

Copyright is owned by the Author of the thesis. Permission is given for a copy to be downloaded by an individual for the purpose of research and private study only. The thesis may not be reproduced elsewhere without the permission of the Author.

GED

A GENERALISED SYNTAX EDITOR

A Thesis Presented in Partial Fulfilment of the Requirements

for the Degree of Master of Science in Computer Science

at Massey University

Giovanni Serafino Moretti

1984

MASSEY UNIVERSITY



1061308248

## ABSTRACT

This thesis traces the development of a full-screen syntax-directed editor - a type of editor that operates on a program in terms of its syntactic tree structure instead of its sequential character representation.

The editor is table-driven, reading as input an extended BNF syntax of the target language. It can therefore be used for any language whose syntax can be defined in EBNF. Print formatting information can be included with the syntactic definition to enable programs to be pretty-printed when they are displayed.

The user is presented with a pretty-printed skeletal outline of a program with the currently selected construct highlighted and all required syntactic items provided by the editor. Any constructs with alternatives, such as "<statement>", which occurs in many languages, are initially denoted by a placeholder in the form of a non-terminal name (i.e. "<statement>") which is expanded when the user indicates which alternative is wanted. All symbols entered by the user are parsed immediately and any erroneous symbols rejected, making it impossible to create a syntactically incorrect program. The editor cannot detect semantic errors as no semantic information is available from the EBNF syntax. However the first use of all identifiers is flagged by the editor as an aid to the detection of undeclared identifiers.

A "help" area at the bottom of the screen continuously displays a list of the correct next symbols and the syntactic definition of the currently selected program construct. This display, together with a multi-level "undo" command and the provision of a skeletal program by the editor, provides a way of exploring the various constructs in a programming language, while ensuring the syntactic correctness of the resultant program.

## Table of Contents

1	Program Preparation - The Traditional Approach.....	1
1.1	Integrated Programming Environments.....	3
1.2	Interpretive BASIC Systems.....	3
1.3	Keyword Entry.....	4
1.4	Syntax-Directed Editing Environments.....	5
1.5	Cornell Program Synthesiser.....	7
1.6	ALOE - A Language Oriented Editor.....	16
1.7	Editor Allan POE - A Pascal Oriented Editor.....	28
1.8	COPAS - A Conversational Pascal System.....	34
1.9	"Z" - The 95% Program Editor.....	40
1.10	Summary.....	42
2	GED - Giovanni's Editor.....	44
2.1	Language Input Definition.....	45
2.2	The User's View.....	50
2.3	The Display.....	51
2.4	Inserting User Input.....	53
2.5	Displaying Optional and List Placeholders.....	55
2.6	Cursor Movement Commands.....	56
2.7	Marking and Returning to Marked Nodes.....	62
2.8	The Delete Command - F5.....	64
2.9	The Insert Command - F6.....	67
2.10	Reading and Writing Files.....	67
2.11	Undo Function - F12.....	69
2.12	A Command Summary in Function Key Order.....	70
2.13	Summary.....	72
3	GED - Its Internal Architecture.....	73
3.1	The Input Language Syntax.....	73
3.2	Definition of the Extended BNF Accepted by GED.....	77
3.3	Requirements of the Internal Syntactic Representation.....	85
3.4	Representating Tokens of the Meta and User Languages.....	87
3.5	Describing the Names Of Productions.....	90
3.6	Non-terminal Syntax Nodes.....	92
3.7	Concatenation and Alternation of Productions.....	93
3.8	The Data Structure used to Represent Optional Symbols.....	98
3.9	The Data Structure used to represent the List Construct.....	99
3.10	Storing a Representation of the User's Program.....	105
3.11	Recording the State of a Parser Without a Stack.....	106
3.12	The Initial Form of the Program Node Tree.....	108
3.13	The Program Node Field Definitions.....	112
3.14	Automatic Inclusion of Necessary Terminal Symbols.....	116
3.15	The Cursor - the Concept of a "Current Node".....	119
3.16	Where does the Cursor Stop?.....	119
3.17	The Inclusion of User Symbols into the Program Tree.....	123
3.18	The Structure Created by the Expansion of Loop Nodes.....	130
3.19	Unparsing - Deriving a Display from the Program Tree.....	132



3.20	Defining Layout - A Table-Driven Pretty Printer.....	134
3.21	GED Print Formatting Commands.....	137
3.22	Associating Formatting Commands with the Syntax.....	138
3.23	Generating the Screen Display.....	143
3.24	Optimising the Rewriting of the Screen Display.....	144
3.25	The Implementation of User Commands.....	146
3.26	Primary Cursor Movement Commands.....	146
3.27	Reading and Writing the Program and Clipped Subtrees.....	147
3.28	The Clip/Delete and Insert Commands.....	151
3.29	Marking, and Moving to, Specific Nodes in the Program.....	154
3.30	The Implementation of the "Undo" Command.....	155
4	The Implementation of Syntax-Editors for New Languages.....	156
4.1	Preparing the Extended BNF Grammar.....	157
4.2	A Case Study - The Implementation of a Snobol Editor.....	159
4.3	Areas of Alteration in the Snobol Grammar.....	165
4.4	Are Identifiers, Numbers, Strings and Comments.....	166
4.5	Hiding Optional Placeholders.....	167
4.6	Removing the Production for <BLANKS> from the Snobol.....	170
4.7	Rewriting the Productions to Remove Common Start Symbols.....	170
4.8	Defining the Print Formatting Commands.....	173
4.9	The Implementation of Pascal and Lisp Editors.....	174
4.10	Problems Encountered in the Addition of Formatting.....	178
4.11	Summary.....	180
5	Conclusions.....	182
5.1	A Short Description of the System.....	182
5.2	The Realisation of Design Goals.....	182
5.3	Generality of the Editor.....	183
5.4	Ease of Setting-up.....	184
5.5	Ease of Use.....	185
5.6	Future Developments.....	186
5.7	Final Thought.....	187
	Acknowledgements.....	188
	Bibliography.....	189

## Chapter 1

### Introduction

#### 1 Program Preparation - The Traditional Approach

The most common method of program preparation involves the repeated use of a text-editor and a compiler. This method has an inherent limitation - even if the user is sitting at a terminal, it enforces an essentially batch mode of operation. The programs are prepared, and then submitted to a compiler for verification and translation. There are two error classes that could be eliminated if the editor itself was cognizant of the syntax of the programming language in use. The first class is composed of errors that violate the lexical grammar of the language and the second of errors in the constructive syntax - the productions that define how the lexical symbols may be combined.

#### Lexical Limitations

A text editor accepts programs, as an arbitrary sequence of characters, whereas logically a program is a sequence of unique symbols. Some of these symbols are required by the syntax, others occur in syntactically-ordered pairs or groups and some may be chosen by the programmer.

The only items in a program whose textual nature is significant are identifiers, numbers, strings and comments. These are composite items consisting of sequences of characters, and the fact that reserved words

are externally represented as sequences of characters is irrelevant and in this context misleading. It is irrelevant because although reserved words look like identifiers, they are treated in the syntax as unique symbols - a single incorrect character destroys the validity of a reserved word, whereas even several altered characters may leave a symbol still conforming to the syntax of an identifier.

More importantly, in this context it is misleading to treat reserved words as character sequences as it leads the user to think of a program as being composed of characters, not symbols. A text editor, having no knowledge of program syntax, manipulates the program as text, reinforcing this view.

