and the E interpreter displayed and edited source text.

Satterthwaite uses two markers called "alpha" and "beta" to synchronize the R and X interpreters. When a "alpha" is encountered, control switches from the X interpreter to the R interpreter. When a "beta" is encountered, control switches from the R interpreter to the X interpreter. These markers are explicitly introduced for conditional statements. In a TFI, the X interpreter and R interpreter may still be desired for the tracing and execution of programs. The "alpha" and "beta" markers could be introduced by semantic routines and the switching between interpreters would just involve setting a switch which chose one or the other inside the TFI.

The profiling capability could also be implemented by inserting another instruction sequence at the beginning of all statements which increments a counter field in the line heads.

There is one major problem with the two mechanisms mentioned above. They violate the separation between IPS address space and the object program address space. Perhaps this could be overcome through the use of tables or shared memory.

5.9 A transaction-based parallel process

The system which has been discussed in this thesis has been serial in nature. Each phase of the system must wait for the some other phase to complete before starting up. If the editor, parser and the TFI/C run concurrently, then the organization of the system must be changed to run on transactions.

Although transactions have been used for a long time in data base systems, they were used in a parallel garbage collector [Deutsch and Bobrow, 1976] which inspired the following proposal.

The editor would create transactions and pass them on the a parser which runs in parallel. The parser would, in turn, pass them to the TFI. The following sorts of transactions would be created by the editor.

1. Create a new line before the line pointer p
2. Delete the line pointed to by p

The transactions created by the parser would be:

1. Attach new tree to node n
2. Destroy the tree at node n

The system would create transactions for type changes, i.e., Change the type of variable v to type t

The TFC would use these transactions to create new code. It is not clear when the various processes would run. If resources are limited, the system may choose to have another process called the referee which decides which process may run on the basis of available resources. For example, on a system with limited virtual memory, the TFC may want to run as often as possible to prevent large numbers of transactions from building up. On the other hand, if processing time is limited, then the TFC may want to process the transactions immediately. The changes to the line structure for the incremental method are minimal. As mentioned before, the principal problem is when to decide to parse. It is quite clear that parsing "before all the lines are in" would be disastrous. Often the programmer may decide that the statement is in error and rewrite the statement before calling the TFC. If the system had already parsed the line, then the work done by the system would have been wasted.

5.10 Bootstrapping based on microprogrammed implementation

In his thesis, Mitchell discusses the bootstrapping of a TFI system. If the TFI were microprogrammed, the bootstrapping could be done faster.

The implementation of an IPS could proceed along the following steps:

1. Implement a translator in BCPL (or any other systems programming language) that translates the language L into a machine code which is executed by a microcoded IPS.

2. Use this translator to implement another translator in L which translates L into the machine code for the IPS. This would give the implementer a chance to recover from the mistakes in the BCPL version.

One question is whether the implemented system could use instructions that dealt specifically with the interpretation of a parse tree.

Chapter 6

Future directions for research

This thesis has touched on several possible areas for further research. They are:

1. The use of a abstract types and type definitions in the implementation of semantic routines;
2. The implementation of transaction based processes. Further research is needed into incremental flow analysis as well as incremental parsing;
3. The use of semantic routines in helping the system know about language peculiarities.;
4. The use of abstract data types in a formal semantics language could be useful for the verification of compilers and interpreters;
5. The microcoded implementation of a TFI/TFC.
6. Detailed measurement on the runtime efficiency of the TFI, TFPC and TFTC.

6.1 Conclusion

Today, with increasing emphasis on interaction and personal computing, most compilers are "batch" compilers; they accept input at one time and emit code one time. The TFI

represents a reasonable alternative to the inefficiency of interpreters and the lack of flexibility of compilers. Although many unanswered questions remain in the area of interactive computing, this implementation of a TFI may convince system designers that it is possible to design systems which can run fast and interact pleasantly with the user.

Appendix A

Editor algorithms

Display updating algorithm: [1]

```
lineNumber = 0;
pointer = lineCache ! lineNumber;
IF pointer eq NIL THEN RETURN;
WHILE lineNumber ls displayLines DO
// displayLines is 12 for the IPS
[
    // Find next marked line if onlyMarkedLines
    IF onlyMarkedLines THEN
        WHILE not pointer >> LINE.marked &
            pointer ne NIL DO
            pointer = pointer >> LINE.next;
    IF pointer eq NIL THEN RETURN;
    // See whether to show a deleted line
    TEST visibleLines & pointer >> LINE.deleted
    IFSO lineNumber = lineNumber +
                      DisplayOldLines(pointer,lineNumber);
    IFNOT WriteString(pointer >> LINE.body);
    // Invert the background if marked
    IF pointer >> LINE.marked THEN
        SetBlackBackground(lineNumber);
    lineCache ! lineNumber = pointer;
    // return if the end of text is reached
    IF pointer >> LINE.next eq NIL THEN RETURN;
    lineNumber = lineNumber+1;
]
```

