The University of Calgary

AN EDITOR FOR TREES

by

Radford M. Neal

A thesis submitted to the Faculty
of Graduate Studies in partial fulfillment
of the requirements for the degree of
Master of Science

Department of Computer Science
Calgary, Alberta
September, 1980

THE UNIVERSITY OF CALGARY

FACULTY OF GRADUATE STUDIES

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies for acceptance, a thesis entitled, "An Editor for Trees" submitted by Radford M. Neal in partial fulfillment of the requirements for the degree of Master of Science.

Supervisor
Prof. D. R. Hill
Department of Computer Science

Dr. S. P. Soule
Shell Canada Resources

Dr. G. M. Birtwistle
Department of Computer Science

Dr. R. E. Dewar
Department of Psychology

1980-10-01

ABSTRACT

An editor is described which allows the user to manipulate information represented in the form of trees rather than as text. Applied to the editing of programs, this approach aids the programmer in the creation or modification of his program by eliminating the possibility of syntax errors, providing automatic formatting of the program, and reducing the number of keystrokes required to enter a program. The significance of regarding programs as trees in the development of an advanced program development system is discussed, and a powerful macro language exploiting the tree representation is described.

ACKNOWLEDGEMENTS

I wish to thank Steve Soule for his supervision during most of this project, and my present supervisor, David Hill, for his greatly appreciated assistance during a time when he has had many other obligations. I also thank James Gosling and others with whom I have had helpful discussions.

Table of Contents

INTRODUCTION. 1

THE USERS'S VIEW OF THE TREE EDITOR 4
Forms of Trees 5
The Basics of Language Descriptions. 8
Basic Operation of the Editor. 13
Entering and Modifying Trees 15
Further Aspects of Language Descriptions 19
Automatic Actions of the Editor. 23
Window Manipulation. 25
Elision. 26
The Macro Language 28

THE PRESENT IMPLEMENTATION. 34
Representation of Trees. 34
Operations on Trees. 38
Producing the Display. 40
Cancellation of Commands 42

RATIONALE FOR DESIGN DECISIONS. 45
Requirements of the Design 46
Basic Choices in the Design of an Editor 48
The Language-description Language. 51
The Editing Commands 54
The Keyboard Layout. 58
The Display. 59
Implementation Choices 63

DISCUSSION OF THE TREE EDITING CONCEPT. 66
Alternatives to Tree Editing 67
Advantages and Disadvantages of Tree Editing 70
Applications of Tree Editing 72
Further Development of the Tree Editor 75
Handling Context-sensitive Syntax. 77
Possible Human-factors Improvements. 80
Implications for Language Design 82
Significance of the Macro Expansion Language 85

CONCLUSION. 89

REFERENCES. 90

APPENDIX A - DESCRIPTION OF THE LANGUAGE-DESCRIPTION LANGUAGE . . . 92
APPENDIX B - DESCRIPTION OF PASCAL. 94
APPENDIX C - DESCRIPTION OF THE DOCUMENT LANGUAGE 99
APPENDIX D - SUMMARY OF EDITING COMMANDS. 100

List of Figures

Figure		Page
1	An example of a tree.	6
2	A tree representing an if-statement	6
3	A tree representing a sentence.	6
4	A partial language description for documents.	11
5	A description of an example language.	11
6	A tree of the example language.	12
7	The display representation of the tree of figure 6.	12
8	Illustration of the cursor movement commands.	16
9	Illustration of the entry of a subtree.	18
10	Illustration of tree modification	20
11	An improved description of the example language	22
12	Illustration of the use of windows.	27
13	The effect of cursor position on elision.	29
14	An extension to the example language.	30
15	Implementation of the extension	30
16	A program using the extension	31
17	Expansion of the program of figure 16	31
18	Structure of blocks	37
19	Creation of a modified tree	39
20	Layout of commands on the keyboard.	60
21	Description of strings and string functions	86
22	Definition of string functions in the macro language. . . .	86

INTRODUCTION

Information may be represented in many ways. Selecting a proper method of representing information is a central problem of computing science. This project concerns the representation of information that is to be manipulated directly by people via an editor.

Traditionally, people have communicated information to computer systems as text - a string of characters, sometimes grouped into lines. This text is then parsed by the machine to reveal its underlying structure. In most cases this structure is that of a tree - a hierarchy in which each component is composed of separable sub-components whose relationships can be diagrammed in a way that resembles the branching of a tree. This hierarchy is illustrated by this document, which is composed of sections, which are in turn composed of sub-sections, which have a further structure in terms of paragraphs, sentences, words, and finally letters.

The tree editor described in this document allows its user to bypass the stage of representing information as text and instead enter and manipulate it in its underlying tree form. The user may then refer to the information in terms that reflect its true structure; he need not decide how to represent his information as text; and the possibility of a syntax error - an inability of the machine to parse a text - is no longer present.

The use of the tree editor to edit computer programs written in high-level languages is of particular interest. These programs typically exhibit a well-developed tree structure and their entry and modification occupy a significant fraction of a programmer's time. In addition to eliminating syntax errors, it is hoped that performing modifications with the tree editor will be faster and easier than with a text editor.

Furthermore, representing programs as trees rather than text should make it easier to create an environment in which the computer can intelligently aid in the development of a program. [Winograd, 1979], [Sandewall, 1978], and [Teitelman, 1979] discuss existing or proposed systems of this kind. A fundamental question with any such system is "How do the programmer and the machine communicate about the program?" I feel that trees are a more effective medium of communication than text for this purpose. For example, consider a program to compare two versions of a program and inform the user of the differences. If one difference is the replacement of the statement "S" by "if B then S fi", a textual comparison would reveal two apparently unrelated differences; a comparison of the tree structure would reveal that there is only one change and this difference could be communicated to the programmer in intelligible terms.

The succeeding sections of this document describe the appearance of the editor from the user's point of view, the implementation of the editor, the rationale for the design and implementation decisions

taken, and a discussion of the tree editing concept and its possible development.

THE USER'S VIEW OF THE TREE EDITOR

This section describes the tree editing system from the user's point of view, but it is not meant to be a complete user manual. Discussion of the implementation and of the justification for design and implementation decisions is postponed to later sections.

The tree editing system consists of a number of programs. First, there is the language-description compiler. The language for which the tree editor is being used determines the allowable forms of trees, the way in which these trees are displayed, and the way in which they are entered. The language-description compiler takes a specification of a language and produces an encoded version of this description which is used in producing a set of programs for editing trees of that language.

The most important of these programs is the editor itself. The editor for a language allows files containing trees of that language to be created and modified by the user.

Also important is the listing program. This program takes a file containing a tree of the language and produces a textual representation of the tree suitable for printing or for passing on to a compiler which cannot operate on the tree directly.

A macro expansion program allows the user to extend the language with new constructs and expand them into equivalent combinations of old

constructs when necessary.

Finally, there are slightly specialized versions of the editor and listing program for use in producing documents. These programs were used to produce this document itself.

Forms of Trees.

A tree may be recursively defined as consisting of a production and a series of zero or more trees. The trees which are immediate components of a tree are known as the children of the tree (by analogy with family trees). The parent of a tree is the tree which contains it as a child. Two trees which are children of the same parent are called siblings. The concept of a subtree may be defined by saying that a tree is a subtree of itself and the subtrees of a tree are also subtrees of the parent of the tree. A tree is a root if it has no parent and a leaf if it has no children.

This terminology is illustrated by the tree in figure 1. The productions of the trees are represented by letters; arrows point from a tree to its children. The tree with production A is the parent of B and also of C. Hence B and C are siblings and are children of A. Note that the trees with productions E, F, G, H, and I have no children. All the trees shown are subtrees of A; B, D, H, and I are subtrees of B. A is a root.

In general, it is possible for several trees to have the same

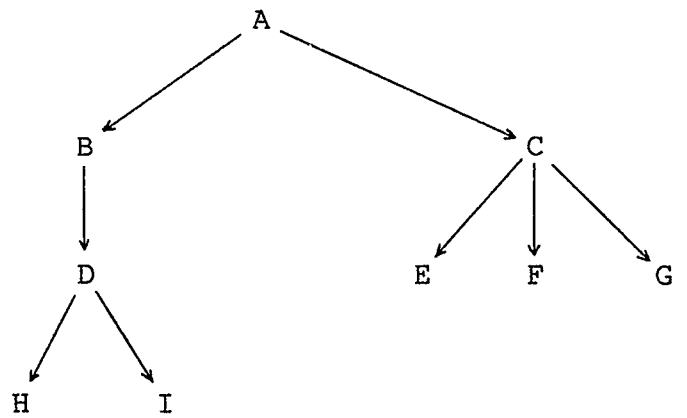


Figure 1. An example of a tree.

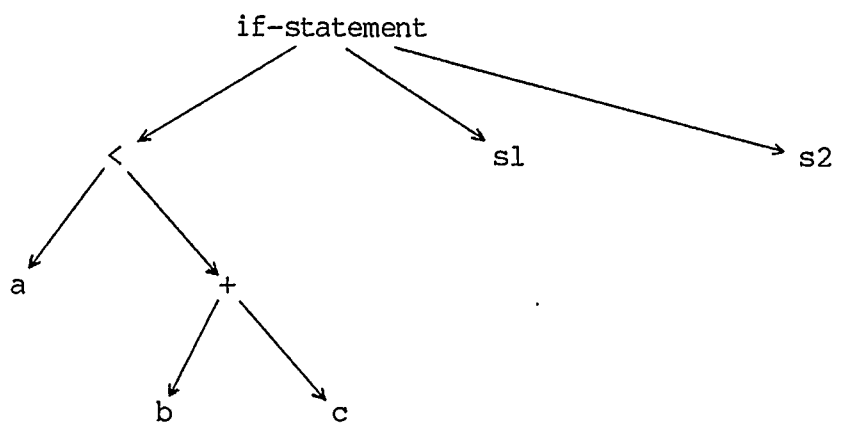


Figure 2. A tree representing an if-statement.

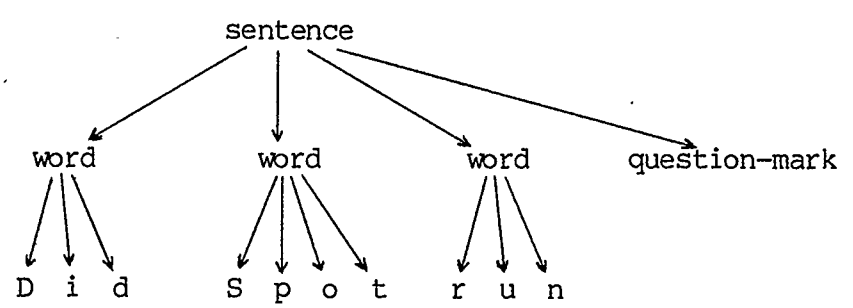


Figure 3. A tree representing a sentence.

production (unlike the example just given); the trees are distinguished by having different children (see figure 3 for an example). The production of a tree also determines the number of children the tree must have. For instance, any tree with production C might be required to have three children. These children have a definite ordering. More generally, the children of a tree belong to child groups, with the children within a group having a definite ordering and the groups themselves also being ordered. The production of a tree determines exactly how many groups of children the tree will possess and the kinds of groups these will be. Groups are of four kinds - solitary groups containing exactly one child, optional groups containing either zero or one child, series groups containing one or more children, and optional series groups containing zero or more children.

This structure may be used to represent many kinds of information. For example, the Pascal statement "if a<b+c then s1 else s2" may be represented by the tree in figure 2 (some detail at lower levels has been omitted). A tree with the if-statement production must have three child groups - a solitary group containing the condition of the if statement, another solitary group containing the statement to be performed if the condition is true, and an optional group containing the statement to be performed if the condition is false or containing nothing if that statement is absent. The < and + productions require two solitary child groups.

Another example is shown in figure 3. This tree represents the

sentence "Did Spot run?". The sentence production specifies that the root of this tree should have a single series child group. The children of this group are the words and punctuation of the sentence. The word production specifies the same structure, but the children in this case will be letters. The question-mark production requires that there be no child groups. There is a separate production for each letter, all of which have no child groups.

This completes the description of the trees which are manipulated by the tree editor. Note that although much of the information in the trees is represented by which productions are present, the tree structure itself also gives part of the information.

The Basics of Language Descriptions.

A language is a class of trees together with means of displaying and entering these trees. A language description is a description of these three components of a language.

Describing the class of trees comprising a language consists of listing the possible productions of trees and describing the child groups of these productions. A production class is a set of productions; each child group of a production has associated with it a production class which limits the possible children in that group to those with productions in the associated class. There is also a production class associated with the root of the tree. Names of production classes are written in angle brackets.

Figure 4 lists the production classes and productions describing the tree structure of a simple document language. The production classes are listed with the productions belonging to them listed underneath. The child groups for each production are shown, with a class name alone indicating a solitary child group in which the child must have a production in the indicated class, while a class name enclosed in square, round, or curly brackets indicates an optional, series, or optional series child group respectively. By convention, the root of the tree is associated with the first production class listed.

Figure 4 may be paraphrased as saying that a document consists of a series of zero or more paragraphs; a paragraph consists of one or more sentences; a sentence consists of one or more parts; a sentence part may be one of several punction marks or it may be a word; a punctuation mark has no subcomponents; a word consists of one or more letters; and a letter may be a, b, c, etc.

Describing the manner in which trees are entered by the user consists mainly of specifying names for the various productions. These names are used to select which production is desired for a subtree. Production names may be any series of characters, but are typically a single character for ease of typing; they are written at the beginning of the description of a production and need be unique only within a production class.

Describing how trees should be displayed can be complex; only the

simple aspects will be described here. Productions are divided into two types for display purposes - those which should be displayed on separate lines indented from the surrounding text, and those which may be displayed as part of a line. The types are distinguished by writing | or - between the production name and the rest of the description of the production. Characters may be interspersed among the descriptions of the child groups of the production; the display of a tree with that production consists of the display of the children with these characters added in the indicated positions. Characters may also be placed before the class name within the description of an optional child group; these are displayed only if the group actually contains children. Characters after the class name within the description of a series child group are used to separate the displays of the children of that group.