### Structural Limitations

A text editor has no knowledge of the syntactic structure of a program. Therefore common errors such as unbalanced bracketing symbols and the omission of required symbols are not recognised at a stage where it is possible to correct them easily. Only later, during the compilation of the program, will these errors be detected, and then immediate correction will be impossible.

If the editor knew the target language syntax then these syntactic errors could either be detected immediately and corrected, or prevented.

### 1.1 Integrated Programming Environments

The integration referred to here is that of the editor and the program that actually translates the user's program, be it compiler or interpreter. The most common such translators are interactive systems for the language BASIC but languages with dynamic data structures like APL, LISP and SNOBOL are also usually interpreted and often interactive.

Traditional interactive systems were in general originally designed for use with printing terminals and have had a line-oriented syntax - the slow speed of such terminals made the interactive editing of multi-line syntactic items impractical. Examples of this approach are interactive versions of BASIC, LISP, APL and the JOSS system although the most common by far is BASIC. For a language with an appropriate syntax, line oriented program entry is easy to use on both fast and slow speed terminals as the incremental parsing alerts the user to errors in a line as soon as that line is entered.

### 1.2 Interpretive BASIC Systems

The BASIC language was developed for teaching and was specifically designed to be interactive. The reasons for this are threefold:

- (a) The input is checked for errors at the end of each line and erroneous lines may be corrected immediately.
- (b) An altered program is immediately executable without the need to invoke a compiler or leave the BASIC system.

- (c) A line trace is available during execution and it is possible interactively to find and alter the values of all variables for debugging purposes.

This first two of these are the most important, as having a single environment in which to create, edit and execute programs is an important contributor to BASIC's ease of learning and use. As the system can be left in "BASIC Mode", beginners do not need to learn about the operating system and editor environments.

### 1.3 Keyword Entry

A letter from Mr G.J. Tee of the Auckland University Computer Science Department contains a reference to what must be one of the earliest systems for the entry of complete keywords in a single keystroke: "I visited the Computer Centre at the University of Moscow during the International Congress of Mathematicians, in about June 1966. I saw there card punches being used to prepare ALGOL source programs, with the key-board including keys for the reserved words in ALGOL. For instance, one key had the Russian equivalents of BEGIN and END as the lower-case and upper-case symbols" [Tee 1983]. More recently the Sinclair ZX81 and the Spectrum microcomputers have their BASIC interpreters and keyboards arranged so that any keyword can be obtained by depressing (possibly in conjunction with a shift key) an appropriately labelled single key [Vickers 1980,1982]. This helps to avoid spelling errors and to ease program entry. The use of keyword entry reduces the program entry time simply by reducing the number of characters that need to be typed - this is especially valuable for

beginner who are often unfamiliar with a keyboard - and thereby reduces the opportunity for error. The editing of existing lines of program is also symbol oriented, with keywords being skipped, added and deleted as single entities. The systems are interpretive and check the syntax on a line-by-line basis which also contributes to their ease of use. This single keystroke token entry is the first form of syntax-directed program entry to be widely available.

#### 1.4 Syntax-Directed Editing Environments

In the BASIC systems discussed in the previous section, the user is constrained by the syntax of language being entered and it is impossible to construct erroneous program units larger than a single line without the generation of an error message.

A contrasting technique made possible by the widespread availability of high-speed terminals has been the development of full-screen editors that provide an window into a file, instead of a view based on lines. Such editors may provide commands for editing the file in textual constructs - word processors deal with letters, words, lines, sentences, paragraphs and pages - or alternatively provide an editing environment in which the editing units are not textual but syntactic. Given the high speed at which the screen may be redrawn, the syntactic constructs need not be line-oriented and can therefore extend over several lines.

Syntax-directed editors permit the user to create programs that conform to the syntax of the programming language in use. The BASIC systems previously discussed are line-oriented examples of syntax-directed editing environments. More recently, syntax-directed editors for languages with a nested syntactic constructs have been developed.

These include the Cornell Program Synthesiser for PL/C (a subset of PL/1) [Teitelbaum 1981], the ALOE syntax-editor generator [Medina-Mora 1981], the POE editor for PASCAL [Fischer 1981] and the COPAS system for Pascal [Atkinson 1981]. The Z editor [Wood 1981] is a text editor but has features relating to program structure normally found only in true syntax-directed editors.

Each of these editors will be discussed to illustrate the user's view of the editor and the commands available. Where relevant the internal structure is also discussed.

### 1.5 Cornell Program Synthesiser

The stated goals for the program synthesiser [Teitelbaum 1981] were to provide ".... a unified programming environment that stimulates program conception at a high level of abstraction, promotes programming by step-wise refinement, spares the user from mundane and frustrating syntactic details while editing programs, and provides extensive diagnostic facilities during program execution." The synthesiser is designed on the premise that programs are not text but hierarchical structures, and should be constructed and manipulated as such. The language implemented is PL/C, an instructional subset of PL/1.

The Cornell Synthesiser was first used on PDP-11s under Unix and later on TERAk microcomputers. The microcomputer implementation has been used for teaching introductory computing students and has received most use on relatively small programs.

#### The User's View

The user is presented with a skeleton of a program into which new statements and expressions may be incorporated. This approach automatically enforces a top-down view of a program. The synthesiser is designed to be used with a high speed video terminal and provides the user with a window into the program in its current state of refinement. After the user has indicated which file is being created and that the "main" procedure is to be edited, the display has the form shown in fig 1.1.



```
/* comment */  
abs: PROCEDURE OPTIONS (MAIN);  
    {declaration}  
    {statement}  
END abs;
```

Fig 1.1 - Initial display of PL/C main procedure.

Notice that the program even in its initial state is a correct sentential form (i.e. structurally correct). This feature is common to most syntax-directed editors. The lowercase words on the display indicate where the user may insert extra constructs into the program. These words are called "placeholders." The replacement of placeholders is the only way in which the user can alter the form of the program. This implies a hierarchical structure as the replacement of one placeholder may itself contain other placeholders.

The cursor is denoted by underlining in these examples and indicates which placeholder is currently selected for refinement. Braces are used to indicate iteration and square brackets are used to indicate optional terminal symbols or productions, according to the conventions of Extended BNF as defined by Pagan [1981].

#### Insertion of User Input

There are two methods of program entry. The first is to request the inclusion of a "template" - a predefined compound syntactic structure such as a complete IF statement. The second is to enter a "phrase" - a method of entering text not constrained by the language syntax.

### Templates

A template is requested by typing its name and then pressing a special function key. Examples of names are ".i" for an IF statement, ".pl" for a PUT LIST statement and ".dw" for a DO - WHILE statement. When a template is requested, the editor checks to ensure that the structure is valid at the current cursor location. If so, then the structure is included in program and the display is altered to reflect the change. An erroneous request is detected immediately and the command rejected. Consequently while the program may be incomplete, it is always structurally correct. If the cursor in fig 1.1 was on the placeholder for "statement" and the user requested the IF template the display would become:

```
/* comment */
abs: PROCEDURE OPTIONS (MAIN);
  {declarations}
  IF ( condition )
    THEN statement
    ELSE statement
  END abs;
```

Fig 1.2 - After Requesting an IF Template

Notice that it is not necessary to fill in the placeholders in order. Both "comment" and "declarations" may be expanded later.