[1] Note that all algorithms are given in BCPL [Richards, 1969]

```
Mouse actions:
// First calculate the position from the window top
lineNumber = (xmouse-xtop)/fontHeight;
SWITCHON MouseButtons() INTO
[
CASE up:
    lineCache ! 0 = lineCache ! lineNumber;
    ENDCASE // up;
CASE down:
    FOR backPointer = 1 TO lineNumber DO
        [
        // If the pointer is not nil, then follow
        // the chain of backward pointers
        IF linePointer.previous eq NIL THEN BREAK;
        linePointer = linePointer >> previous;
        ] // end FOR
    lineCache ! 0 = linePointer;
    ENDCASE // down;
CASE middle:        // Jump mode; depends on next button push
    SWITCHON MouseButtons() INTO
    [
    CASE up:
        lineCache ! 0 = Pop(linePointerStack);
    ENDCASE //  up;
    CASE down:
        Push(lineCache[0],linePointerStack);
    ENDCASE //  down;
    CASE middle:
        JumpTo((xtop-xmouse)*lineCount/(12*fontHeight));
    ENDCASE // middle;
] // of the case statement

Undo Algorithm:

topPointer = pointer;
lastPointer = pointer;
pointer = pointer >> LINE. old;
WHILE pointer NEQ 0 DO
[
    // If the line pointed at is marked then
    // insert before the line
    if pointer >> LINE. marked THEN
    [
        InsertLine(pointer,topPointer);
        // Patch the line out
        lastPointer >> LINE.old =
            pointer >> LINE.old;
    ]
```

```
    lastPointer = pointer;
    pointer = pointer >> LINE. old;
]
```

Bibliography

Aho, A., and Ullman, J. 1972. Theory of Parsing, Translation and Compiling, vol. 1. Prentice Hall.

Aho, A., and Ullman, J. 1977. Principles of Compiler Design. Addison Wesley.

Baker, C.L. 1966. JOSS: Introduction to a Helpful assistant. Rand memorandum RM-5058-PR. Rand Corporation.

Beals, A. 1969. The generation of a deterministic parsing algorithm. Report no. 69-304. University of Illinois at Champaign-Urbana.

Braden, B., and Wulf, W. October, 1968. The implementation of BASIC in a multiprogramming environment. Communications of the ACM, vol. 11 no. 10, pp. 688-692.

Deutsch, L. P., and Lampson, B.W. December 1967. An online editor. Communications of the ACM, vol. 10 no. 12, pp. 793-799.

Deutsch, L. P., and Bobrow, D. September 1976. An efficient incremental garbage collector, Communications of the ACM, vol. 19 no. 9, pp. 522-525.

Earley, J., and Caizergues, P. December 1972. A Method for Incrementally Compiling Languages with Nested Statement Structure. Communications of the ACM, vol. 15 no. 12, pp. 1040-1044.

Englebart, D., and English, W.K. 1968. A research center for augmentation of human intellect. In AFIPS 1968 Fall Joint Computer Conference. pp. 395-400.

English, W.K., Englebart, D., and Berman, M.L. 1967. Display-selection techniques for text manipulation. IEEE Transactions on Human Futures in Electronics. vol. HFE-8 no. 1. pp. 5-15.

Feldman, J.A. May 1964. A Formal Semantics for computer oriented languages. Ph.D. dissertation. Carnegie Institute of Technology.

Floyd, R.W. October 1961. A descriptive language for symbol manipulation, Journal of the ACM, vol. 8 no. 4, pp. 579-584.

Frost, M. 1978. E. Online documentation at Stanford Artificial Intelligence Laboratory: E.ALS[UP,DOC].

Goldberg, A., and Kay, A. 1976. Smalltalk-72 instruction manual. Xerox Palo Alto Research Center Technical Report, SSL 76-6.

Gries, D. 1971. Compiler construction for digital computers. Wiley. New York.

Guibas, L., and Wyatt, D. January 1978. Compilation and Delayed Evaluation in APL. In Fifth Principals of Programming Languages conference, pp. 1-8. Tucson, Arizona.

Hay, R.E., and Rulifson, J.F. March 1968. MOL940, preliminary specifications for an ALGOL-like Machine Oriented Language for the SDS-940. Internal Technical Report NAS-1-5940, SRI Project 5890.

Ingalls, D.H. The Smalltalk-76 Programming system: Design and Implementation, In Fifth Principals of Programming Languages conference, pp. 9-15. Tucson, Arizona.

Iverson, K.E. 1962. A Programming Language. Wiley.

Iturriaga, R., Standish, T., Krutar, R., and Earley, J. October 1966. The implementation of Formula ALGOL in FSL, Internal Report. Carnegie Institute of Technology.