Figures 5, 6, and 7 contain a description of a simple programming language, an example tree belonging to the language, and the way that tree would be displayed by the editor. The "include" construct in the description language allows one to include in a production class all or some of the productions of another production class. The productions of the predefined class <character> represent the various characters of the character set. This example will be used in the following sections to illustrate the editing commands.

```
<document>
  production for a document: {<paragraph>}

<paragraph>
   production for a paragraph: (<sentence>)

<sentence>
   production for a sentence: (<part>)

<part>
   production for a word: (<letter>)
   production for comma: no child groups
   etc.

<letter>
   production for the letter a: no child groups
   etc.
```

Figure 4. A partial language description for documents.

```
<statement>
  # | #
  I | if <expression> then(<statement>){else<statement>}fi
  W | while <expression> do(<statement>)od
  = | <identifier> := <expression>
  P | print <expression>
  R | read <identifier>
  B | begin(<statement>)end

<identifier>
  # - (<letter>)

<expression>
  include <identifier>
  N - (<digit>)
  + - <expression>+<expression>
  - - [<expression>]-<expression>
  * - <expression>*<expression>
  / - <expression>/<expression>
  < - <expression> < <expression>
  > - <expression> > <expression>
  = - <expression> = <expression>

<letter>
  include <character> a, b, c, d, e, f, g, h, i, j, k, l, m,
   n, o, p, q, r, s, t, u, v, w, x, y, z

<digit>
  include <character> 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
```

Figure 5. A description of an example language.

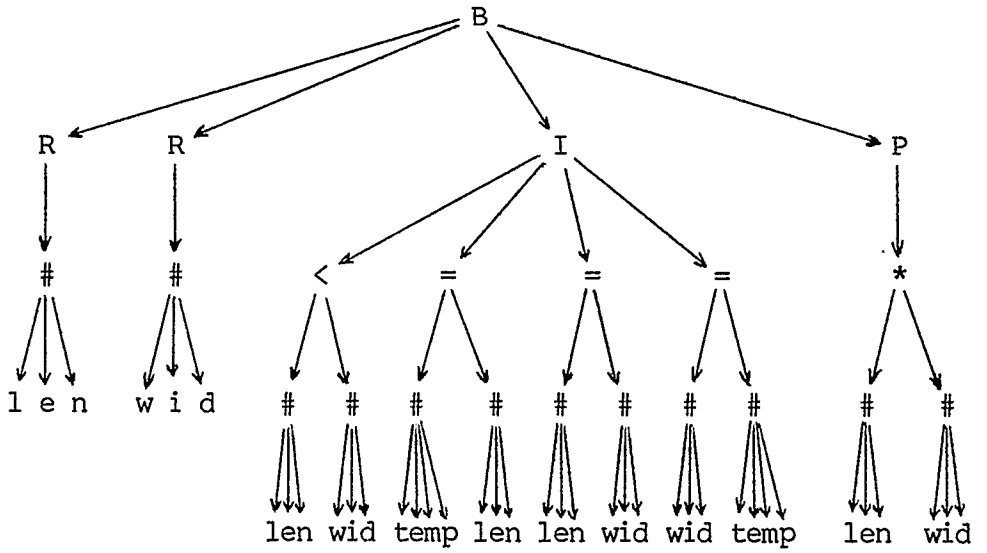


Figure 6. A tree of the example language.

```
begin
  read len
  read wid
  if len < wid then
    temp := len
    len := wid
    wid := temp
  fi
  print len*wid
end
```

Figure 7. The display representation of the tree of figure 6.

Basic Operation of the Editor.

When the user starts the editor, tailored for a particular language, he gives it a file containing a representation of a tree of that language (not a file containing the textual representation of the tree). The editor displays the textual representation of that tree (or as much of it as it can) on the screen before the user. The user then types commands that modify the tree, eventually typing the Quit command, at which time the editor writes the modified tree back out to the file.

The commands the user types are single keystrokes, i. e. they are typed by hitting a single key, possibly while holding down a shift key of some sort. A list of the commands, and the keystrokes used to invoke them, is given in appendix D. In this exposition, the commands will be referred to by their names, e. g. Quit, Cancel, InsertB. These names are not actually typed; they are used here to make the exposition clearer. Generally, after each command is typed, the display on the screen will be immediately changed to reflect the effect of the command.

If the editor is unable to perform the action requested by a command because the context is such that the command does not make sense, it rings the terminal's bell and ignores the erroneous command. If the user decides that a command he typed (and the editor acted upon) was a mistake, he may type the special command, Cancel, which undoes the effect of the previous command. Cancel may be typed several times

to undo several commands (erroneous commands that the editor ignored do not count here). The effect of Cancel may be thought of as going "back in time" to the point before the cancelled command was typed.

In addition to the textual representation of the tree, the editor also displays a cursor. The cursor designates a particular subtree of the tree being edited and is used as a reference point for the operation of the editing commands. It is drawn as two brackets at the corners of the textual representation of the subtree at which it is positioned. If the cursor is positioned at a subtree whose production has a display type that indicates it should be displayed on a line (or lines) by itself, the two brackets are drawn at the upper and lower left corners of the display of that subtree. If the display type is such that the object the cursor is at does not deserve a line to itself, the two brackets are at the upper left and lower right corners.

The cursor also indicates by its shape whether it is positioned at a last and/or first child. If it is at a first child, one of the sides of the first bracket will be lengthened; if it is at a last child, one of the sides of the second bracket will be lengthened; if it is at an only child, both brackets will have lengthened sides. (These subtle distinctions may not always be apparent in the figures.)

The cursor may be moved about in the tree by the Up command which moves the cursor to the parent of the subtree it was at previously, the DownLeft command which moves to the first child, the DownRight command which moves to the last child, the Right command which moves to the

next sibling, and the Left command which moves to the preceding sibling. Errors, resulting in the bell ringing, occur if one types Up when at the root of the tree, DownLeft or DownRight when at a leaf, or Right or Left when there is no sibling in that direction.

Figure 8 shows the an example of the use of these commands. This and other such figures show a series of "snapshots" of the screen as it would appear after each keystroke. The sequence is shown by the arrows, which have beside them the names of the commands which were typed between the two screens.

Entering and Modifying Trees.

The basic means of changing a tree with the editor is to replace existing subtrees with newly created subtrees. The user positions the cursor at a subtree he wishes to change and then types a production name. The subtree at the cursor is replaced by a subtree with the specified production and with holes for all its children. A hole is a subtree whose production name is "#" and whose children are all holes themselves. If the language description does not define a production with name "#" for a particular production class, the editor will assume one without any child groups and which is displayed as "#". Generally, if a production class contains only one production, it will be made a hole production so that the user will not have to select it explicitly. This was done with <identifier> in figure 5.

This is illustrated in figure 9. In the first screen shown, the

```
begin                                   begin
  read len                                read len
  read wid                                read wid
  if len < wid then                       if len < wid then
    temp := len        DownRight            temp := len
    len := wid        ------->              len := wid
    wid := temp                             wid := temp
  fi                                      fi
  print len*wid                           print len*wid
end                                     end
```

```
                          Left
                        <-------
begin                                   begin
  read len                                read len
  read wid                                read wid
  if len < wid then                       if len < wid then
    temp := len        DownLeft             temp := len
    len := wid        ------->              len := wid
    wid := temp                             wid := temp
  fi                                      fi
  print len*wid                           print len*wid
end                                     end
```

```
                          DownLeft
                        <-------
begin                                   begin
  read len                                read len
  read wid                                read wid
  if len < wid then                       if len < wid then
    temp := len        DownRight            temp := len
    len := wid        ------->              len := wid
    wid := temp                             wid := temp
  fi                                      fi
  print len*wid                           print len*wid
end                                     end
```

```
                          Up
                        <-------
begin                                   begin
  read len                                read len
  read wid                                read wid
  if len < wid then                       if len < wid then
    temp := len        Right                temp := len
    len := wid        ------->              len := wid
    wid := temp                             wid := temp
  fi                                      fi
  print len*wid                           print len*wid
end                                     end
```

Figure 8. Illustration of the cursor movement commands.

user has positioned himself at the "while" statement. When he types "I" the subtree representing the "while" statement is replaced by a subtree representing an "if" statement in which all the components are holes. Note that the editor also moves the cursor to the first of these holes; this is done automatically so that the user will be in a position to fill in the details of the "if" statement.

The remainder of figure 9 shows how the user can continue filling in holes until the entire "if" statement has been entered. When the "a" was typed when the cursor was at a hole for an identifier, the editor assumed that the "a" was intended as the first letter of the identifier, since "a" was not a valid production name for an identifier but was a valid production name for a letter. Notice that the editor then added a new hole after the "a". This was done because the language description specifies that an identifier consists of a series of letters, not necessarily just one. The new hole was added to allow the user the option of typing another letter if he wished. In this case, he did not want another letter in the identifier, so he typed the Remove command to delete the unwanted hole.

A similar situation arose when the user finished with the "then" part of the "if" statement. The language description specifies that there is an optional "else" part of the "if" statement, so the editor inserted a hole to allow the user to enter this part.

Although this example has shown the user persisting in entering the statement until all parts have been filled, it is permissable to leave

```
begin                    begin                    begin
  read a                   read a                   read a
  while a > 0 do           if # then                if # < # then
    print a                  #                        #
    a := a-1     --I-->    fi          --<-->       fi
  od                       print b                  print b
  print b                end                      end
end                                               end
```

a

```
begin                    begin                    begin
  read a                   read a                   read a
  if a# < # then           if a < # then           if a < b# then
    #                        #                        #
  fi           --Remove-->  fi          --b-->       fi
  print b                  print b                  print b
end                      end                      end
```

Remove

```
begin                    begin                    begin
  read a                   read a                   read a
  if a < b then            if a < b then            if a < b then
    #            --P-->      print #      --b-->      print b#
  fi                       fi                       fi
  print b                  print b                  print b
end                      end                      end
```

Remove

```
begin                    begin                    begin
  read a                   read a                   read a
  if a < b then            if a < b then            if a < b then
    print b                  print b                  print b
    #            --Remove-->  else        --P-->      else
  fi                         #                        print #
  print b                  fi                       fi
end                        print b                  print b
                         end                      end
```

a

```
begin                    begin                    begin
  read a                   read a                   read a
  if a < b then            if a < b then            if a < b then
    print b                  print b                  print b
  else         --Remove-->  else        --Remove-->  else
    print a#                 print a                  print a
  fi                         #                      fi
  print b                  fi                       print b
end                        print b                end
                         end
```

Figure 9. Illustration of the entry of a subtree.

a subtree with holes in it and move off to do something else.

The Remove command mentioned above can be used to remove any optional subtree, not just holes. The reverse of a Remove command is an insert command, of which the editor has two varieties - InsertA for inserting a hole after the spot where the cursor is positioned and InsertB for inserting before the cursor. The user would generally type a production name to fill in the hole after one of these commands.

When a subtree which is not a hole is replaced by a new subtree, the editor saves the old subtree. The most recently used erased subtree may be used to replace a subtree somewhere else in the tree by typing the UnErase command. The children of the last erased subtree may be successively inserted into the tree by typing UnErasNxt.

The InsertB and UnErasNxt commands are illustrated in figure 10.

Further Aspects of Language Descriptions.

Appendices A through C contain real language descriptions for Pascal, a class of documents, and the language-description language itself. This section will describe the additional details of language descriptions used there.

First, production classes need not be defined all in one place. There may be several sections modifying a production class, with the productions listed accumulating until the end of the description. There is also a method of removing productions from production classes;

```
begin
 ⌈while a > 0 do
    print a
    a := a+1
 ⌊od
end
```

InsertB →

```
begin
 ⌈#⌋
 while a > 0 do
    print a
    a := a+1
  od
end
```

R →

```
begin
  read ⌈#⌋
  while a > 0 do
    print a
    a := a+1
  od
end
```

a →

```
begin
  read a⌈#⌋
  while a > 0 do
    print a
    a := a+1
  od
end
```

Remove →

```
begin
  read a
  ⌈#⌋
  while a > 0 do
    print a
    a := a+1
  od
end
```

Remove →

```
begin
  read a
  ⌈while a > 0 do
    print a
    a := a+1
 ⌊od
end
```

DownRight →

```
begin
  read a
  while a > 0 do
    print a
   ⌊a := a+1
  od
end
```

DownRight →

```
begin
  read a
  while a > 0 do
    print a
    a := ⌈a+1⌋
  od
end
```

- →

```
begin
  read a
  while a > 0 do
    print a
    a := ⌈#⌋-#
  od
end
```

UnErasNxt →

```
begin
  read a
  while a > 0 do
    print a
    a := a-⌈#⌋
  od
end
```

UnErasNxt →

```
begin
  read a
  while a > 0 do
    print a
    a := ⌈a-1⌋
  od
end
```

Figure 10. Illustration of tree modification.

the fact that at some point in the language description a production appeared to be a member of a production class is of no significance, only the members at the end of the description count (however, the "include" construct operates on the situation at the point in the description where it is placed).

Secondly, optional and optional series child groups may be declared "very optional"; this is indicated in the display of the language description by a question-mark after the opening square or curly bracket. Very optional children are sometimes automatically skipped by the editor. The "assume" declaration directs the editor to assume a particular production when certain characters are typed. Note that these features are part of the description of how trees are entered, <u>not</u> of the tree structure itself.

Lastly, there are several additional display features. The language description can specify that a new line should be started in the display of a subtree by placing a | in the desired position. A particular column may be skipped to with an item in exclamation marks. In addition to the two display types of productions already mentioned (| and -), a display type of * acts like - except that the editor will never split the display of that subtree between lines, and the display types 0, 1, 2, and so on up to 9 indicate a production with a certain precedence. If a subtree has a lower precedence than its parent, it will be displayed in parentheses. The example language described earlier needs this feature to eliminate ambiguities in expressions.

```
<statement>
  # | #
  I | if <expression> then(<statement>){else<statement>}fi
  W | while <expression> do(<statement>)od
  = | <identifier> := <expression>
  P | print <expression>
  R | read <identifier>
  B | begin(<statement>)end
  assume = before abcdefghijklmnopqrstuvwxyz

<identifier>
  # - (<letter>)

<expression>
  include <identifier>
  N - (<digit>)
  + 1 <expression>+<expression>
  - 1 [<expression>]-<expression>
  * 2 <expression>*<expression>
  / 3 <expression>/<expression>
  < 0 <expression> < <expression>
  > 0 <expression> > <expression>
  = 0 <expression> = <expression>
  assume N before 0123456789

<letter>
  include <character> a, b, c, d, e, f, g, h, i, j, k, l, m,
   n, o, p, q, r, s, t, u, v, w, x, y, z

<digit>
  include <character> 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
```

Figure 11. An improved description of the example language.