The cursor is positioned at the first placeholder within the new template. In this example the placeholder "condition" does not have any associated templates. All expressions (including "condition") are entered purely as text. An expression is therefore one example of a "phrase."

### Phrases

The user's view of an expression doesn't usually correspond to the internal parse tree and therefore the manipulation of expressions as syntactic entities can be awkward. To avoid this, assignment statements and expressions are entered and edited as text, and then parsed. Phrase editing appears to the user just like full-screen text editing. Directing the cursor away from the phrase invokes the parser and the user is notified of any errors. Errors in phrases are permitted and the user may ignore them, but an erroneous phrase will remain highlighted until it is either made correct - by correcting the phrase or declaring any undeclared variables - or deleted. In the case of undeclared variables, the highlighting would disappear as soon as the variable was declared.

Comments are treated as phrases and can therefore be entered and manipulated as text in the usual fashion.

### Moving the Cursor

The commands for moving the cursor reflect the underlying syntactic structure of the program. The cursor may be moved to placeholders, to phrases, and to the first symbol in a template. This means that the cursor may be moved to the IF, the "condition" or either of the "statement" placeholders in fig 1.2 but not to any symbols entered by the editor itself such as the THEN or ELSE. There is no way to alter the IF statement template, it can only be expanded or deleted. The movement commands are given in table 1.

up/down	Move to previous/next template, phrase or placeholder.
left/right	Like up/down but also stops at every character within a phrase.
RETURN	Move to next template, phrase, placeholder or optional placeholder within lists.
long up/down	Move to previous/next template, phrase or placeholder not at a structurally deeper nesting level.
long RETURN	Like RETURN but not at a greater nesting level.
\,/	Move to previous/next immediately enclosing program element. Eg a "\" would move from inside an IF statement to the IF.

Table 1 - Cornell Synthesiser Cursor Movement Commands

Note - The "long" command is a single key on the TERAK microcomputers and is used as a prefix to the main command.

#### Optional Placeholders

There are many options and optional items that are possible during the entry of a program. To display all of these is confusing and would quickly clutter the screen. To display optional components in lists of elements, such as the possibility of a statement between two others in a list of statements, the RETURN key is used. To display the optional part of a placeholder, such as the possible label on every statement, the ".o" command is used.

Moving Sections of Program

Templates and phrases may be clipped from a program and inserted elsewhere. When either is clipped the original placeholder will reappear. The cursor can then be moved and the clipped section inserted elsewhere. The commands are as follows:

.clip	Move template or phrase to the file CLIPPED
.delete	Move template or phrase to the file DELETED
.mv "Filename"	Move template or phrase to the file "Filename"
.insert	Insert CLIPPED at current cursor location
.ins "Filename"	Insert "Filename" at current cursor location

Table 2 - Synthesiser Program Modification Commands

Comments

The insertion of comments is restricted to three places; after variable declarations, the "comment" field of a procedure template, and the "comment" field of a comment template.

The comment template is a compound item, a combination of a comment and a subordinate list of statements. This unusual structure is used to provide elision, a feature whereby the statements themselves are not displayed, just the comment. This is used to hide irrelevant detail when displaying program structure by enabling more of it to fit on the screen at once. For example, suppose the program outline in fig 1.2 had been expanded to:

```
/* comment */
abs: PROCEDURE OPTIONS (MAIN);
  {declaration}
  IF ( condition )
    THEN
      /* exchange x and y */
      temp = x;
      x = y;
      y = x;
    ELSE statement
  END abs;
```

Fig 1.3 - IF Statement Before Elision

The statements "temp=x; x = y; y = x;" are subordinate to the comment "/\* exchange x and y \*/." Typing the command "<...>" would cause the statements to disappear and be replaced by "...". (fig 1.4). The statements themselves are not deleted, just not displayed. This permits the detailed functions of a program to be suppressed to display the overall structure. Typing <...> again would cause the statements to reappear.

```
/* comment */  
  
abs: PROCEDURE OPTIONS (MAIN);  
  {declaration}  
  IF ( condition )  
    THEN  
      /* exchange x and y */  
      ...  
    ELSE statement  
  END abs;
```

Fig 1.4 - IF Statement after Elision

(The cursor is on the first of the three dots)

### Execution Capabilities

During the construction of a program, code is generated for each template and a program may be executed, even if it is incomplete. If an unexpanded template is encountered, execution is suspended. The template may at that stage be refined and execution continued. During execution the display can be divided into three sections; one to display the output of the program, one to display the program source code being executed and a third to display the current values of any desired scalar variables.

As programs would normally run too quickly for the display to be of any use, execution may be slowed or single stepped.

If execution has changed a variable before the user has stopped the display, execution may be run in reverse for a limited number of program steps.

The synthesiser is a functioning syntax-directed editor with a powerful execution and debugging facilities for PL/C. It has been used successfully to teach programming to large numbers of students.



### 1.6 ALOE - A Language Oriented Editor

The ALOE language oriented editor generator is part of the GANDALF project at Carnegie-Mellon University. The ALOE (A Language Oriented Editor) System is unusual in that it is a syntax-directed editor generator. It has been used to build editors for numerous languages, the more well-known ones being C [Kernigan 1978], PASCAL [Jensen 1974] and ADA [Ada 1980]. Developing an ALOE editor for a new language involves generating a description of that language in accordance with the grammar for ALOE descriptions. Since this grammar may be defined syntactically, another ALOE editor tailored for its own input syntax, is used instead of a text editor to prepare descriptions. When seen in terms of the GANDALF project whose aims are the construction of many System Development Environments, large programs, and many programmers, the reason for this generality is evident [Habermann 1982].

The ALOE is described in its user manual as:

" ... a tool which supports the construction and manipulation of tree structures while guaranteeing their syntactic correctness" [Medina-Mora 1981].

The program is represented inside the ALOE as an abstract syntax tree which is manipulated directly by the user. It is important to note that a syntax tree is distinct from a parse tree. In a parse tree the nodes are operators whereas in a syntax tree they are the non-terminals of the language. This distinction is important because a syntax tree more closely resembles the user's view of a program than does a parse tree.

### The User's View

The screen is initially divided into two windows, but this (like all attributes of the system) is user definable. These windows display the program itself and a one line status display. Errors, requests for help, and displays of clipped subtrees all cause extra windows to be overlaid on top of the current display.

### The Cursor

The cursor is a highlighted region, as distinct from the point cursor used in the Cornell Synthesiser. This is to give a clear indication of the extent of the subtree covered by the cursor whereas a point-cursor would be ambiguous. Cursor movement is not defined in terms of the textual display but is described as part of the unparsing scheme - the definition of how the internal syntax trees should be displayed. Unparsing schemes and their uses will be described later.

### Constructive Commands

The tree created by the ALOE will have some nodes that cannot be expanded without more input from the user. These nodes correspond to the Cornell Synthesiser's placeholders - in this context these nodes are known as "meta-nodes." Whenever the current node is a meta-node it is possible for the user to generate a subtree by entering the name of the operator or its synonym. The cursor will be placed at the first meta-node within the subtree if there is one or at the next meta-node if there isn't. It is possible to cause terminal symbols to appear automatically in newly generated subtrees.