Johnson, W.L., Porter, J.H., Ackley, S.I., Ross, D.T. 1968. Automatic generation of efficient lexical processors using finite state techniques. Communication of the ACM, vol. 11 no. 12, pp. 805-813

Katzan, H. Jr. 1969. Batch, conversational and incremental compilers. In AFIPS Spring Joint Computer Conference, pp. 47-56.

Kay, A. 1969. The reactive engine, Ph.D. dissertation, University of Utah

Knuth, D.E. 1967. Semantics of Context Free Languages. Mathematical System Theory, vol. 2 no. 2, pp. 127-145.

Lampson, B.W., and Sproull, R.F. December 1979. An Open Operating System for a Single-User Machine In Seventh Symposium on Operating Systems Principles, pp. 98-105.

Lampson, B.W., Horning, J.J., London, R.L., Mitchell, J.G., and Popek, G.J. February 1977. Report on the programming language Euclid. SIGPLAN notices, vol. 12 no. 2.

Lindstrom, G. 1970. The design of parsers for incremental language processors. In Second SIGACT Theory of Computation Conference, pp. 81-91.

Liskov, B., Atkinson, R., Bloom, T., Moss, E., Schaffert, C., Scheifler, B., and Snyder, A. 1979. CLU Reference Manual. MIT LCS TR-225.

Liskov, B., Snyder, A., Atkinson, R., and Schaffert, C. 1977. Abstraction Mechanisms in CLU. Communications of the ACM, vol. 19 no. 9, pp. 564-576.

Lock, K. 1968. An object code for interactive applied-mathematics programming. IBM San Jose technical report TR 02.425.

Lock, K. 1965. Structuring programs for multiprogram time-sharing on-line applications. In AFIPS Fall Joint Computer Conference, pp. 457-472.

McCarthy, J., Abrahams, P.W., Edwards, D.J., Hart, T.P., Levin, M.I. 1963. LISP 1.5 Reference Manual. MIT Press. Cambridge, Massachusetts.

McKeeman, W. July 1965. Peephole optimization, Communications of the ACM, vol. 8 no. 7, pp. 443-444

Mitchell, J.G., Perlis, A.J. and Van Zoeren, H.V. 1969. LC* - A language for interactive computing, In Interactive Systems for Experimental Applied Mathematics, ed. Klever, M. and Reinfelds, J. Academic Press. New York.

Mitchell, J.G. 1970. The design and construction of flexible and efficient interactive programming system. Ph.D. dissertation. Carnegie Mellon University.

Parnas, D. December 1972. On the criteria to be used in decomposing systems into modules. Communications of the ACM, vol. 15, no. 12.

Peccoud, M., Griffiths, M., and Peltier, M. 1968. Incremental Interactive Compilation. In IFIP 1968 Proceedings, pp. B33-B37

Perlis, A. March 1975. Steps toward an APL compiler - Updated, Yale Department of Computer Science Research Report no. 24.

Reiser, J. August 1976. SAIL Reference Manual. Stanford Artificial Intelligence Laboratory Memorandum 289.

Richards, M. 1969. BCPL - A language for systems programming, In AFIPS Spring Joint Computer Conference, pp. 557-566.

Ryan, J.L., Crandall, R.L., and Medwedeff, M.C. 1966. A conversational system for incremental compilation and execution in a time-sharing environment. In AFIPS Fall Joint Computer Conference, pp. 1-21.

Satterthwaite, E.H. Jr. 1974. Source language debugging tools, Ph.D. dissertation. Stanford University.

Swinehart, D.C. 1974. Copilot: A multiple-process approach to Interactive Programming Systems, Ph.D. dissertation. Stanford University.

Schwartz, J. 1976. A view of program genesis and its implication for future programming languages. In New Directions in Programming Languages, ed. S.A. Schumann, Institut de Recherche d'Informatique et d'Automatique.

Shaw, M., Wulf, W.A., and London, R.L. 1977. Abstraction and Verification in Alphard: Defining and Specifying Iterators and Generators Communications of the ACM, vol. 20 no. 8, pp. 553-564.

Stoy, J.E., and Strachey, C. 1969. OS 6 - An experimental operating system for a small computer, The Computer Journal, vol. 15 no. 2, pp. 117-124 and vol. 15 no. 3, pp. 195-203.

Thacker, C.P., McCreight, E.M., Lampson, B.W., Sproull, R.F., and Boggs, D.R. 1980. Alto: A personal computer. to appear in Computer Structures: Readings and Examples, ed. Siewiorek, Bell and Newell. McGraw-Hill. New York.

Teitelman, W. 1978. InterLisp reference manual. Xerox Palo Alto Research Center report.

Wirth, N. 1977. Modula: a language for modular multiprogramming. Software Practice and Experience, vol. 7 no. 1.

Wirth, N. 1977. The design and implementation of Modula. Software Practice and Experience, vol. 7 no. 1.

Wulf, W.A., editor February 1978. (Preliminary) An informal definition of Alphard. Carnegie Mellon University Computer Science Department report CS-78-105.