Figure 11 is a description of the example language using some of these features.

Automatic Actions of the Editor.

When the editor receives a command which cannot be directly executed, it attempts to produce a situation in which the command will be valid by performing certain actions automatically. After the basic action of a command has been performed, the editor may perform certain additional actions designed to allow the smooth entry of trees.

The principle situation in which automatic actions before the basic command action are needed is when the user types a printing character and the character is not a valid production name for that point in the tree. The editor will first try changing the subtree at that point to the hole production, descending to the first hole child of the new subtree and trying again to interpret the character as a valid production name. This may continue several levels downward, terminating when the editor finds a place where the character typed is valid or when the hole production has no children. If trying the hole production fails at any point in this process, the editor will try assuming the production associated with the character typed by an "assume" entry in the language description. In this way the editor searches for a set of production names which when assumed produce a situation in which the character actually typed is valid. If the above process does not resolve the situation, the editor will try removing

the hole where the cursor is positioned, performing the automatic actions which would be normal after a Remove command, and interpreting the character typed as a production name at the new cursor position.

The effect of all this is to eliminate the need for the user to type a production name explicitly when the editor can deduce the desired production from the requirement that the character actually typed by the user become valid. One exception is the action taken when a space is typed. If the space cannot be directly interpreted as a production name, the editor does not follow the above procedure but instead treats the space almost as if it had been a Remove command. The slight difference is described below.

Automatic actions performed after the basic action of the command are intended to create the situation necessary for the user to continue entering subtrees smoothly. These follow-up actions may be done after any command, but always result from indicators set as a result of typing a production name. When a production name is typed, the basic action is the setting of the subtree at that point to that production with holes for children. The cursor is left at the same place by the basic action. However, indicators are left at that point in the tree which cause the cursor to be moved to the first child of the new subtree as a follow-up. These indicators also cause the cursor to move right when it is next positioned at the newly created subtree and cause a new hole to be inserted after if permissible in that context. Similarly, when an optional hole is removed, the editor will

automatically insert a hole belonging to the next child group if there was not one already and the indicators are appropriate. The editor will not insert such a hole if the child group is very optional and the removal was the result of typing space. This allows the user to avoid having to be concerned with infrequently-used options.

Window Manipulation.

The editor provides a facility whereby the user can edit two trees concurrently in two separate "windows" of the screen. This facility is convenient when rearranging the order in which objects occur in a tree and when merging or splitting programs or documents.

When the editor is started, the user may give the names of two files containing trees rather than the name of just one file. The screen will then be divided in half by a dashed line, with the first tree displayed in the area above the line and the second tree displayed below the line. If the Quit command is typed at a time when there are two windows the two trees are saved in the two files mentioned at start-up. The trees have their own cursors which may be moved independently. Initially, the cursor of the top tree will be slightly brighter than that of the bottom tree. This indicates that the user is currently "in" the top window and that all cursor movement and tree modification commands will apply to the tree in that window. The Switch command allows the user to switch to the other window and manipulate the other tree.

Objects may be copied from one window to the other with the Copy command. This command sets the subtree at the cursor of the window the user is "in" to the subtree at the cursor of the window that the user is not "in". The Automatic follow-up actions are such that the user may copy several items of a series from one tree to the other by simply typing Copy several times. It is also possible to move data from one tree to the other by using the UnErase and UnErasNxt commands.

The user may discard the tree in the lower window (and get rid of the window itself) by typing the Discard command. Conversely, if there is only one window, the user may type the Dup command to create a second window which will initially contain a copy of the tree in the other window. This allows the second window to be used as a "scratch pad" when editing a single tree.

Figure 12 shows an example of how these commands work.

Elision.

If the full display of the tree is too large to fit on the screen (or half the screen if there are two windows), the editor must <u>elide</u> certain parts of the textual representation of the tree. These portions are replaced with elipses (...). The user need not be especially concerned with how the editor decides which parts of the display to elide; the general principle is that areas near the current position of the cursor are shown while areas far from the cursor are elided. "Near" and "far" in this context refer to distance in the

```
begin
 ⌈c := d
  e := f
  g := h
end
```

Dup →

```
begin
 ⌈c := d
  e := f
  g := h
end
```

Remove →

```
begin
 ⌈e := f
  g := h
end
```

```
begin
 ⌈c := d
  e := f
  g := h
end
```

W →

```
begin
  while ⌈# do
    #
  od
  g := h
end
```

```
begin
  while # do
    ⌈#
  od
  g := h
end
```

```
begin
  while # do
    c := d
    ⌈#
  od
  g := h
end
```

Right →

```
begin
 ⌈c := d
  e := f
  g := h
end
```

Copy →

```
begin
 ⌈c := d
  e := f
  g := h
end
```

```
begin
  c := d
 ⌈e := f
  g := h
end
```

Copy →

```
begin
  while # do
    c := d
    e := f
    ⌈#
  od
  g := h
end
```

```
begin
  while # do
    c := d
    e := f
  od
  ⌈g := h
end
```

```
begin
  while # do
    c := d
    e := f
  od
  ⌈g := h
end
```

Remove →

```
begin
  c := d
  e := f
 ⌈g := h
end
```

Discard →

```
begin
  c := d
  e := f
 ⌈g := h
end
```

Figure 12. Illustration of the use of windows.

tree, not distance in the textual representation of the tree.

Figure 13 shows the full display of a program in the example language and how it would appear when forced into an twelve-line section of the screen when the cursor is at the root and when the cursor is at the "while" statement.

The Macro Language.

This section describes a method whereby the user may extend the description of the language he is using by adding new productions and new production classes, write programs in the extended language, and have these programs converted to equivalent programs in the unextended language for input to the compiler.

Figure 14 shows an extension to the example language used before to provide a "for" statement similar to such statements in other programming languages. The statement has a number of optional parts. If no "from" clause is present, the iteration starts with whatever value the variable already had; if no "by" clause is present, a step of 1 is assumed; if there is no "to" clause, the iteration has no limit. Figure 15 shows the macro definitions used to expand statements of this type into combinations of other statements in the language. Figures 16 and 17 show a program in the extended language and the program which results from expanding the "for" statements using the definitions in figure 15.

```
begin
  read n
  if n < 0 then
    factorial := 0
  else
    factorial := 1
    while n > 0 do
      factorial := factorial*n
      n := n-1
    od
  fi
  print factorial
end
```

```
begin
  read n
  if n < 0 then
    factorial := 0
  else
    factorial := 1
    while n > 0 do
    ...
  fi
  print factorial
end
```

```
begin
  read n
  ...
  else
    factorial := 1
    while n > 0 do
      factorial := factorial*n
      n := n-1
    od
  fi
  print factorial
end
```

Figure 13. The effect of cursor position on elision.

```
<statement>
  F | for <identifier>[ from <expression>][ to <expression>]
   [ by <expression>] do(<statement>)od
```

Figure 14. An extension to the example language.

```
find statement
  for <identifier.a> from [expression.b] to [expression.c] do
    {statement.d}
  od
reduce to
  for <identifier.a> from [expression.b] to [expression.c] by 1 do
    {statement.d}
  od

find statement
  for <identifier.a> from <expression.b> to [expression.c]
   by [expression.d] do
    {statement.e}
  od
reduce to
  begin
    <identifier.a> := <expression.b>
    for <identifier.a> to [expression.c] by [expression.d] do
      {statement.e}
    od
  end

find statement
  for <identifier.a> to <expression.b> by <expression.c> do
    {statement.d}
  od
reduce to
  while <expression.a> < <expression.b>+1 do
    {statement.d}
    <identifier.a> := <expression.a>+<expression.c>
  od

find statement
  for <identifier.a> by <expression.b> do
    {statement.c}
  od
reduce to
  while 0 = 0 do
    {statement.c}
    <identifier.a> := <expression.a>+<expression.b>
  od
```

Figure 15. Implementation of the extension.

```
begin
  junk := 0
  for i from j to k by 1 do
    read x
    print x
  od
  for a to b do
    print junk
  od
  for lots do
    print lots
  od
end
```

Figure 16. A program using the extension.

```
begin
  junk := 0
  begin
    i := j
    while i < k+1 do
      read x
      print x
      i := i+1
    od
  end
  while a < b+1 do
    print junk
    a := a+1
  od
  while 0 = 0 do
    print lots
    lots := lots+1
  od
end
```

Figure 17. Expansion of the program of figure 16.

A macro definition such as is shown in figure 15 consists of a number of reductions. A reduction is of the form "find production-class pattern replace by template". The macro expander searches the tree being expanded for subtrees which match a pattern given in one of the reductions and replaces the subtree matching the pattern by a new subtree constructed according to the template. The pattern and the template look like (and may indeed be) ordinary subtrees of the production class specified at the beginning of the reduction (<statement> in the case of the reductions in figure 15). A pattern which is just an ordinary subtree matches only subtrees identical to it; a template which is an ordinary subtree causes the matched subtree to be replaced with just that subtree. However, most patterns and templates will contain special items which allow a pattern to match a wide class of subtrees and templates to cause replacements that vary with the pattern matched.

There are four types of special items, analagous to solitary, optional, series, and optional series child groups. They are entered by typing certain special production names and are displayed as angle, square, round, or curly brackets surrounding the production class at that point in the tree and an identifying character (the production class is for orientation purposes only). A special item in angle brackets matches any child which is actually there; a special item in square brackets matches any actually present child or the indicator that that child is absent; a special item in round brackets matches any series of children up to an end-of-series indicator; etc. The pattern

as a whole matches a subtree if the parts of the pattern other than the special items match the tree exactly, the special items all match individually, and all occurrences of special items with the same identifying character match the same thing.

Once a match has been found, the template is used to construct a replacement for the matched object by replacing all occurrences of special items in the template by the subtrees (or absence indicators) that those items matched in the pattern.

This may sound complicated, but in application it is quite simple. You simply give a "picture" of the kind of subtree you wish to change and a picture of what you want it changed to. In figure 15, the first reduction inserts a clause specifying a step size of one in "for" statements that lack a "by" clause; the second reduction handles the presence of a "from" clause; and the last two reductions expand "for" statements without a "from" clause but with a "by" clause in two different ways depending on whether a "to" clause is present. The first two reductions ensure that the conditions for applying one of the last two will be met.

THE PRESENT IMPLEMENTATION

The tree editing system is implemented on a Unix [Thomson and Ritchie, 1974] system running on a DEC PDP 11/60. Two types of terminals may be used with the editor. The principal terminal for which the editor was designed is a DEC VT11 vector graphics display attached to a PDP 11/40 acting as a satellite processor to the PDP 11/60. The editor may also be used on Infoton 400 terminals attached to the PDP 11/60, but with degraded display capability.

The editor and associated programs are written in the programming language C [Kernighan & Ritchie, 1978] with a few PDP 11 assembly language routines. They comprise approximately 5000 lines of code in totality. This code is organized into about 60 modules which are combined to form the various versions of the editor, the language-description compiler, and other programs of the system.

The PDP 11 has an address space of 64K bytes. This limitation and the requirement of instantaneous (one tenth of a second or less) response to keystrokes are the principal constraints which the implementation must meet.

Representation of Trees.

A representation of a tree must contain the production present at that point in the tree, a list of child trees, and an indication of how

the children are divided into child groups. It may be convenient to store other information in the representation also in order to speed up certain operations. Note that the amount of information required to represent the tree varies with the number of children.

This variable amount of information is represented in this implementation by a linked list of fixed size blocks. This simplifies storage reuse compared to the option of using variable-sized blocks. The "next" field of a block contains this link to the next block in the list holding the data on one tree.

When a new block with specified contents is required, the editor checks to see if a block with those contents already exists. If so, the old block is used for the new purpose as well instead of a new block being allocated. A reference count is stored in the "ref" field of a block to indicate how many places refer to that block. A hash table is used to determine rapidly if a block with specified contents already exists. Blocks with the same hash are linked into a ring via the "chain" field of a block. When a new block is required, a block with zero reference count is selected arbitrarily and its space used for the new block. A block with a reference count of zero is not necessarily destined to be reused however; its reference count may later increase if a block with the same contents is required or if the operation which reduced its reference count to zero is cancelled.

The "prod" field of a block contains a pointer to a structure describing the production of a tree if the block is the first in the

list for that tree. Blocks after the first contain zero in this field.

The "misc" field of a block contains various kinds of information depending on what phase of operation the editor is in at the time.

Each block holds a fixed number of "links", the number being specifiable at compile time, and usually about six. A link may be a pointer to the first block of a list of blocks describing a child tree, or one to indicate that an optional child is not present, or zero to indicate that the end of the child list has been reached. In this list of links, a solitary child group is represented by a single link pointing to the child, an optional child group is represented by a single link which either points to the child or contains the absence indicator, a series child group is represented by two or more links containing pointers to the children in the group terminated by the absence indicator, and a optional series child group is represented by one or more links as for a series group.

This structure is illustrated in figure 18.