### Moving the Cursor

The following editing commands are common to all ALOE editors:

- `._IN` Move the cursor to the first meta-node within the current subtree.
- `._OUT` Move the cursor to the parent of the current node.
- `._NEXT` Move to the next meta-node at the same level. If none exist then move to the next meta-node at the same nesting level as the parent node. This continues recursively until either a new meta-node is found or the remainder of the tree has been searched. If no meta-nodes remain the the cursor stays at its current node.
- `._PREVIOUS` Move to the previous meta-node in the same manner as `._NEXT`
- `._HOME` Move to the root node of the current window. If the current node is already the root node then move to the root node of previous window.
- `.BACK` Swap the cursor's current position with its last position.
- `.FIND "what"` Search the current window for an occurrence of "what". The last string given is used again if none is supplied. ".GLOBALFIND" is used in the same manner to search all windows.

With the exception of .BACK and .FIND, any of the above can be prefixed with a repetition count.

### Help Facilities

".HELP" will display either a list of language commands and their synonyms (if the current node is a meta-node) or a list of editing commands.

### Tree Manipulation Commands

- .CLIP treename To clip the current tree into a named subtree.
- .INSERT treename Insert the named tree at the current position. For an insertion to be correct the current node must be a meta-node and the subtree must be a valid expansion of it.
- .DELETE Delete the current subtree. If the current node is an element of a list then replace it with its meta-node otherwise delete it.
- .REPLACE This is the same as DELETE except that if the current node is an element of a list, it is deleted but a meta-node left in its place. This meta-node will become the new current node.
- .NEST <operator> Clip the current subtree and nest it in a subtree of root node "operator." Although no example is given, from the written description this command appears to act in the following way: If the current subtree was <statement> then the command ".NEST IF" would clip

the current <statement>, insert an IF 'statement and then search for the first occurrence of <statement> inside the IF statement and insert the clipped subtree there.

**.TRANSFORM name** Change the operator of the current node to "name."  
This will work only if the tree definitions are identical.

#### List manipulation commands

Four commands exist to extend a list in both the forward and reverse directions (.APPEND & .PREPEND) and to include new meta-nodes inside a list (.EXTEND & .BEXTEND - Extend Backwards).

#### Text Editing

".EDIT" is a command to invoke EMACS, an extensible screen editor [Stallman 1981] to edit constants or text nodes. When the user returns to the ALOE the screen will show the updated text.

#### Checkpointing

The ALOE will write out a checkpoint file after a set number of tree modifying commands. The number is usually thirty but can be altered during the definition of the ALOE.

#### Action Routines

Action routines are optional but can be included to be called by the editor in various situations. These routines can perform such actions

as semantic checking, emitting code or manipulating the syntax tree itself.

### Unparsing Schemes

The display format for these trees is defined in one or possibly many "unparsing" schemes. The unparsing scheme is used to define how the internal syntax tree is to be displayed. The unparsing scheme is defined in terms of print formatting commands, examples of which are; increasing and decreasing the current indentation level, returning to the left margin, and skipping to a new line. This means that the display format may change depending on which unparsing scheme is in force at the time. This can be used to provide different display formats depending on either tree-depth or position of the tree relative to the cursor. Both elision and altered formats are possible. Figs 1.5, 1.6 and 1.7 illustrate the reformatting and elision that is possible by altering the unparsing scheme.

```

PROGRAM program_name;
VAR f, found : boolean; ch : char;
BEGIN
  found := false;
  REPEAT
    IF condition THEN
      f := found
    ELSE f:= not found;
    writeln(f);
    read(ch);
  UNTIL ch = 'Z';
  writeln(f);
END;

```

Fig 1.5 - ALOE Display - Cursor on the IF Statement

can be displayed as above if the cursor is on the IF statement, or as :

```

PROGRAM program_name;
VAR f, found : boolean; ch : char;
BEGIN
  found:= false;
  REPEAT
    IF condition THEN f := found ELSE f:= not found;
    writeln(f);
    read(ch);
  UNTIL ch = 'Z';
  writeln(f);
END;

```

Fig 1.6 - IF statement no longer under cursor so reformatted

or if the cursor is moved further down the program, as:

```
PROGRAM program_name;
VAR f, found : boolean; ch : char;
BEGIN
  found:= false;
  REPEAT
    <statements>
  UNTIL ch = 'Z';
  writeln(f);
END;
```

Fig 1.7 - As cursor moves away - IF statement is Elided

The unparsing scheme can be used to alter the display format, for example reformatting the THEN and ELSE parts to show the whole IF statement on one line (fig 1.6), or to hide subtrees to provide elision (fig 1.7). The unparsing scheme can be altered dynamically, either by the user to cater for different layout preferences, or automatically to provide elision.

#### Extended Commands

It is possible to cause the editor to execute routines which manipulate the tree and/or start up other UNIX processes. These user-written routines can call a set of library routines to access and manipulate the syntax tree. These library routines are provided so the editor can retain control of modifications made to the tree, in order to guarantee its correctness. The user routines can be written in any language that is load-compatible under UNIX.



ALOE Input Grammar

The example language has two statement types - PRINT and FOR. This illustrates the form of input grammar required by an ALOE editor.

Language Name: INTERP

Root Operator: PROGRAM

```
{      /* terminal operators */

LOOPVAR =      {v}          - It's a variable
              | (0) "@s"    - Unparse scheme - print name
              | action <none>
              | synonym: ", " ;

INT         =      {c}          - It's a constant
              | (0) "@c"    - Print its constant value
              | action: aINT  - Name of procedure to call
              | synonym: "#" ;

EMPTYSTEP  =      {s}          -
              | (0) "1"
              | action: <none>
              | synonym: <none> ;

}

{      /* non-terminal operators */

PROGRAM    =      stmts
              | (0) "@1"    - Start in Column 1
```

```

| action: <none>
| synonym: <none>
| precedence: <none>
| Filenode;           - Subtree stored in a file
PRINT  =  <exp>
| (0) "print @0"      - PRINT follow by <exp>
| action: <none>
| synonym: <none>
| precedence: <none>
| Non-filenode;

FOR    =  loopvar exp exp stepexp stmts
| (0) "for @1 = @2 to @3 step @4@+@n@5@-"
| (1) "for (@1 = @2; %1 <= @3; %1 += @4)@+@n@5@-"
| action: <none>
| synonym: <none>
| precedence: <none>
| Non-filenode;

PLUS   =  exp exp
| (0) "@1 + @2"
| action: <none>
| synonym: "+"
| precedence : 1
| Non-filenode;

STMTS  =  <stmt>
| (0) "@0@n"         - print expansion of stmt,
| action: <none>     and skip to new line
| synonym: <none>

```

```

    | precedence: <none>
    | Non-filenode;
}

```

The "@" followed by a number refers to a particular item in the definition list. In the definition of the FOR statement for example @1 refers to the "loopvar" and @5 refers to "stmts". The other symbols preceded by "@" or "%" define various actions to control the display formatting.

The FOR statement has two unparsing schemes defined. This means that the print formats can be either:

```

Either for i = 4 to 8 step 2   OR   for (i = 4; i <= 8; i +=2)
      print (i + 3) * i, i           print (i + 3) * i, i

```

### Meta-node Classes

```

{      /*  ~ Classes  */
stmts      = STMTS ;
exp        = INT LOOPVAR PLUS ;
loopvar    = LOOPVAR ;
stepexp    = INT PLUS EMPTYSTEP ;
stmt       = PRINT FOR ;
}

```

The classes define the valid expansions for a meta-node. For example, either an integer, a loop variable or a PLUS node (which will itself have expressions as its leaves) is a valid subtree for the "exp" meta-node. The ALOE system is designed as a general purpose syntax-directed editing system. To generate a new ALOE, the language grammar is defined and translated into tables and then linked to any action routines needed and any other environment-specific routines. Details of an ALOE editor for a simple language and its action routines are described in detail in "ALOE Users' and Implementors' Guide" [Medina-Mora 1981].

### 1.7 Editor Allan POE - A Pascal Oriented Editor

POE [Fischer 1981] is more similar to the Cornell Synthesiser than to ALOE, previously described. It is specifically designed for Pascal although versions for other languages are envisaged. The commands for cursor movement, and the display format are similar to those in the synthesiser, but the method of insertion is by entering the required symbols, not by command. The program is automatically pretty-printed and checked for structural correctness.

#### The User's View

This initial display of a program is shown in fig 1.8. Although it is not explicitly stated in the reference, the cursor appears to be a point cursor, not a highlighted region.

```
PROGRAM <ID> ( <FILE ID LIST> ) ;
{LABELS}
{CONSTANTS}
{TYPES}
{VARS}
{PROCEDURES}
BEGIN
  {STMT LIST}
END .
```

Fig 1.8 - Initial Display of Editor Allan POE

#### Optional and Required Placeholders

The symbols in fig 1.8 surrounded by "<" and ">" are placeholders whose expansion is required before the program is complete. Placeholders surrounded by "{" and "}" are optional. Notice that this use of braces in the syntactic meta-notation is different from that used in the synthesiser, where braces are used to indicate iteration.

### Insertion of User Text

In order to insert symbols the user moves the cursor to the required placeholder and then types the actual Pascal or the start symbol of the production. Two examples given in the reference are entering "VAR i:integer" to obtain a variable declaration from {VARIABLES} and IF to obtain a complete IF statement template. Whether the editor provides the colon and prompts for <TYPE> (in the VAR example) is not described.

The template provided for the IF statement is shown in fig 1.9. POE like the other editors mentioned guarantees structural correctness. However, if the user enters a symbol that is erroneous in the current position, POE, unlike the Synthesiser and an ALOE, attempts to fit the symbol into its most logical position. For example, entering THEN at a statement prompt will also cause the insertion of an IF template. Incorrect replacements can simply be deleted. The display after the replacement of {STMT} with the IF template is shown in fig 1.9.

```
PROGRAM <ID> ( <FILE ID LIST> ) ;
BEGIN
    IF <EXPR>
    THEN {STMT}
    {ELSE CLAUSE} ;
    {MORE STMTS}
END .
```

Fig 1.9 - After Replacement of {STMT} with IF Template

Notice that the optional placeholders have disappeared. They are displayed only if they occur after the cursor. Once the cursor moves past the optional prompts, they are suppressed and not redisplayed unless specifically requested, wherever the cursor is moved.

**Cursor Movement**

To aid portability, the arrow and function keys found on many terminals are not used. Instead the exclamation mark "!" is used to indicate a command following.

The commands relating to cursor movement are shown in table 3.

Space bar	Move cursor one symbol right.
Back space	Move cursor one symbol left.
Return	Move to leftmost symbol on the next line.
!b	Back one screen
!f	Forward one screen
!d	Down half a screen
!g	Top of program
!G	Bottom of program
!t	Top of screen
!B	Bottom of screen

Table 3 - POE Cursor Movement Commands

Unlike both the synthesiser and the ALOE, no commands are provided for moving in syntactic increments larger than one symbol.

### Deletions

The DELETE key is used to delete the smallest syntactic unit containing the current symbol. Successive DELETES will delete successively larger sections of the program - the most nested being the first to be deleted. This can be thought of as replacing templates with their placeholders (instead of the other way around). This corresponds to ascending the tree representation of the program deleting expansions of



non-terminal derivations. This form of deletion is not designed for replacement, the clipped subtree is no longer accessible. To enable the user to recover from commands with unexpected results, an "undo" command is provided.

#### The Undo Command

To recover from editor command errors, the user can enter "!u". This will undo the effect of the last command. Multiple undo commands are also handled. The effect is to undo the most recent commands excluding the undo commands themselves. To actually undo an undo, the "!U" command can be used.

#### Copying and Replacement Commands

Structures that would have been deleted if the DELETE key had been used, can instead be moved to named subtrees. These subtrees can be edited if necessary and inserted at other points in the program. Only a syntactically valid subtree may be inserted.

#### Prompting Commands

Although the editor prompts the user with a name relating to the symbol expected as a replacement for a placeholder (e.g. STMT for statements) at times this level of prompting will be insufficient. A command is provided ("!p") to display the options, one at a time. For example, if the user requests help on the possible expansions of {STMT}, the first option displayed will be "{STMT} --> nothing", then "{STMT} --> {LABEL} {UNLABELLED STMT}". Only one option is displayed at a time and the

list rolls around, reverting to the first option if the list runs out. If the user enters "!" the currently selected option becomes the replacement for the placeholder.

### Elision

Subtrees can be elided only by the specific command "!>". To revert to the unelided form, the complementary command "!<" must be given. No automatic elision or tagging of comment fields is supported.

### Execution Capabilities

The POE system can also execute programs, but unlike the synthesiser it will do only so if they are complete (no remaining placeholders) and are semantically correct. During execution, program input is taken from the keyboard and output is displayed on the screen.

### 1.8 COPAS - A Conversational Pascal System

The COPAS system [Atkinson and North 1981] is an interactive Pascal program development system developed at the University of Sheffield. It more closely resembles an amalgamation of an editor and a compiler than the systems previously described.

#### Acceptance and Execution Modes

The COPAS system has an "Acceptance Mode" and an "Execution Mode." The distinction between the acceptance mode employed here and the methods of program construction previously described is marked. During program entry under the COPAS system, the user is effectively using a conventional text editor. There are no constraints imposed by the editor relating to the Pascal syntax. Each line is verified as it is entered. If an error is made the user can only modify lines prior to and including the line in error. When the program is complete the user is notified. The program may then be executed or extra program lines added.

The User's View

The version of COPAS described is intended for use on a printing terminal and so the editing commands illustrated relate directly to a line-oriented text editor.

The initial command is "Accept program-name" and the system responds with the first line of a Pascal program numbered as line 0 (fig 1.10).

```
0 PROGRAM demo (input, output);
10
```

Fig 1.10 - Initial View of Program under COPAS

and the user must enter the remainder of the program. The input is buffered into lines and errors may be corrected using the BACKSPACE key in the usual way.

With the exception of ACCEPT, all the editing commands available to the user could be from a conventional line-oriented text editor. They are listed in table 4.

ACCEPT program-name	Provide a standard program heading line.
BREAK line-nos	Split lines
CHANGE line-nos	The indicated line is to have characters inserted, deleted.
DELETE line-nos	Delete a line
MOVE from to	Move a set of lines.
PRINT line-nos	Print the indicated section of the program

REPLACE line-nos	Same as DELETE followed by INSERT but TRACE status (see later) is maintained.
ACCEPT data	Accept data without providing a line number.

Table 4 - COPAS Editing Commands

The "line-nos" may be either a single line number, a range of lines or a set of lines or ranges.

After the first line printed by the system in response to the "Accept demo" command, further lines may be entered by the user. The system is already expecting text so an "INSERT" command is unnecessary.