The editor represents trees on files by a form of Polish prefix notation. Each production is assigned a <u>production number</u>. The representation of a tree on a file consists of its production number followed by the representation of its children, with special indicators for end-of-series and absent optional children. Repeated subtrees are represented only by a reference to the position in the file where they first occurred. Files containing trees are generally about the same

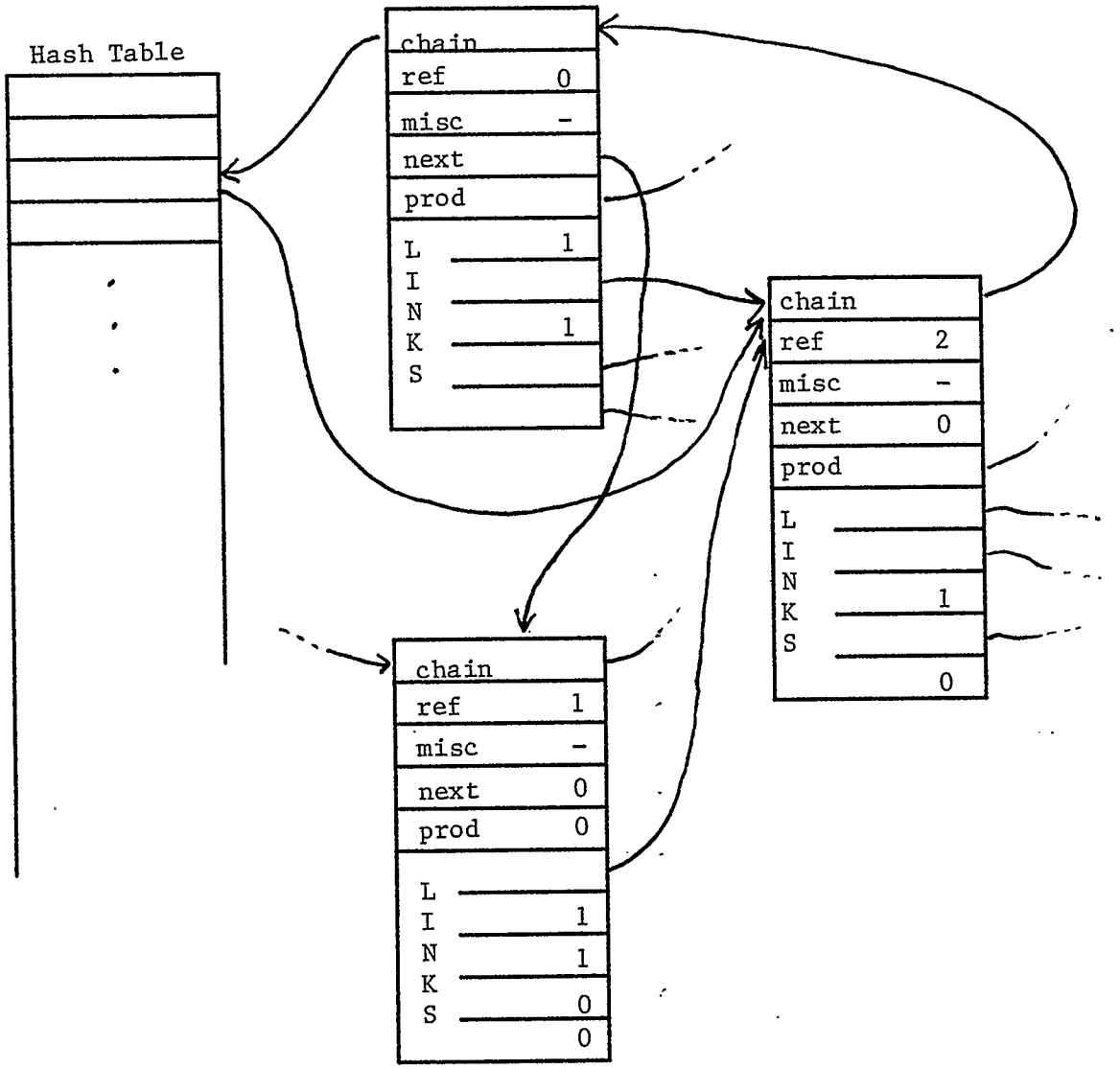


Figure 18. Structure of blocks.

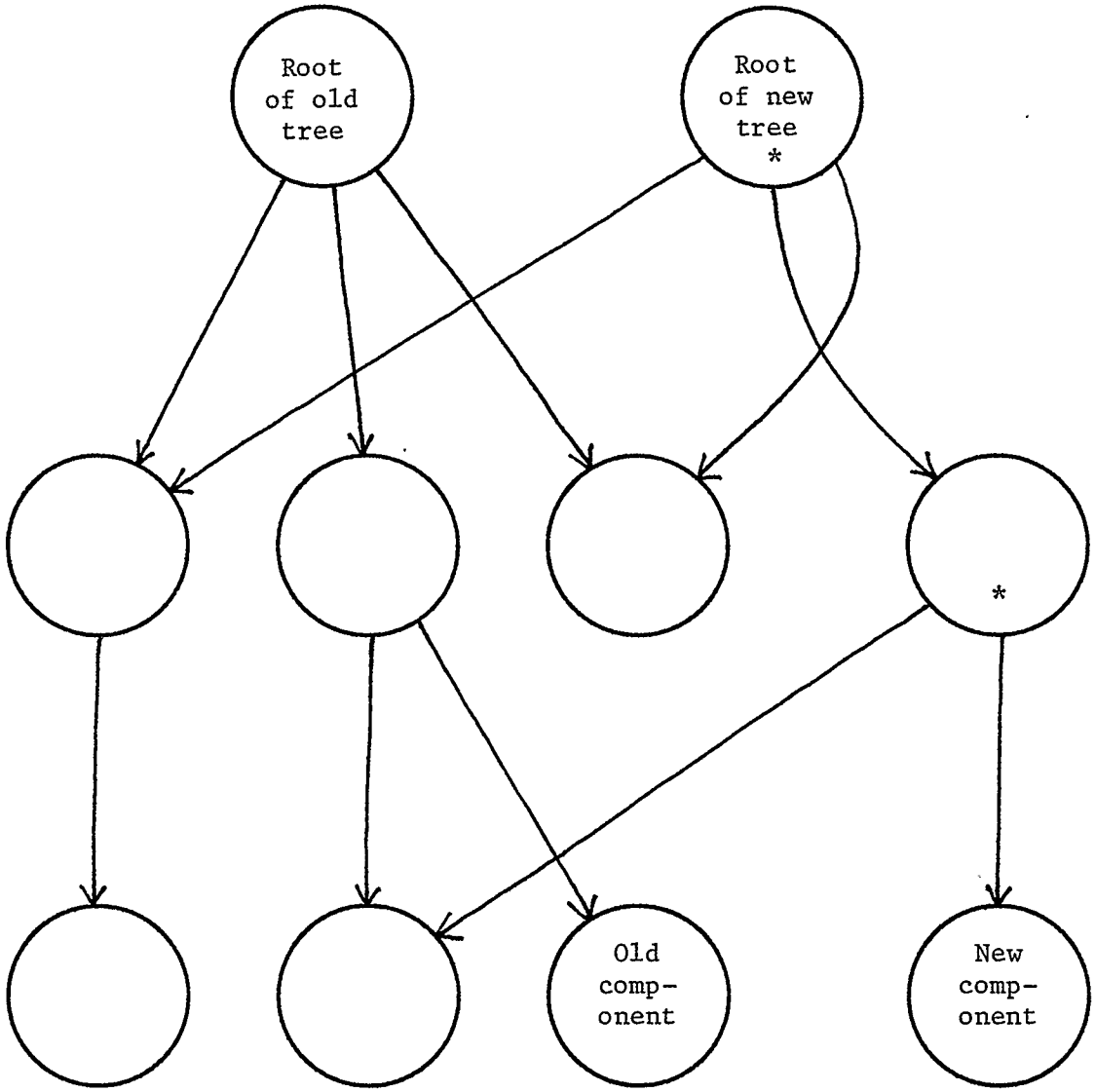
size as the corresponding text files, ranging from about fifteen percent larger for typical documents to a bit smaller for Pascal.

Operations on Trees.

Since the blocks used to represent trees may be shared, it is not possible to perform alterations in a tree by changing the contents of the blocks representing it. Instead, an entirely new tree must be created which incorporates the change, and this tree used to replace the old. If this is not done, then when the user attempts, say, to change the expression "a+b*a" to "a+b*c" the expression will instead change to "c+b*c" as the editor will be using the same block to represent the two occurrences of "a".

Figure 19 shows how this is done. Note that in creating the new tree, only the blocks representing the ancestors of the block being "changed" need be recreated, the rest of the new tree is shared with the old tree.

For similar reasons, the user's cursor may not be represented by a simple pointer to the block representing the subtree where the cursor is, because if the subtree occurs in the tree several times it would not be apparent at which of these occurrences the cursor was pointing. Instead, the cursor position must be represented as a set of directions for finding the cursor starting at the root. For example, the cursor might be described as being at the third child of the second child of the root of the tree.



* indicates newly allocated block list

Figure 19. Creation of a modified tree.

Producing the Display.

The central procedure used in producing the display of the tree is one which, when given a tree produces a display element list which describes how that tree should be displayed. This list contains elements of two types. The first type is simply a line of text. The second type is a reference to a subtree. The full (unelided) display of the tree is obtained by replacing these subtree references by the lines (indented) comprising the full display of the indicated subtrees. Note that these subtrees need not be immediate children of the tree nor will all children necessarily appear in the list. The controlling factor is the display type of the subtrees; a display type of '-' or '*' leads to the display of that subtree being simply incorporated into a line element; a display type of '|' causes the editor to include a reference element in the list without actually producing the display for that subtree.

These display element lists define another tree, related but not identical to the original. The leaves of this tree are the line elements; the non-leaf subtrees correspond to subtrees of the original tree whose display type was '|'. The position of the cursor in the original tree is transformed into a position at a subtree in this new tree or within a line that is a leaf of this tree.

The production of the elided display shown on the screen is accomplished in two passes. First a table is constructed which gives the number of line elements of the display tree which are found at each

distance (in the display tree) from the location of the cursor; this involves a traversal of the display tree. Based on this table and the known size of the display (or the window of the display being used), the editor can determine how close a line element must be to the cursor for it to be displayed. A second traversal of the display tree then produces a list of the line elements comprising the display.

This process is basically fairly simple, but is complicated by the details of how to calculate the distance function for siblings of direct ancestors of the subtree at which the cursor is located and by the fact that line elements which are not displayed still require space for their elipses. These factors are the prime reason why the elision method employed is rather simple-minded.

The display tree described above is not actually constructed. Instead, the procedure for finding the display element list of a subtree (of the original tree) is called again each time the list is required. On the other hand, this procedure does not necessarily actually calculate the element list each time it is called, as it would require several seconds to perform this calculation for all the subtrees of a large tree. A cache scheme is used to save the most recently calculated elements lists. During the main phase of the editor, the "misc" field of a block is used to point to the element list for the subtree that that block represents, if it has been saved.

Cancellation of Commands.

In order to allow cancellation of commands, the editor stores backup information on a stack held on disk. This stack holds a number of frames each consisting of a list of items and a count of how many items there are in the frame. Each item contains the information needed to reverse one action of the editor.

Normally, the editor puts two frames on the stack for every command (keystroke). The first of these contains items required to undo the basic action of the command, the second the items required to undo the automatic follow-up actions. The command to cancel the automatic follow-up of a command is implemented by reading the top frame off the stack and undoing actions as described in the items of the frame. The command to cancel the whole last command reads the top two frames.

The simplest items consist of two words - the address of a word of memory which has been changed by the editor and the previous contents of that word. Items of this form suffice to allow the reversal of anything the editor does, since all changes in the state of the editor are reflected as changes in certain memory words. Note that not all alterations of memory need be saved - in particular, changes to local variables that do not form part of the representation of the state of the editor need not be recorded.

To reduce the amount of data saved on the stack, specialized items are used to allow the reversal of certain operations. These items

consist of the address of a reversal routine and a varying number of words whose number and interpretation is determined by that routine. A simple example is the reversal of the incrementation of a variable. Using a basic item, this requires that two words be stored on the stack - the address of the variable and the contents of the variable before it was incremented. By using a specialized item, this can be reduced to one word - the address of a routine which will decrement that particular variable, reversing the effect of the incrementation.

The principal application of these specialized items is in the reversal of the allocation of a block. Undoing the allocation of a block requires taking the block out of its hash list, decrementing the reference counts of blocks pointed to by the contents of the block, restoring the previous contents of the block, incrementing the reference counts of blocks pointed to by the previous contents, and putting the block back into its old hash list.

An important property of this method of recording information for command cancellation is that the amount of information to be recorded is approximately constant, regardless of the command whose cancellation is being provided for. Deleting a large subtree does not cause the entire contents of that subtree to be saved to allow for the deletion command's cancellation. Instead, the reference count of the root of the subtree will be reduced to zero and the contents of the subtree will be saved a bit at a time as subsequent commands reuse the blocks which made up the subtree. The commands which cause the reuse of these

blocks must necessarily be cancelled (and the blocks restored to their previous state) before the deletion command can be cancelled. This contributes to a constant response time, as saving a large amount of information on disk would be time consuming.

Typically, each keystroke results in about one hundred bytes of information being written to disk to allow for the keystroke's cancellation. The transfer rate of the floppy disk used is about 5000 bytes/second, so time to write cancellation information to disk is not a factor in supporting a response time of a tenth of a second. Some response time problems arise when actually cancelling commands due to peculiarities of reading the disk backwards.

Finally, the cancellation mechanism is also used to back out of error situations and to backtrack when exploring different possible automatic actions.

RATIONALE FOR DESIGN DECISIONS

In this section I will explain the reasoning behind some of the design decisions taken and indicate where I feel better alternatives are possible. More speculative suggestions are deferred to the next section.

The editor described in the preceding sections was the second version I implemented. The first version was completely experimental and is described in [Neal, 1979a]. Although it was based on the same principles, it differed from the present version in a number of ways. Most notably, the implementation was totally different. It lacked the window facility of the present editor and was in many ways less general.

The method used in designing the tree editor was subjective evaluation of the alternatives available, supplemented with trial and error. The creation of a first version, intended from the beginning to be discarded, is naturally the most prominent example of trial and error, but the method was used on a smaller scale also. In particular, the details of how the automatic actions work required a lot of iterations to get right. In general, however, it is just not practical to implement all the alternatives in order to try them out.

Requirements of the Design.

The primary application of the tree editor is the modification of computer programs by expert programmers. One expects that the programmers would use the program very frequently (several hours a day, on and off). Time to learn the editor is thus less important than the ease with which it can be used by experts. I have assumed that the user types in the conventional, skilled manner (two-handed, all four fingers), but typing another way should not cause great problems.