If the user entered the lines shown in fig 1.11, and the END was misspelt as "ENF" the system would respond with:

Note: All user input is underlined.

```
0 PROGRAM demo (input, output);
10 begin
20 write('Hi There');
30 enf
30 enf
   ^
END or ; expected
Now what?
```

Fig 1.11 - Initial Entry of a program under COPAS

The user can now correct line 30 and the program would be immediately accepted without a request from the user. It wouldn't be executed but the user would be notified that it had been accepted.

If line 20 is replaced with the "REPLACE" command to become:

```
20 writeln('The date is the', date);
```

```
Identifier not declared
```

```
Now what? I 25 { Insert line 25 }
```

```
You cannot edit beyond line 20
```

### Correcting Errors

Errors found during the parse may be modified by the user by editing the program text in the manner of a conventional text-oriented editor. There is however one constraint - no text after the first error may be edited, only preceding text.

Collecting all the COPAS examples given so far into a sample terminal session will indicate how the system is used.

Note : All user input is underlined.

ACCEPT demo

```

0 PROGRAM demo (input, output);
10 begin
20 write('Hi There');
30 enf
30 enf

```

END or ; expected

Now what? REPLACE 20 {Replace line 20}

```

20 writeln('The date is the', date);

```

Identifier not declared

Now what? INSERT 25 { Insert line 25 }

You cannot edit beyond line 20

Now what? INSERT 5 { Insert line 5}

```

5 var date : integer;

```

Program accepted

Now what ? PRINT {Print complete program}

```

0 PROGRAM demo (input, output);
5 var date : integer;
10 BEGIN
20 writeln('The date is the', date);
30 END.

```

Now what?

Fig 1.12 - A Sample COPAS Terminal Session

### Execution Capabilities

COPAS can only execute complete programs. If a run-time error occurs, the user can request a display of all currently visible (i.e. in scope) scalar variables including parameters. If the error was the attempted use of an undefined scalar variable then the user can provide a value and request that execution be continued. All other errors cause execution to be abandoned. The TRACE command is provided to enable the user to find the values of variables while a program is executing. It will set a trace flag on a line or set of lines. During execution, if COPAS encounters a line with its trace flag set, it displays the line number and the values of any variables changed by the execution of that line. If no argument is given for the trace command, all lines are traced.

### Internal Representation

The text is converted into tokens and then stored as a linked list of lines. The complete program is recompiled each time acceptance is attempted. If the compilation is error-free then an interpreter executes an intermediate code representation generated by the compiler. This method of operation would be too slow for large programs but the system was intended for student programs (which are usually small) and its speed has proven satisfactory.



### 1.9 "Z" - The 95% Program Editor

Z is a full screen text editor which although it has no knowledge of program syntax, can pretty-print programs, skip complete syntactic structures and provide elision of nested syntactic constructs [Wood 1981]. I have included it in this survey to illustrate the diversity of approaches taken to provide editing based on a program's structure.

#### The User's View

The user impression of the text as manipulated by the editor is of a window into a plane of text that can extend infinitely in both the horizontal and vertical directions.

Although there are many commands in Z for textual manipulation and word processing, those of interest in this context are those concerned with the manipulation of and movement by syntactic entities. They include automatic indentation, balancing of matched pairs of tokens (such as parentheses), movement in syntactic increments, and elision.

The authors have augmented the editor with a table that describes the tokens of the language. This table indicates which tokens should cause tabbing and backtabbing and also which tokens occur in pairs, two examples being "begin - end" and parentheses. All the language dependent capabilities are based on the information contained in this table - no knowledge of the syntax is available.

### The Many Uses of Indentation

Using the list of tab and backtab tokens, Z can pretty-print the program. Once the program is in this format the provision of skipping over syntactic units of the program becomes straightforward. The editor can move in complete syntactic units using the same visual cues as the programmer - the indentation level.

### Elision

Eliding sections of program text is done in the same manner. The "ZOOM" command has one operand which indicates the maximum level of indentation to be displayed. A zoom level of zero displays only the top level lines - the procedure headings and declarations - and a zoom level of infinity displays the whole program. This provides elision related to the nesting level of structures but cannot provide elision related to the position of a structure relative to the program cursor.

### Balanced Expressions

Using the list of which tokens open and close balanced expressions, the editor can move over balanced expressions and structures as single units. The editor can also provide the matching right bracketing symbol for the most recent unbalanced construct. If there isn't a current unbalanced construct the editor will indicate the position of the most recent balanced construct.

This system currently includes tables for LISP, BLISS, PASCAL, RATFOR (rational FORTRAN) and APL.

### 1.10 Summary

The syntax-directed editors in this chapter illustrate a wide variety of approaches to incorporating knowledge of a programming language syntax into an editor.

With the exception of COPAS, all of the editors display the program in a pretty-printed form which is immediately updated whenever the program is modified. Features available only on video terminals such as highlighting sections of the program provide a view of the program unattainable on slow or printing terminals.

Whether or not the versions of COPAS intended for video instead of printing terminals follow this approach is unclear.

There is more diversity in the types of user commands than in the display formats. Commands for easily moving around and manipulating the displayed program are crucial, especially if the editor inserts templates for complete constructs in their syntactically correct place rather than at the current cursor position. This can cause the cursor to jump ahead an unexpected amount and insert an unexpected construct. The user must be able to revert to the previous state without undue difficulty.

The use of an editor for more than one language is approached only by the ALOE system, but the input form of grammar it uses is completely different from the more conventional forms of a syntax definition. This precludes its use without learning a new form of grammar specification and rewriting the grammar for the new language.

This thesis explores the development of a syntax-directed editor that has as its input the language specification in extended BNF notation. The language syntax is not written into the editor but is read in as data at the start of an editing session. To enable the editor to pretty-print the program, print formatting information is read in as well. From the information contained in the syntax, the editor provides a program outline, complete with all the required terminal symbols. Placeholders are left for non-terminal derivations that require further information from the user.

The notation used for the language description is powerful and easy to use. This makes the generation of a syntax-directed editor for a new language straightforward. The editor has so far been used to construct programs in PASCAL, LISP and SNOBOL. To add languages it is necessary only to define their grammar in extended BNF.

## Chapter 2

### GED - A Purely Syntax Directed Editor

#### 2 GED - Giovanni's Editor

This thesis describes the development of a syntax-directed editor in which the syntax is not implicitly built into the editor, but is defined in a standard machine-readable notation. Because of this feature, the editor may be initialised with the syntax for any language and will thereupon become a syntax-directed editor for that language. The language syntax is input in Extended BNF and may be augmented with pretty-printing (program formatting) information if the final program layout is important. The programmer, on starting to use the system is presented with a skeletal outline of the program in the appropriate language and this can be modified by expanding the placeholders provided by the editor. The syntactic production represented by the current placeholder, and the set of next symbols that would be correct at the cursor position are also displayed. This provides the user with a simple way of exploring the constructs of the language.

The use of a standard notation (Extended BNF) for the language definition precludes the detection of semantic errors, as a one level syntax definition does not include the necessary information for this. Therefore type mismatches and undeclared identifiers (in languages that treat these as errors) will not be detected. However, the occurrence of undeclared or incorrect identifiers due to omission of a declaration

or misspelling is a common error. As an aid to their detection, the editor provides an indication every time a new identifier symbol is used.