As I am also interested in the use of the editor for document preparation and the manipulation of other information, I hope that people other than programmers will be able to use the editor, but this was not a principal design goal. The possibility of using the editor in teaching programming (thereby constraining the student to producing syntactically correct programs) is another interesting possibility which was not considered a primary goal. The editor is thus intended primarily as a tool for expert programmers.

Programmers using batch systems and interactive systems with hard-copy terminals usually write and revise their programs by hand and only then enter them into the machine. Minor modifications may be made directly, but major modifications are generally scribbled on listings or otherwise written out. When a good screen-oriented text editor is available, many programmers dispense with the step of writing their programs on paper and compose them using the editor directly; this is possible because of the superiority of the screen text editor for

performing modifications. The behaviour of a programmer using an editor to compose a program will be quite different from that of a programmer entering an already-written program, since he will make false starts and skip about filling in portions of the program non-sequentially. The tree editor is intended to be used in this later manner, an important point because, if the user has text written out before him, the pressure to think of the program as text rather than a tree is greater, particularly if he is a good typist.

The most basic requirement if a user is to successfully use a system is that he have a proper conceptual model of objects and operations the system deals with. For the tree editor, the user must understand the forms of trees and the operations that may be performed on them. A programmer should not find this difficult, since trees are an ubiquitous data structure throughout computing science. The prevalence of hierarchical structures in the ordinary world, for example, organizations with committees which have subcommittees, shows that the ideas are familiar to non-programmers also; trees are a basic method of organizing complexity.

Simplicity is an important design goal since it speeds learning, reduces unexpected occurrences, and makes implementation easier. [Nelson, 1974] advocates a "ten-minute rule", which states that a user should be able to learn the basics of using a human-computer system in ten minutes given a working system and a knowledgable demonstrator. This may be a bit too restrictive, but is certainly to be strived for.

Uniformity and generality are important means of achieving simplicity; there should not be two commands if one will do; there should be no arbitrary restrictions on when commands are valid.

The user should be provided with feedback on what he has done and what his current situation is. The goal for this project was to provide instant feedback; within a tenth of a second or less of his typing a key. When this feedback indicates to the user that he has made a mistake, it should be easily correctable.

The editor should allow the user to perform his tasks quickly. This is particularly important with an intensively-used system such as an editor for programs. A very rough estimate of how long a task takes can be obtained by counting the number of keystrokes required, provided the limitations of this approach are kept in mind.

Basic Choices in the Design of an Editor.

Several fundamental design decisions must be made before an editor (of either text or trees) may be designed in detail. The first of these is <u>interaction style</u> - the basic way in which information is communicated from man to machine and vice versa. One may distinguish between <u>immediate</u> and <u>delayed</u> interaction style. With the former, the machine acts on the user's input immediately and communicates the results of these actions to him immediately. With the latter style, the user's input is not acted on until she indicates that this should be done and the machine does not communicate the results of commands to

her until she explicitly requests it. Delayed interaction is typical of time-sharing systems due to technical constraints and lack of imagination. Common sense and established principles dictate that delaying feedback to the user is harmful, and this judgement is confirmed by the near-unanimous opinion of those who have used text editors based on both styles.

The input method is related to the interaction style. Delayed interaction is usually accomplished with the keyboard as the input medium, with the keystrokes typed representing a command which may well have a fairly complex structure. With immediate interaction a keyboard may also be used, but the keys will generally have independent interpretations, not be simply components of a more complex command. Menus are another input method - the machine displays a list of options and the user selects one of these in some manner. Note that although a keyboard may be used to select a menu item, this use is fundamentally different from the use of keys to represent commands; with menus, the user is presumed not to know which key is used to select which option, or there would be no point in displaying the menu. Menus are inappropriate for this tree editor because the desired speed of information entry cannot be achieved if the user must search a menu before each keystroke.

The designation method used is another fundamental property of an editor. Most commands to the editor operate with respect to a particular position in the object being edited - delete that, insert

here, etc. Also, an immediate style editor must somehow decide which part of the object being edited should be displayed at any time. A cursor provides a solution to both these needs - all commands act where the cursor is positioned and areas "near" the cursor are displayed. Of course, the user may desire independent control of these two aspects, in which case two cursors could be used, or a cursor could be used to control the display while the objects of commands could be indicated by pointing (with a light-pen, say) or by naming a label associated with the object (see [Neal, 1979b]). A single cursor was used in this project for simplicity and because the alternatives seemed to have no clear advantages; testing other methods would be desirable.

A third basic choice is whether the editor is to operate in several modes, with a different set of valid commands for each mode, or whether all commands are to be valid at all times (and have the same effect). Particularly popular with delayed-interaction text editors is a special "input mode" used for entering text in which all normal commands are disabled. Some immediate-interaction text editors have an "insert mode" and a "replace mode" which controls the manner in which input text effects the file. I feel that such distinctions intrude upon the user and make it difficult for editing operations to become automatic, since the user must always remember in what mode the editor is operating and adjust accordingly. The sole justification of multiple modes would be to break up a complex task into simpler tasks, but editing should not be a complex task.

Finally, one must consider the means the user has to correct the editing errors he inevitably makes. The general cancellation scheme employed in this project is a near-perfect solution to this problem and eliminates the need for redundancy in editing commands (difficult with single-keystroke commands in any case).

The Language-description Language.

A language description consists of three parts - the forms of trees in the language, the way these trees are entered, and the way they are displayed.

The first aspect is one of the few features of the editor which has changed hardly at all from the beginning of the project. To simplify the possible forms of trees by allowing only solitary child groups would lead to unnatural tree structures, even though such trees are theoretically capable of representing any information. Conversely, to add further types of child groups, say a child group in which two or more children are required, would complicate the structure without adding significant benefit, as such groups would find little application in present programming languages. If such a requirement on the number of children is present in the language it would make more sense to consider it to be a context-sensitive constraint and to deal with it via some general facility for detecting such errors (as discussed later).

One change I did consider seriously was to restrict series child

groups to being the only child group of their production. This would simplify certain aspects of the implementation and may be wise anyway because the user is likely to think of a series as a separable entity. I rejected this option because it can increase the depth of the trees undesirably; for example, the Pascal "with" statement allows a series of record variables to be specified, but almost all actual "with" statements specify only one; the user would continually be descending an extra level to reach the variable if the series had to be a separate entity.

Another possibility would be to treat comments specially and allow them to be tacked on to any part of the tree. In actual programs, most comments are in conventional positions and can be handled quite adequately by having the language description allow (or require) comments in these particular places, so I rejected any special feature as unnecessary.

The fundamental problem concerning the display aspect of the language descriptions is whether one should attempt to reproduce completely the usual manner in which all programming languages are displayed or whether one should instead aim at simply allowing <u>some</u> display of the tree which is easily read and which makes the tree structure evident. I now incline toward the latter view. The earlier version of the editor allowed more freedom in how the tree was displayed, but this caused implementation problems because the display of a subtree could not be calculated independently of its parent and

the facility obscured the tree structure, if actually used, in any case. This latter point is important because the user must be encouraged to perceive the program in terms of trees if the whole tree editing concept is to work.

Production names are the main aspects of the language descriptions concerned with how the program is entered. The first version of the editor allowed only single-character production names. This was somewhat confining, so multi-character production names are allowed in the current version. Nevertheless, single-character names should be used whenever possible simply because they can be typed quickly. If multi-character production names are used, it is best if one production name cannot be obtained from another by adding characters at the end, as the editor then cannot automatically determine when a production name is completed.

Choosing production names for a language can be difficult, particularly if the language has several statements whose keywords begin with the same letter or several operators which commence with the same character. C, for instance, has five operators which begin with a minus sign. One possible solution, which has been experimentally implemented, is to give several productions the same production name and assume the first production when that name is typed; if the user sees that was not the production he wished, he types a special character and the next production is assumed; if he is satisfied he continues just as he normally would. This seems to work better with

keyword-style statements than with operators.

The "very optional" aspect of the language-description language was included to handle certain frequently used constructs of common programming languages characterised by optional components which are (relatively) rarely included and which would have to be explicitly deleted by the user each time if it were not for the "very optional" feature. It is somewhat of a fudge, but the added complication seems justified considering the number of keystrokes saved. A better way to handle the situation would be desirable.

The Editing Commands.

The editing commands are the central objects of the editing system. It is through them that the user interacts with the editor; it is they that determine more than anything else the acceptability of the system.

Appendix D lists twenty-two editing commands, half are basic and half are more-or-less non-essential embellishments. This number compares favorably with the complexity of the command languages for text editors; although some text editors have fewer commands, most have considerably more. The complexity of the individual commands is more difficult to evaluate. The basic actions of all the commands are simple, but the automatic actions performed by the editor before and after the basic actions can be fairly complex. Since these automatic actions are intended to anticipate what the user wishes they hopefully

will not be seen as a source of complexity by the user. On the whole, I think the commands themselves are rather simple, although using them skilfully may take some practice (as is the case with text editors). Any modifications jeopardising this simplicity should be regarded with suspicion.

The manner in which the commands and the automatic actions accompanying them allow programs to be entered in a smooth way is particularly worthy of note. In this way, the need for a special editing mode for entering subtrees, with the problems any such mode brings, is avoided. An example of the way the commands dovetail for this purpose is the action of the Remove command when the object removed is the last child of its parent. Initially, one might feel that the cursor should be positioned at the new last child after this operation; instead, the cursor is moved to the parent of the object that is removed. The cursor will then encounter any indicators of automatic actions set at the parent to provide for continued entry of subtrees.

The CancelAuto command is also related to the entry of new trees. Its primary use is to cancel an automatic action which the editor made on the assumption that the user wished to continue entering subtrees, when in fact, the user wishes to go off and do something else. The point to note here is that the editor cannot read the mind of the user; all it can do is to make it easy for the user to do what he wishes. To have a special mode for entering a subtree is to assume that the user

will persist in this task until the entire subtree has been entered; an unwarranted assumption, since flitting about making corrections and revisions is normal behaviour when composing a program. Without a special input mode, this flitting about is easier, even if occasional automatic actions must be cancelled. Refining the automatic actions to eliminate some unwanted ones may still be possible, however.

The command for repeating the previous command indefinitely provides a simple means of performing certain operations such as deleting the remainder of a series or moving to the root of the tree, for which one might otherwise be tempted to provide special commands. The RepeatFive command also provides a powerful capability for reducing tedious keystrokes without adding much complication to the command language.

The principle behind the interaction of the Cancel command with the repeat commands is that any single keystroke error should be correctable by a single keystroke. If one were allowed to repeat the Cancel command by typing Cancel followed by RepeatFive, the RepeatFive key could be undone only by typing the five keystrokes that were cancelled. In any case, the Cancel would not be a true "movement back in time" if the RepeatFive key did that. Instead, it must repeat whatever key was last typed at the time back to which the Cancel command has taken us. Similarly, a Cancel after a RepeatFive must cancel all five repetitions, not just one. Allowing for the cancellation of Cancel commands would greatly increase the complexity

of the command structure and of the implementation, because the history of the editing session could no longer be regarded as a simple one-dimensional progression but would instead have to incorporate a tree structure of cancelled command sequences.

In general, complex rearrangements of the program are to be done by duplicating the tree and using the Copy command. The UnErase and UnErasNxt commands are included to allow certain common operations to be performed without the bother of using this procedure. In particular, such operations as changing the expression "fred+george" to "fred-george" or changing "mary" to "mary*10" may be easily perfomed using these commands.

The reader may wonder what the purpose of the Null command is, since it does nothing. One use arises if, after typing CancelAuto, the user wishes the automatic actions he cancelled to come back. Typing Null will cause this to happen since the indicators specifying the automatic actions are still present and these actions are performed after the Null command just as they are after other commands. Another essential use is in terminating a production name when another production name for that class has the production name typed so far as a front part. Any command other than a printing character will cause the editor to assume that the production name is finished, but the user may not wish anything done other than the specification of that production.

The Keyboard Layout.

The particular assignment of commands to keys is significant for two reasons - certain keys are easier to type than others, and the association of a key with a command should, ideally, be easy to remember. The first of these is more important, since for a typical user, the time to learn will be much less than the time spent as an experienced user.

All editing commands are associated with control characters which are typed by simultaneously pressing the control key and another key (almost always a letter). Note that all non-control characters are required for use as production names. Several of the control characters can also be generated by pressing a special key; in particular, RETURN is control/M, LINE FEED is control/J, ESC is control/[, and BACK SPACE is control/H. The local Unix system uses RETURN to indicate the end of a line and BACK SPACE to erase a character; the use of these characters becomes quite ingrained in users and it is thus imperative that they be used for the analogous operations in the tree editor. Since there is only one control key, at the left side of the keyboard, control characters on the right side are easier to type than those on the left. Less obviously, control characters in the middle of the left side are easier to type than others on the left side. (Note: these observations assume that the user is a touch typist; two-fingered techniques may lead to different results.)

The keyboard layout used is shown in figure 20. An attempt has been made to distribute the commands so that the most frequently used are in the most easily used positions and so that related keys are near each other (for mnemonic reasons). For example, the cursor movement keys are grouped in an easily typed position because they are very frequently used. Within that group, the Left and Right keys are on the "home" row because they are the most heavily used of the group (due to their use in scanning through lists). The Left and DownLeft keys are at the left side of the group and the Right and DownRight keys at the right side for ease of remembering. Almost no attempt has been made to associate commands with control characters for which the corresponding letter has mnemonic significance. Attempts at this inevitably encounter problems with the non-uniform distribution of first letters of English words.

The lengths that the editor goes to to use the space key to mean various things is justified by the fact that space is the easiest character to type. It also has a conventional meaning in documents which it is desirable to preserve if possible.

The Display.

Three aspects of the display merit discussion here - the elision method employed, the display of the cursor, and the handling of windows.