The use of a two-level grammar [van Wijngaarten, 1969] would enable these errors to be detected, but two-level definitions are very complex and not widely understood. Consequently, although their use would render a syntax-directed editor very powerful, it would place a pragmatic restriction on its general applicability.

The examples that follow use the language Pascal. This is for consistency and is not meant to imply that the editor is tailored to Pascal. An example using Lisp will be included later.

### 2.1 Language Input Definition

The editor builds an internal data structure representation of the grammar from the input Extended BNF version. The structure mimics the form of each definition and the definition can therefore be regenerated from it. A simplified grammar for Pascal in the editor input format is shown in fig 2.1 and its corresponding data structure shown in fig 2.2.

The first four lines define the structure of identifiers by enumerating the set of characters that may start an identifier and those that may occur after the first character. The grammar in fig 2.1 will allow the use of "\_" (as in "first\_node") within a Pascal identifier but not as its start character. This scheme will also allow initial characters that are not permitted within an identifier body. An example of this

is the use of "&" as a reserved word flag in SNOBOL - The "&" may be used only as the initial character (eg "&ANCHOR").

IDENTIFIER\_START\_SET

abcdefghijklmnopqrstuvwxyABCDEFGHIJKLMNOPQRSTUVWXYZ

IDENTIFIER\_BODY\_SET

abcdefghijklmnopqrstuvwxyABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789\_

<program> ::= PROGRAM <program\_name> [<list\_of\_files>] ; <block> . \$

<program\_name> ::= IDENTIFIER \$

<list\_of\_files> ::= <file\_name> { , <file\_name> } \$

<file\_name> ::= IDENTIFIER \$

<block> ::= BEGIN <statements> END \$

<statements> ::= ..... Defn of a list of statements and do on.. \$

\$\$

Fig 2.1 - Editor Input Grammar for Pascal Subset

The layout is free format. Each definition is terminated by a "\$" character and the complete grammar by two "\$" characters.

This is to allow the skipping of erroneous definitions at the grammar input stage. The error recovery while reading the syntax is limited to indicating which symbol was encountered and the symbol actually received. Then the remainder of the syntactic part of the definition is then skipped. If a print format part of the definition is present (detailed in chapter 3), is parsed separately. The error handlers for both the syntactic and formatting parts attempt to leave any erroneous definitions in such a state that their use will not cause the editor to fail.



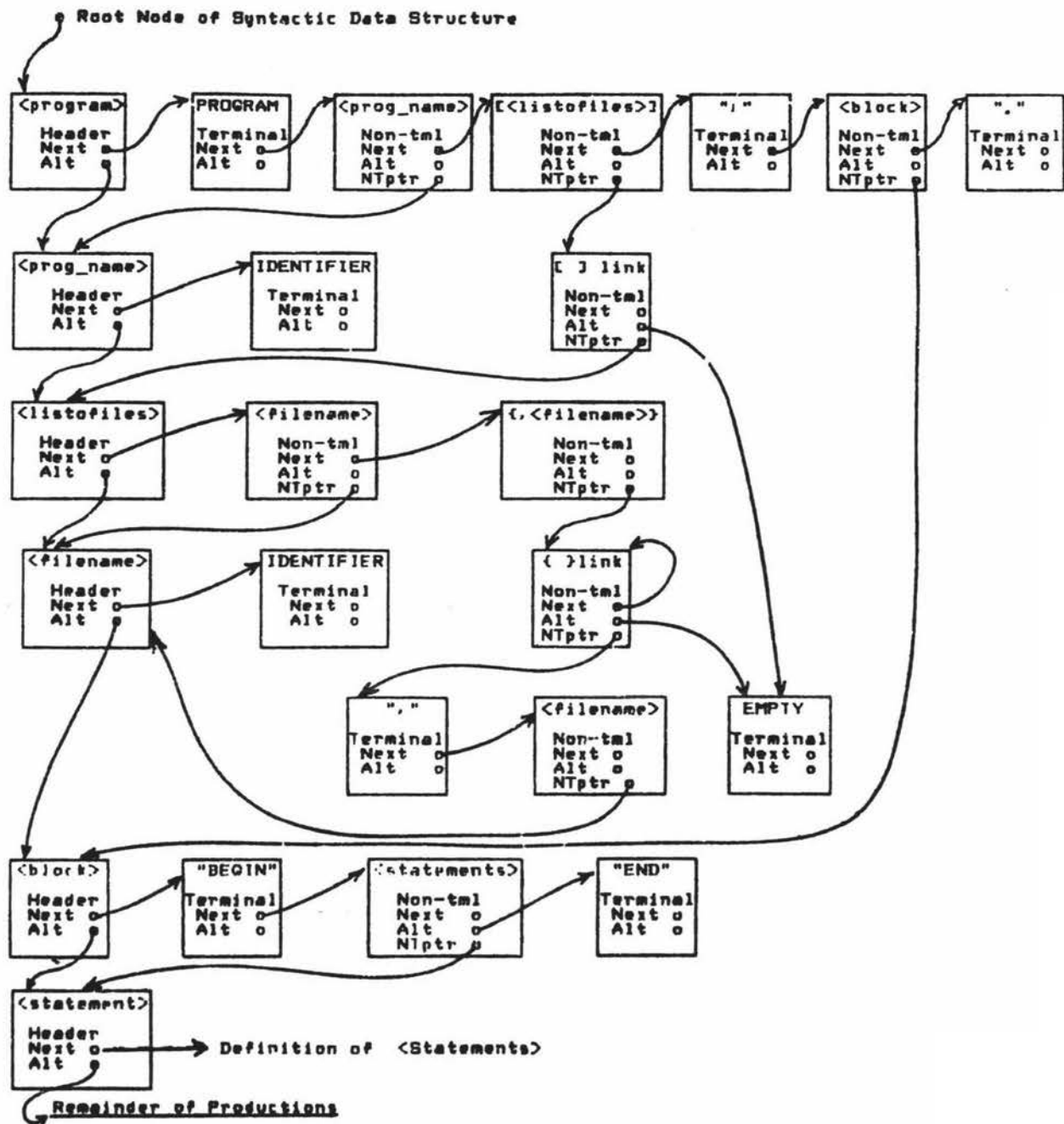


Fig 2.2 - Data Structure for the Pascal Subset

When the data structure is used to guide a parser, it is possible to scan the structure and find which symbols may possibly come next. In fig 2.2 for example, examining the definition of <program> it is evident that the symbol PROGRAM must be present - it has no alternative. The non-terminal "<list\_of\_files>" has an alternative of EMPTY - a terminal symbol that matches any input symbol - and is therefore optional. The semicolon, the non-terminal "<block>" and the dot are also required. However <block> produces BEGIN and END and so these too are required. From the syntax alone, the initial form of a program can be deduced and (with a little pretty-printing) displayed (fig 2.3). The undefined non-terminal productions are represented by the name of the non-terminal symbol.

```
PROGRAM <program_name> [ <list_of_files> ] ;
BEGIN
END .
```

```
Correct Symbols   : identifier
Syntax            ::= <program_name>
```

Fig 2.3 - Program Form derived from Syntax Definition

The EBNF syntax may be regenerated from the structure, and so it is possible to provide at all times a display of the production currently being parsed and a list of all the possible input symbols (fig 2.3). Neither of these uses any information other than that contained in the input grammar itself, so these displays can be generated for any input grammar.