I am not very satisfied with the current elision algorithm. It is

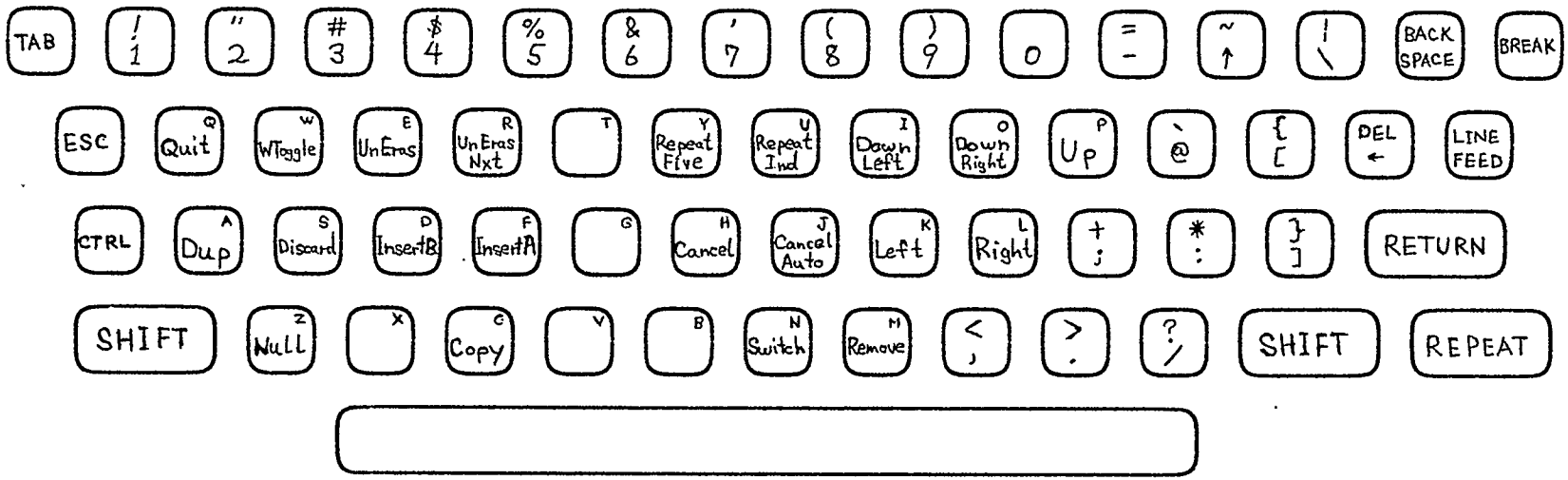


Figure 20. Layout of commands on the keyboard.

unstable in the sense that small changes in cursor position will often result in large changes in the display. Directly correcting this would require adding "interia" to the elision algorithm to allow it to select a suboptimal set of lines to display if this allows the display to remain the same or nearly the same as it currently is. Also, perhaps the unit of elision should not be a line but an entity more closely related to the original tree structure.

I now feel that it may be best to include in the language description a designation of certain productions, always including all the root productions, as display break productions. With Pascal, a program, procedures, and functions might be the display break productions. Elision would proceed as before, but the distance function would count crossing a display break as an especially large jump. Also, perhaps elision would be done not with reference to the cursor position but rather with respect to the closest display break ancestor of the cursor. The object of this is to produce a stable display by "chunking" the program and displaying primarily the chunk the cursor is in. Only cursor movements that cross display breaks would then cause large-scale changes in the display. Appropriate choice of display breaks would lead to the chunks being natural units.

The cursor of the first version of the editor was drawn as a simple box around the display of the subtree at which it was positioned. This was generally satisfactory, but was changed in the second version because I felt that the full box was somewhat obtrusive, distracting

one from viewing the program itself, and because I wished to encode more information in the cursor. The present two-corner cursor also eliminates the strangely-shaped boxes that resulted from subtrees whose display was split between lines. The prime motivation in encoding the information as to whether the cursor is at the first and/or last child was not so much to provide the user with this specific information but rather to ensure that some change in the display occurred as the result of certain commands. For example, if the cursor is positioned at a word of only one letter and the user moves the cursor down to that letter, the position of the cursor will not change but its shape will indicate that it is now positioned at an only child. The previous version of the cursor would not change at all. This is upsetting, as one is not sure that the command has actually been recieved by the editor.

The previous version of the editor displayed the name of the production class in effect at the position of the cursor in a corner of the screen. This was also largely intended to provide some feedback when the user moved the cursor in a way that wouldn't otherwise cause any change in the display. This was abandoned because the user will generally be looking at the position of the cursor, not the corner of the screen, so such a change will either not be noticed or will disrupt visual continuity.

The window scheme of the present editor is rather primitive. The second window is seen as a place for holding information temporarily in

order to allow one to perform operations requiring a rearranging of information. This is why there is a command for discarding the secondary window contents but not one for discarding the primary window contents. As soon as one considers the second window as being on a par with the first, one feels that third and fourth windows would be nice too. Splitting up the screen for these soon causes the windows to become impractically small. The solution would appear to be the adoption of overlapping windows as described in [Teitelman, in press].

Implementation Choices.

The present version of the editor suffers from a fairly small limit on tree size and a failure to quite meet the response-time goal of a tenth of a second in some situations. The description of Pascal given in Appendix B is close to the maximum size which can be handled by the language-description editor. Response can take up to half a second in certain situations when much of the display representation must be recomputed, or when many blocks must be allocated. Problems also arise when the user types quickly; even when the editor takes less than a tenth of a second for each keystroke, it may fall behind.

In retrospect, the decision not to employ a general storage allocation scheme capable of allocating and freeing variable-sized blocks of memory was a mistake. Doing so would save some space now wasted in fixed size blocks and would speed up the editor by making tree manipulations easier.

Still, if significantly larger trees are to be handled, the editor must either have more memory available or resort to keeping part of the tree on secondary storage. Using the current editor to edit large programs is quite practical if the program is separated into modules which are independently edited. A modest increase in available memory (not possible on the PDP 11 due to address-space limitations) would eliminate what problems there are with this approach. If large programs are to be edited in one piece, storing the entire tree in main memory is probably not practical. Parts of the tree that are distant from the cursor can be kept on disk as long as the display routine is able to realize that these distant parts are not needed in computing the display.

The decision whether to make common subtrees shared is difficult. Sharing subtrees has the direct effect of decreasing space requirements by a factor of about 1.25 (this varies quite a bit with the language). However, when the user duplicates the tree in order to perform a reshuffling operation, sharing gains one a factor of two. Thus if we wish always to allow the user this option, we reduce space requirements by a factor of about two and a half by sharing subtrees. On the other hand, modifying the tree is significantly more time consuming with shared trees, as nodes may not be actually changed. Macro expansion can benefit from sharing of subtrees by implementing automatic dynamic programming by saving the reduction of a subtree and reusing it without recomputation when the same subtree's reduction is required again.

Decisions on whether to share subtrees and on whether to keep parts of the tree on secondary storage also affect command cancellation. Not sharing subtrees would speed up the present command cancellation scheme in most cases, since fewer new nodes would have to be created, but it would greatly slow down some operations involving duplicating large subtrees since the previous contents of the space the new copies occupy would have to be saved. If subtrees are shared and parts of the tree not recently used are moved to secondary storage, a different cancellation scheme could be used, in which all the trees created during the editing session are kept, with older ones moving naturally to secondary storage.

The current implementation of the editor performs tree manipulations without regard for the affect they have on the display representation of the tree. This simplifies the tree manipulation routines and allows the standard display routine to be replaced by a specialized display module as was done with the document editor. However, recomputing the entire display after every command is too slow, hence the need for the display routine to remember previously computed parts of the display.

The reader may wonder why I did not implement the editor in Lisp, since it would appear to be the ideal language for the project. I sometimes wonder about this myself, but I judged that the time and space constraints were to stringent for Lisp, particularly since only interpretative Lisp implementations were available.

DISCUSSION OF THE TREE EDITING CONCEPT

In this section I discuss the basic tree editing concept and possible extensions to it. Of the more speculative possibilities mentioned, I would particularly like to have implemented some of the ideas I discuss in the section on how to check for context-sensitive syntax errors. The advantage of working with a tree rather than text should be substantial for this task; however, the problem is still difficult and time and available equipment did not allow for its solution.

Tree editors are not in common use. As far as I know, the only editors operating on the tree structure of programs with a substantial user community are those for Lisp (described below). The particular kind of tree editor I have designed is characterised by the complete absence of parsing - programs are entered as trees from the beginning. This idea was developed without knowledge of any closely related work, but it turns out to be not completely new. [Hoffmann, 1973] describes some related earlier work and a system of his own. Details are sketchy, but it appears that the system used menus for production selection, an option I rejected above. Hoffmann seems to feel that the editor can guide the user in creating a program that is not only syntactically correct but also semantically correct, provided the language designer makes the productions semantic options rather than purely syntactic options. I think the problem is not that simple; I'm

not sure I would like to be guided in that way.

Alternatives to Tree Editing.

This project originally had the rather vague aim of writing an editor for high-level language programs which would, in some sense, "know about the language in which the program is written". It was hoped, for instance, that syntax errors in the program being edited would be detected soon after they were committed. This section will examine several ways of accomplishing this objective.

A minimal approach to this problem is to augment an ordinary text editor with a command which asks the editor to parse the program and report on any errors found. This amounts to little more than a new way to invoke the compiler, although the convenience of, say, having the editor automatically move the line pointer of the text editor to the location of the first syntax error could be substantial. An extension of this concept would be to have the editor also automatically indent the program when requested. All manipulations of the program would still be performed in terms of its textual representation, however, and syntax errors would remain undetected until the user explicitly asked for his program to be checked.

A more ambitious editor might attempt to perform error checking and/or program formatting as the program is typed in or modified. A few text editing commands to perform operations particular to programs in the language might also be added, while keeping the general

text-editing orientation. Emacs [Greenberg, 1979] has some modes resembling this.

This approach has a number of problems. The look-ahead required for parsing results in program formatting and error checking lagging behind the text actually entered. Manipulations are performed in terms of the textual representation of the program and hence there may be times when the program is unavoidably incorrect syntactically, resulting in spurious error messages or bizarre formatting. Implementation effort would vary depending on ambitions, but could involve complex incremental compilation techniques if full error checking after each command with instantaneous response is required.

There are a number of editors for Lisp [McCarthy, 1962] programs (see, for example, [Sandewall, 1978] and [Neal, 1979b]) which typically feature automatic program formatting, identification of positions within the program in terms of the tree structure, and entry of data as text which is parsed to create the new subtree. Lisp is a natural language for a tree editor, since programs and data have a common representation as trees. Lisp has a very simple, non-redundant syntax which makes attempting to detect syntax errors immediately rather pointless - most errors result in quite correct, albeit unintended, programs. Immediate formatting of the program as it is entered would be useful, but is not usually attempted.

The Lisp editor approach can be applied to other programming languages, such as Pascal, but significant complications arise. The

tree structure of Pascal programs is much more complex than that of Lisp and this tree structure does not serve as the data language. The syntax of the textual representation of the trees is correspondingly complex, is more difficult to parse, and presents numerous possibilities for syntax errors which produce invalid programs. Immediate parsing, error checking, and formatting of text as it typed would be a complex task. Deferring these operations until a chunk of text has been completely entered is probably the most feasible approach. An editor for Pascal along these lines is described in [Douzeau-Gorge, et al, 1975].

One final approach was considered - this was to treat the program neither as text nor explicitly as a tree, but as an abstract data structure defined by the operations which may be performed on it. For example, there would be operations for adding and removing variable declarations, but the actual representation of the declaration section of the program would be of no concern to the user. The plan was to reinterpret ordinary phrases in the programming language as editing commands to perform these operations. Thus the phrase (in Pascal) "var I: char;" would be seen not as a part of the program, but as a command to add a new variable. Some new phrases (e. g. "novar I;") would also be needed. This plan was abandoned because it leads to a very complex editor which is specific to a single language and because it does not lend itself to an immediate feedback interaction style.

Advantages and Disadvantages of Tree Editing.

There is a gap between the textual input and display devices of computers and the tree structure of the information manipulated with them. Systems based on text editors keep the user as much as possible on the textual side of this gap, and bridge it by parsing. Tree editors bring the user across the gap and bridge it by command interpretation and "unparsing". The advantages and disadvantages of the two approaches derive directly from this basic difference.

First, use of a tree editor eliminates the possibility of syntax errors. More generally, a tree editor reduces the number of "unintentional" errors - those errors which arise from a mistake in the entry of the program rather than a mistake of logic. It remains possible that the user of the tree editor could, through a slip-up, create an unintended tree structure without noticing it. This is not too likely however.

The elimination of syntax errors _per se_ is perhaps less important than it might appear. Experienced programmers do not insert too many of these errors in their programs, and correction of them when they are detected by the compiler is generally straightforward. It may be that the elimination of errors which like syntax errors are "unintentional", but which happen to result in syntactically correct programs, is more important, since these errors can persist undetected (or at least unlocated).

Second, the user of a tree editor manipulates his data in terms of its tree structure rather than its textual representation. The extent to which this is an advantage rather than a disadvantage is determined by whether the user thinks in terms of trees or not. Clearly, a programmer must have some understanding of the underlying structure of his program if he is to function; the question is whether this information is easily usable during editing.

Third, the user need not bother with laying out his data in a nicely formatted way, since the editor creates the textual representation of the tree on its own. Some programmers like to have control over the formatting of their program; specially formatting certain statements to make them more readable. I feel that a point of diminishing returns is reached; that as long as the automatic formatter produces reasonable formatting in most cases, the fact that the reader can rely on absolutely consistent formatting more than compensates for the occasional case where the programmer could have formatted a statement more readably than the editor did. The advantage of automatic formatting is even more apparent when documents, rather than programs, are being edited.

Fourth, the user need not type in the characters that are displayed. In particular, the user can view a display with verbose keywords, redundant display cues, etc. but still type a short non-redundant form of the program. In this way the program can be made more readable without increasing the time required to enter it.

Fifth, since the information being edited is always stored in tree form, other programs dealing with the information (such as compilers) do not have to parse it. Of course, this advantage arises only if these programs were written with the tree editor in mind. The possibility of creating advanced program development systems capable of aiding the programmer in testing, verifying, debugging, documenting, optimizing, and other aspects of the program development process has been discussed by several authors (e. g. [Sandewall, 1978]). The user of such a system must understand the actions of the system; using trees as the communication medium can help ensure this.

Finally, the principal disadvantage of tree editors is the less direct correspondence between the display the user sees and the objects he manipulates. For example, when the user of a text editor sees that the cursor is in one place and he wishes it to be somewhere else, the direction the cursor must be moved is directly visible on the screen. With a tree editor, the direction of cursor movement will have no close relation to the appearance of the display.

Applications of Tree Editing.

The primary motivation for the tree editor was its application to the editing of computer programs, but the concept is potentially applicable to a wide variety of tasks in which structured information is manipulated by people with the aid of computers.

The editing of documents has received some attention in this

project. Documents have a natural tree structure and hence should be good candidates for a tree editor. One difference in the tree structure of documents as compared to that of a typical programming language is that the information in documents tends to be concentrated in the leaves of the tree. This makes a linear representation of the information somewhat more attractive than with programs. Another difference is the greater difficulty of hand-formatting a document, since changes tend to propagate over many lines. This argues in favor of a tree editor, but the same property also leads to large-scale changes to the display when small changes are made with the tree editor, a phenomenon which can be distracting.

These considerations make a decision on how to edit documents difficult. Considering the large size of the application area, it may make sense to construct a specialized editing system rather than use a general tree editor (or a slightly specialized tree editor as was done in this project). Still, the tree editing approach solves many problems which arise in attempting to design such a system. For example, a document editor should have commands for deleting letters, words, sentences, and paragraphs. The obvious approach leads to four commands. The tree editor provides a command framework in which these four operations are unified into a single command.

A third area touched on in this project was the use of the tree editor to prepare input to computer programs which require structured information as input. Language descriptions given to the language

description compiler are an example of this. Input languages of this kind are fairly common; examples are input to a compiler-compiler, data-base descriptions for a data-base management system, and file dependency descriptions for a program management system. One advantage of Lisp for applications such as theorem-proving is that the programmer need not parse an input language since the input may be represented as S-expressions parsed by the Lisp system; this is a considerable convenience, especially in an experimental system in which the input language is changing. The tree editor could provide similar advantages.

An area not developed in this project is the representation of information in a data-base by trees and the entry and modification of such information using the tree editor. For example, a language description for a record of employee personal information containing fields for name, address, etc. could easily be written, and the tree editor then used for display, entry, and modification of the record. How this would fit in with the rest of payroll system is a matter for future research.

System command languages provide commands for deleting, renaming, and otherwise manipulating files, for running programs, and for other tasks performed in the course of an interactive session. The tree editor could be used to enter these commands, but a more interesting approach would be to use the tree editor to edit file directories, and thus eliminate the need for the file manipulation commands (see

[Fraser, 1980] for a somewhat similar idea). The ultimate expression of this approach would be to regard all the information in the machine as one big tree, manipulated by the tree editor; what are now seen as file directories would simply be the higher levels of the tree.

Finally, one area not appropriate for the tree editor in its present form is the editing of two or three dimensional information. Although a two-dimensional array of objects could be represented by a tree consisting of a series of rows with each row consisting of a series of columns, the natural transition from a row and column to the same column in the row above does not correspond to a natural transition in the tree. Graphical objects may exhibit a tree structure at a higher level, however.

Further Development of the Tree Editor.

The editor as implemented illustrates the essential features of the tree editing approach. Several more-or-less obvious enhancements would be desirable in a production system.

First, the macro expansion language should be integrated into the editor to allow the user to perform large-scale transformations, such as changing the name of a variable in his program, correcting a spelling mistake in a document, or removing all third parameters of the call of a particular procedure. I have implemented this extension in a version of the editor, but have not bothered to provide for the user aborting the transformation while it is in progress and restoring the

state before the transformation was begun. This facility would be necessary in a production system since some transformations are non-terminating.

Secondly, it would be desirable for the editor to read the language description itself rather than having the language description compiled by a second program. This is an essentially minor change which was not made because it increases the size of the editor (and memory space is precious in this implementation).

Once the second change has been made, it may be desirable to allow the user to change the language description in the middle of editing a program in the language. This would allow the user to very conveniently extend the language to fit the requirements of one particular program. Some problems arise in keeping track of all the changes and in merging extensions to the language made at different times.

More powerful facilities for handling multiple trees or multiple versions of trees would be desirable. The current one-or-two tree editor allows all operations (splitting a program up, merging programs, rearrainging a program) to be performed, but some operations (e. g. merging three programs) are clumsy.

Many improvements are possible in how the tree is displayed by the editor. The algorithm for splitting long lines could be improved. For the document editor, allowing for hyphenation of words would be

desirable. Perhaps most importantly, the elision algorithm could be much improved. With all these improvements, it must be remembered that not much time is available for computing the display.

Finally, the biggest improvement that could be made would be to implement the editor on an appropriate machine. Single-character at a time interaction with instant response is an essential requirement of a tree editor of this type. A high-resolution screen which can be instantly updated is almost as important. Neither of these needs are likely to be satisfied by a multi-user time-sharing system. The single-user system for which this implementation was primarily intended suffers from serious deficiencies also, notably a lack of memory and a display which can't be refreshed fast enough to avoid flicker.

Handling Context-sensitive Syntax.

It would be nice if the editor knew something about the context-sensitive syntax of the language being edited and could inform the user of such errors as undeclared identifiers, wrong numbers of parameters for procedure calls, or type clashes.

One way of so informing the user would be to signal an error whenever he tried to, say, enter an undeclared indentifier (except, of course, in a declaration). There are potential problems with this approach. The user may know that the identifier is undeclared and be intending to enter its declaration later. The spelling of the identifier may not match that in its declaration and it may be the

declaration that is wrong. An override would probably be necessary to allow the user to enter the incorrect identifier anyway, but the existence of this error might confuse further error checking.

An alternative would be to check for errors only when explicitly requested. The editor could move the cursor to the location of an error automatically. Of course, determining the real location of an error is in general impossible.

More sophisticated things might be attempted beyond simple error-checking. The editor might automatically fill in parts of the program which are uniquely determined by context-sensitive constraints. For example, if the user types 'a' as the first letter of an identifier and the only valid identifier beginning with 'a' is "aardvark", the editor could fill in the rest of the identifier automatically. Similarly, the editor could create the proper number of holes for parameters of a procedure call as soon as the name of the procedure was known. Warnings of ridiculous contructs could also be issued in addition to actual errors.

Scope rules could be incorporated into a facility for making global changes. The user could then ask for the name of a variable to be changed without variables in other scopes with the same name being changed also. To implement this feature in full generality is more difficult than it appears. Consider, for instance, the problem of changing the name of a field of a Pascal record, taking full cognizance of the possible existence of other records with fields of the same

name, of with statements, and of inner blocks in which names are masked by new declarations.

Implementation of these features may be divided into two parts - specifying the context-sensitive syntax for a language and using this specification in the editor. Attribute grammars [Knuth, 1968] seem to provide a appropriate framework for the first task. An attribute grammar associates certain values with each point in the tree; these values are computed from zero or more values at neighboring points in the tree. The computations must be non-circular, of course. Certain restrictions may have to be imposed on the ways attributes can be computed. One possibility is to restrict the computations to allow them to be computed in a single traversal of the tree. Requiring that all values be computed without reference to values higher in the tree is another possibility. Neither of these restrictions impair the theoretical completeness of the method but may lead to practical difficulties in writing the language descriptions. Note that the actual computations used may be expressed in whatever language is convenient.

These attributes could indicate to the editor that an error is present at a particular place in the tree or that a certain option should be automatically filled in by the editor. Naturally, those features which require the editor to check syntax on every keystroke require very fast computation of attributes. Storing the attributes of all subtrees would help (if the attributes are not invalidated by the

keystroke) but would require a lot of space.

Possible Human-factors Improvements.

This section describes several possible improvements to the editor which might improve the ease and pleasure with which it can be used but which do not affect the over-all design. In one sense these could be considered "minor" changes, but they can nevertheless have a major influence on the quality of the editor. Of the two terminals the editor currently is set up to use, the VT11 is far preferable in theory as it has a larger screen and the ability to draw the cursor properly. In practice, I seldom use that terminal with the tree editor because the flickering of the screen and the high noise level make it more pleasant to use the editor with less adequate terminals in a better environment. As these problems are not unique to a tree editor, I will say no more about them, but they may be more important than the items mentioned below.

One problem with the tree editor is a shortage of characters for commands. There are two contexts where this is a problem - choosing characters to represent editing commands and choosing characters for production names in a language description. The shortage is not really absolute but is rather a shortage of easily remembered and easily typed characters. One possibility is to add further shift keys in addition to the ordinary shift and the control key. Just adding a second control key on the other side of the keyboard would help. Adding extra

labels to the keys to show where the editing commands are located would also be helpful. An alternative to adding more shift keys would be to add a new block of keys used only for entering editing commands, but this requires the user to move a hand away from the main keyboard which disrupts the normal typing process. Using another input device in addition to the keyboard causes similar disruptions. A possibility worthy of investigation is the use of a one-handed keyboard to free a hand for using a light-pen or specialized keyboard. One-handed keyboards using a "chord" method exist, but I don't know much about them.

Currently, the tree editor makes all changes to the display as fast as it can (instantly with the VT11). This is not necessarily desirable. The instant insertion of lines in the display is sometimes confusing as the user must hunt for what he used to be looking at. The alternative is to smoothly create a opening for the lines to be added by moving the adjoining lines apart. If the time for this to occur is not to become annoying, it should take less than a second (a quarter of a second seems about right, but different times would have to be tried out). Since the object is to prevent disorienting changes, this motion should be as smooth as possible; this imposes some constraints on the refresh time of the screen which may be more severe than those required to prevent flicker. No experiments along these lines were performed because the VT11's phosphors are too persistent, causing the objects to appear blurred when one moves them.

A related problem is encountered with the document editor. When a word is inserted in the middle of a paragraph, the rest of the paragraph may have to be rearranged to accomodate the change. This occurs fairly frequently when inserting text in the middle of a document and can be disconcerting. Smoothly moving the text, as described above, may help. Another possibility is to reduce the frequency of these rearrangements by making the right margin "fuzzy", i. e. not requiring that exactly as many words as possible be on each line.

Implications for Language Design.

Considering the great difference between the way programs are entered with the tree editor and with a text editor, it is not surprising that the two methods lead to different preferences in language design.

One problem with writing language descriptions for languages not designed with the tree editor in mind is the selection of appropriate production names. For instance, the description of Pascal given in appendix B uses the first letter of the primary keyword as the production name for statement types. A problem arises with the "while" and the "with" statements, resolved by arbitrarily assigning a production name of "T" to the "with" statement. This discordance between input and display languages could likely have been avoided if the language had been designed for the tree editor. A similar, but

more serious, problem arises with C [Kernighan & Ritchie, 1978], which has a plethora of operators whose lexical representations are optimized for parsing, not for use as production names.

Ridiculous language design decisions become obvious when one attempts to describe the resulting tree structure for the tree editor. The occurrence of colons in the parameters of Pascal's "write" and "writeln" standard procedures is one instance of this. Pascal's pretence that its standard procedures are just like any other procedures, everywhere a bit thin, becomes clearly absurd in this case, as colons are not allowed in the parameter lists of any other procedures. The solution adopted in appendix B, considering colon to be just another operator, is such a blatant fudge that any self-respecting language designer would surely rethink things at this point. In general, one would expect cleaner abstract syntaxes to result from the use of tree editors, because the tree structure is no longer hidden from the user.

The syntax of a language should make plain the underlying tree structure (this is a good idea even if a tree editor isn't being used). Pascal is quite good in this regard; the various components of a block are all set off by keywords. In contrast, Algol's syntax places visually similar declarations and statements next to each other without any separator.

Infix operators are a problem. Their natural ambiguity has lead in the past to baroque precedence schemes and other strange conventions to

disambiguate them. With the tree editor, they are not very natural, since the user must enter the operator before the operands in any case. Despite their long use in mathematics and programming languages, it is tempting to get rid of them in favor of prefix operators. Perhaps their use results from the infix position of the English verb; if so, it is worthwhile to note that there are plenty of human languages with prefix or postfix verb position.

A language that is parsed must be completely unambiguous. No such constraint exists if programs in the language are entered solely with the tree editor. Of course, if the display of the program is highly ambiguous, the user will be unable to determine what the tree structure of the program is and will not be able to use the editor effectively. Occasional ambiguities that are easily disambiguated by context are not necessarily ruled out, however. Naturally, there are no positive advantages to ambiguous languages, but a less cluttered syntax may result from using one. The language description language described in appendix A omits the numerous quotation marks or escape characters which would be needed to produce a completely unambiguous display language. The result is quite useable but can occasionally be confusing.

The macro expansion language associated with the tree editor has effects on programming language design. True syntactic extensibility, while not unknown (see, for instance, [Irons, 1970]), is not present in any commonly used programming language. As a result, many constructs

are included in programming languages which are not fundamental and might more logically be considered extensions. An example from Pascal is the "for" statement. A particularly striking example is the "activate" statement of Simula [Dahl, Myhrhaug & Nygaard, 1970], which is defined by an ordinary procedure call but cast in the form of a statement for syntactic convenience. Easy extensibility allows a language to be tailored to the needs of a particular application area, a particular programmer, or even a particular program.

Significance of the Macro Expansion Language.

The pattern-match and replacement facility described earlier as a macro expansion language has full computing power and thus merits consideration as a programming language in its own right. Figure 21 shows a simple language description for strings of characters with two extra productions that play the role of functions to reverse a string and to append a character to the end of a string. Figure 22 shows the reductions that expand these "functions" into their results. The style used here is very Lisp-like. Whether this programming style is the best for this language or whether I have just not really learned how to program in this language is an open question.

The language is inherently applicative; more so than pure Lisp. Pure Lisp was easily augmented by procedural constructs and functions, but it is difficult to see how to add, say, a "print" construct to the macro expansion language. However, the language could be used to

```
<string>
  " | "{<character>}"
  R | reverse<string>
  A | append <character> to<string>
  # | #
```

Figure 21. Description of strings and string functions.

```
find string
  reverse
    ""
reduce to
  ""

find string
  reverse
    "<character.a>{character.b}"
reduce to
  append <character.a> to
    reverse
      "{character.b}"

find string
  append <character.a> to
    "{character.b}"
reduce to
  "{character.b}<character.a>"
```

Figure 22. Definition of string functions in the macro language.

implement non-applicative programs via a state transition method in which a new state and output is computed from an old state and input by purely applicative means. Many interactive programs (including this implementation of the tree editor) have this structure in any case.

An outstanding characteristic of the language is its simplicity. That such a simple language can easily express transformations which would be very difficult to express with a textual macro language is one illustration of the advantages to be gained from treating programs as trees instead of text. For general programming, the language has available a rich tree structure for representing data and these trees can be immediately manipulated without the need for defining special procedures for doing so. Consider the case of a compiler from Pascal to assembly language. The input and output languages can be easily represented as trees as could the symbol table and any intermediate languages used in the translation process. Not only should the compiler be easier to write than in a conventional language, it should also be easier to understand, since the effect of a segment of the program on the data structures is exactly what is pictured by the pattern and its replacement. I should note, however, that I haven't actually written such a compiler.

Of particular interest would be the writing of all or part of the tree editor itself in the macro expansion language. The computation of the display would be an especially appropriate section to rewrite if speed constraints can be met. This would allow special display modules

to be written with much less effort than at present.

Although it seems a shame to clutter such a simple language, there are some reasons for contemplating extensions to the macro language. The class of patterns is not extensible in the present scheme; it would be nice, for instance, to provide not only for matching a subtree in which two components are equal but also for matching when two components are equivalent in some user-defined sense. Of course, some extensions could be made by applying the language's macro capabilities to itself.

The current implementation of the language is not very fast. The fundamental problem is that finding a subtree and a pattern matching it may involve many unsuccessful comparisons. Since any match may be found (i. e. the process need not be deterministic), the problem seems amenable to the use of multiple processors. One could organize this in several ways - a processor might handle a particular part of the tree or it might handle a particular reduction or the processors might randomly search for matches. This is an interesting problem for future research.

CONCLUSION

The current state of the tree editor project is that an experimental implementation of the tree editor incorporating the basic concepts has been written, along with an interpreter for the macro expansion language. I can state that the tree editing concept is quite viable. I am most experienced with the document editor, and, aside from the defects noted before in the implementation and in the terminals employed, I found the experience positive. However, the present implementation is not of production quality, partly because the equipment used is rather inappropriate. Consequently, it has not been possible to test the value of the editor by judging the reaction of a user community or by conducting experimental comparisons with text editors.

The next logical step would be to produce a production-quality implementation of the editor, with a few relatively minor improvements, on a personal computer with a good display (60 lines by 80 columns, say) and an adequate amount of memory (128KB, or perhaps a bit more). This would allow more experience to be accumulated and provide a sound base for attempting the more ambitious extensions.

REFERENCES

DAHL, O-J., MYHRHAUG, B. & NYGAARD, K. (1970). [Simula] Common Base Language. Oslo: Norwegian Computing Center.

DOUZEAU-GORGE, V., HUET, G., KAHN, G., LANG, B., & LEVY, J. (1975). "A structure-oriented program editor: A first step towards computer assisted programming", International Computer Symposium 1975. Amsterdam: North Holland Publishing Company.

GREENBERG, B. (1979). An Introduction to Using Multics Emacs On-line documentation on the University of Calgary Multics system.

FRASER, C. (1980). "A generalized text editor", Communications of the ACM. vol. 23 no. 3 (March 1980) pp. 154-158.

HOFFMAN, H-J. (1973). "Programming by selection", International Computer Symposium 1973. Amsterdam: North Holland Publishing Company.

IRONS, E. (1970). "Experience with an extensible language", Communications of the ACM. vol. 13 no. 1 (January 1970).

JENSEN, K. & WIRTH, N. (1975). Pascal User Manual and Report. New York: Springer-Verlag.

KERNIGHAN, B. & RITCHIE, D. (1978). The C Programming Language. Englewood Cliffs, New Jersey: Prentice-Hall.

KNUTH, D. (1968). "Semantics of context-free languages", Mathematical Systems Theory. vol. 2 no. 2 (June 1968) pp. 127-145.

MCCARTHY, J., et al. (1962). LISP 1.5 Programmer's Manual. Cambridge, Mass.: MIT Press.

NEAL, R. (1979a). A Tree Editor. unpublished.

- (1979b). A Brief Geli Manual. On-line documentation on the University of Calgary Department of Computer Science Unix system.

NELSON, T. (1974). Computer Lib / Dream Machines. Published by the author.

RITCHIE D. & THOMPSON, K. (1974). "The UNIX Time-Sharing System", Communications of the ACM, vol. 17 no. 7 (July 1974).

SANDEWALL, E. (1978). "Programming in an interactive environment", Computing Surveys, vol. 10 no. 1 (March 1978) pp. 35-71.

TEITELMAN, W. (in press). "A display oriented programmer's assistant", International Journal of Man-Machine Studies.

WINOGRAD, T. (1979). "Beyond programming languages", Communications of the ACM, vol.22 no. 7 (July 1979) pp. 391-401.

APPENDIX A - DESCRIPTION OF THE LANGUAGE-DESCRIPTION LANGUAGE

Beware of confusing the occurances of '(', '|', etc. as characters with their occurances as meta-characters.

```
<language-description>
  # | (<class-modification> )

<class-modification>
  # | <class>(<modification>)

<modification>
  # | <string> <type> {<element>}
  /e | eliminate (<string>, )
  /i | include <class> {<string>, }

<type>
  include <character> 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, -, |, *

<element>
  include <character>
  /[ * [{<del>}<class>]
  /{ * {{<del>}<class>{<del>}}
  /( * (<class>{<del>})
  /< * <class>
  /| * |
  /! * !(<digit>)!

<del>
  include <character>
  include <element> /|, /!

<digit>
  include <character> 0, 1, 2, 3, 4, 5, 6, 7, 8, 9

<string>
  # * (<character>)

<class>
  # * <(<letter>)>

<letter>
  include <character> a, b, c, d, e, f, g, h, i, j, k, l, m,
   n, o, p, q, r, s, t, u, v, w, x, y, z, -

<modification>
  /a | assume <string> before {<character>}
```

```
<element>
  ?[ * [?{<del>}<class>]
  ?{ * {?{<del>}<class>{<del>}}
```

APPENDIX B - DESCRIPTION OF PASCAL

This description is of full standard Pascal as described in [Jensen & Wirth, 1975], except that the name of the case selection field of a variant record may not be omitted (for minor technical reasons). There are also no semicolons and no period at the end of the program; these serve no useful purpose during editing and cause some problems in aesthetically formatting the program. A language description describing the same tree structure but with semicolons in the places required by standard Pascal is available for preparing programs for a compiler.

```
<program>
  # | program <name><program-parameters>|[ <block-comment>]
   [ <label-section>][ <const-section>][ <type-section>]
   [ <var-section>]{ <procedure-definition> } <body>

<program-parameters>
  # - ((<name>, ))

<body>
  # | begin|{<statement>}end

<block-comment>
  # | { (<word> )|}

<word>
  # * (<non-space-character>)

<non-space-character>
  include <character>
  eliminate

<label-section>
  # | label (<label>, )

<const-section>
  # | const(<const-definition>)

<const-definition>
  # | <name> = <constant>[!27!<side-comment>]
```

```
<type-section>
  # | type(<type-definition>)

<type-definition>
  # | <name> = <type>[!27!<side-comment>]

<var-section>
  # | var(<var-declaration>)

<var-declaration>
  # | (<name>, ) : <type>[!27!<side-comment>]

<side-comment>
  # - { (<word> ) }

<procedure-definition>
  # | #
  P | procedure <name>[<formal-parameters>]|[ <block-comment>]
   [ <label-section>][ <const-section>][ <type-section>]
   [ <var-section>]{ <procedure-definition> } <body>
  F | function <name>[<formal-parameters>] : <name>|
   [ <block-comment>][ <label-section>][ <const-section>]
   [ <type-section>][ <var-section>]{ <procedure-definition> }
    <body>

<formal-parameters>
  # - ((<parameter-group>; ))

<parameter-group>
  # - (<name>, ) : <name>
  V - var (<name>, ) : <name>
  P - procedure (<name>, )
  F - function (<name>, ) : <name>

<name>
  # * <letter>{<letter-or-digit>}

<letter>
  include <character> a, b, c, d, e, f, g, h, i, j, k, l, m,
   n, o, p, q, r, s, t, u, v, w, x, y, z

<digit>
  include <character> 0, 1, 2, 3, 4, 5, 6, 7, 8, 9

<letter-or-digit>
  include <letter>
  include <digit>

<statement>
  # | #
```

```
  = | <variable> := <expression>
  : | <label>:<statement>
  ( | <name>[<actual-parameters>]
  W | while <expression> do<statement>
  T | with (<variable>, ) do<statement>
  I | if <expression> then<statement>[else<statement>]
  R | repeat{<statement>}until <expression>
  F | for <name> := <expression> <direction> <expression> do
   <statement>
  C | case <expression> of(<case-list>)end
  G | goto <label>
  B | begin|{<statement>}end
  assume = before abcdefghijklmnopqrstuvwxyz
  assume : before 0123456789

<case-list>
  # | (<constant-range>, ) :<statement>

<constant-range>
  include <constant>
  . - <constant>..<constant>

<actual-parameters>
  # - ((<expression>,))

<direction>
  T - to
  D - downto

<label>
  # * (<digit>)

<variable>
  # - <name>{?<modifier>}

<modifier>
  . - .<name>
  ^ - ^
  [ - [(<expression>,)]

<constant-base>
  L * nil
  " * "(<character>)"
  X * (<digit>)[?.<fraction>][?E<exponent>]

<fraction>
  . * (<digit>)

<exponent>
  E - [<sign>](<digit>)
```

```
<sign>
  include <character> +, -

<constant-intermediate>
  include <constant-base>
  include <name>
  assume X before 0123456789

<constant>
  include <constant-intermediate>
  assume X before 0123456789
  - - -<constant-intermediate>
  + - +<constant-intermediate>

<expression>
  include <constant-base>
  include <variable>
  assume X before 0123456789
  : 0 <expression>:<expression>[:<expression>]
  [ - [{<expression-range>,}]
  + 4 [<expression>]+<expression>
  - 4 [<expression>]-<expression>
  O 4 <expression> or <expression>
  * 6 <expression>*<expression>
  / 6 <expression>/<expression>
  M 6 <expression> mod <expression>
  D 6 <expression> div <expression>
  A 6 <expression> and <expression>
  N 8 not <expression>
  >> 2 <expression>><expression>
  << 2 <expression><<expression>
  >= 2 <expression>>=<expression>
  <= 2 <expression><=<expression>
  <> 2 <expression><><expression>
  = 2 <expression>=<expression>
  ( - <name><actual-parameters>

<expression-range>
  include <expression>
  . - <expression>..<expression>
  assume X before 0123456789

<scalar-type>
  include <name>
  . - <constant>..<constant>
  ( - ((<name>, ))

<structured-type>
  F - file of <type>
  S - set of <scalar-type>
```

```
  A - array [(<scalar-type>,)] of <type>
  R | record{<field-definition>}[?<record-case>]end

<type>
  include <structured-type>
  include <scalar-type>
  ^ - ^<name>
  P - packed <structured-type>

<field-definition>
  # | (<name>, ) : <type>[!27!<side-comment>]

<record-case>
  # | case <name> : <name> of(<record-case-list>)

<record-case-list>
  # | (<constant-range>, ) : ({<field-definition>}
   [?<record-case>])

<constant-range>
  include <constant>
  . - <constant>..<constant>
  assume X before 0123456789
```

APPENDIX C - DESCRIPTION OF THE DOCUMENT LANGUAGE

The description of the document language given below adequately describes only the tree structure and the input aspects of the language. the display aspects of the description are suggestive only, the actual specialized display routine ignores that part of the description. In particular, words which are shown here as being enclosed in accents (`) are instead underlined when printed. Also, the text is single-spaced when displayed by the editor and double-spaced when printed.

```
<section>
  /s | <sentence>{| <section> }
  /p |     (<sentence> )
  /l | (<line>)
  assume /p before 1234567890-^\!"#$%&'()0= |qwertyuiop@[_QWER
   TYUIOP`{asdfghjkl;:]ASDFGHJKL+*}zxcvbnm,.ZXCVBNM<>?

<line>
  # | (<character>)

<sentence>
  # - (<part>)

<word>
  # - (<letter>)

<part>
  include <word>
  include <character> ., ,, :, ;, !, ?
  " - "(<part>)"
  ' - '(<part>)'
  ( - ((<part>))
  ` - `<word>`

<letter>
  include <character>
  eliminate  , ., ,, :, ;, !, ?
```

APPENDIX D - SUMMARY OF EDITING COMMANDS

The summary below lists all commands of the editor, classified by function, with the command's name, the control character used to invoke it, and a brief description of what it does.

Cursor movement:

Up	^P	Move the cursor to the parent of the present node
DownLeft	^I	Move the cursor to the leftmost child
DownRight	^O	Move the cursor to the rightmost child
Left	^K	Move the cursor to the next sibling to the left
Right	^L	Move the cursor to the next sibling to the right

Window manipulation:

Dup	^A	Duplicate the tree, creating the secondary window
Discard	^S	Discard the secondary window
Switch	^N	Switch to the other window's cursor
WToggle	^W	Toggle whether both windows are displayed at once

Tree modification:

InsertA	^F	Insert a hole after the present node
InsertB	^D	Insert a hole before the present node
Remove	^M	Remove the present node
Copy	^C	Copy from the other window's tree
UnErase	^E	Replace with the last erased node
UnErasNext	^R	Replace with the next child of the last erased object

any printing character is used as a production name (or part thereof)

Meta-commands:

Quit	^Q	Terminate the editor, writing the window(s) to file(s)
Null	^Z	Do nothing, but do the automatic follow-up actions
Cancel	^H	Cancel the last command
CancelAuto	^J	Cancel the automatic follow-up of the last command
RepeatInd	^U	Repeat the last command indefinitely
RepeatFive	^Y	Repeat the last command five times