In order to enable the editor to associate the print-formatting commands with the correct parts of a program, such information is input at the same time as the corresponding part of the grammar. The details of the print formatting commands will be described in chapter 3. Here, it is sufficient to note that the layout of any production can be defined in the input grammar. From the information contained in the syntactic definition the editor presents the user with skeletal program (fig 2.4).

## 2.2 The User's View

The display after the editor has been invoked and initialised with the grammar of Pascal is shown in fig 2.4. The current position of the parser within the data structure (to all intents and purposes, the program) is indicated by a highlighted non-terminal name or terminal symbol. In the printed examples given here highlighted regions are underlined. The internal mechanism used by the editor to store the user program is a list of nodes, which parallels the syntactic data structure. The exact structure will be fully described in chapter 3. However it is important to note that the structure forms a tree with non-terminal definitions being the nodes and terminal symbols forming the leaves.

```
PROGRAM <program name>;  
  
BEGIN  
END .  
  
Correct Symbols : identifier  
F0=HELP | Syntax ::= <program_name>
```

Fig 2.4 - The Initial Appearance of a Pascal Program

### 2.3 The Display

The first twenty lines are dedicated to the user's program, the next line is the status line (usually blank) for warnings and questions unrelated to the program, and the last three lines display the help information.

The HELP area normally shows:

- 1) A list of all the symbols that would be correct at the current cursor location.
- 2) An Extended BNF definition of the section of the program under the cursor.

Alternatively, depression of F0 (the "zeroth" function key) will display a brief explanation of the effects of all the function keys, in the HELP area. A prompt to this effect (F0 = HELP) is always on the screen.

The technique of using a highlighted cursor region rather than a point cursor, to indicate the current position of the parser is similar that adopted in the ALOE editors, and here it is used to clarify the operation of the DELETE command. Because the user program is stored in the editor in a data structure whose nodes are organised in the same way as those of the syntactic data structure, the current "position" in the program data structure may either correspond to a leaf node (a terminal symbol) or a non-terminal node (a non-terminal symbol) in the syntactic data structure. In the latter case, a DELETE command will remove as much of the entered program as corresponds to the complete syntactic subtree of the non-terminal node. If only a point cursor were used, the extent of the subtree about to be deleted would be ambiguous, so a highlighted region that covers the current node and its complete subtree is used - the extent of the cursor is now unambiguous.

#### 2.4 Inserting User Input

The symbols surrounded by angle brackets such as <file\_identifier> are called placeholders and wherever they appear, the user must enter something. To expand a placeholder, the user must first position the cursor on the appropriate placeholder (see later for an explanation of the "appropriateness" of a placeholder) and enter a symbol. The symbol entered must be one of those listed in the "correct symbols" list in the HELP display.

User input is buffered into lines to enable typing errors to be corrected with the backspace key in the usual way. A line is accepted by the editor when either the RETURN key or one of the special function keys is pressed.

If the user's input is correct at the cursor position then the placeholder is expanded to include the new symbol or production and the display will reflect the change. Erroneous entries will cause the terminal to beep to alert the user to the correct alternatives at the bottom of the screen. Placeholders may be expanded in any order, so declarations may be added as necessary.

In fig 2.4, both the position of the cursor on the placeholder <program\_name> and the contents of the help display indicate that an identifier is the only valid symbol. Typing any identifier, "demo" for example, will cause the program to change to the representation shown in fig 2.5. The entry of anything other than an identifier would cause the editor to display the erroneous symbol on the status line and emit

a beep at the user as a warning - the program would be unaltered.

```
PROGRAM demo [<list of files>] ;  
  
BEGIN  
END .  
  
First Occurrence of Identifier : demo  
Correct Symbols      : ( Nothing  
FO=HELP | Syntax ::= [ <list_of_files> ]
```

Fig 2.5 - "Demo" is entered as the program name

The "First Occurrence of Identifier : " message appears whenever the editor encounters a new identifier. In languages requiring variables to be declared this message should occur only in declarations - Its occurrence outside declarations indicates an undeclared identifier. The message is accompanied by a beep.

Referring back to fig 2.5, it can be seen that the <list of files> production is optional. The entry of a left parenthesis will cause the entry of the "list of files" template and the display will change to that in fig 2.6. Pressing RETURN will cause the cursor to skip to the next possible place that the user can enter a symbol. RETURN is one of several keys that will move the cursor to the next possible point of user input. A complete list of cursor commands will be given shortly.

```
PROGRAM demo ( <file name> ) ;  
  
BEGIN  
END .  
  
Correct Symbols   : identifier  
FO=HELP | Syntax ::= <file_name>
```

Fig 2.6 - After Entering a "(" to start a list of files

### 2.5 Displaying Optional and List Placeholders

The skeletal programs in figs 2.4, 2.5 and 2.6 lack declarations and statements. These are two examples of optional sections of program that are known to the editor but are not displayed to avoid cluttering the screen.

However, when the cursor is on one of these optional placeholders, it is displayed and the "correct symbols" field in the HELP area will show a list of symbols, each of which will select a particular option. This is illustrated by the appearance of [<list\_of\_files>] placeholder in fig 2.5. The inclusion of the word NOTHING in the list of correct symbols indicates that the placeholder is optional.

These commands are bound to the arrow and function keys on the Visual 200 terminal. The association of specific key sequences with functions is localised within the program and would be easy to modify to suit other types of terminals.



## 2.6 Cursor Movement Commands

The only places the cursor will stop are placeholders, including optional and list placeholders, and symbols entered by the user. The cursor cannot be positioned on any symbol that is required by the context and the syntax, as such symbols are automatically inserted by the editor. It must be possible to stop the cursor on user-entered symbols so that they may be changed if required.

### RETURN key

Move to the next possible alteration point including optional and list placeholders. This makes the optional and list placeholders, if any, visible. The return key only functions in this fashion when the line is empty.

### <- / -> Back / Forward Arrows

Move to the previous/next possible insertion point or user-entered symbol. This causes the cursor to stop at any point that any modification is possible (alteration points).

Top - F7 Move to the first alteration point in the program.

### F1/F2 - Reverse/Forward Symbol Search

The user is prompted for a terminal symbol and the cursor is moved to the appropriate occurrence of that

symbol (towards the top (F1) or the bottom (F2) of the program). If the symbol cannot be found, a message to that effect is displayed. The entry of an empty line as the search symbol will result in the last symbol being reused and to avoid confusion, the editor redisplay the last symbol before proceeding with the search. This prompt is for the user's benefit and prevents the case of the NOT FOUND message being returned when the last symbol is not as the user remembered. The repetition key (F13) will cause the command to be repeated, using the same symbol.

The search command is useful for quick movement around the program and for locating all instances of a specific symbol. Unlike a conventional editor's string search, searching for "b" will find only the locations where "b" is used as a complete symbol, not all occurrences of the letter "b", such as in BEGIN. When it is used to search for comments or strings, all occurrences of these constructs are stopped on - the actual string or comment is not examined. As most searching is for identifiers and reserved words, this is not normally inconvenient. To skip through the program locating the "repeat" key can be used.

F3/F4 - Previous/Next Placeholder

Move the cursor to the previous/next essential placeholder.

Up Arrow Key

ASCEND the program tree. This will cause the cursor to encompass (and therefore highlight) larger and larger sections of the program. It is used in preparation for a CLIP or DELETE command.

Down Arrow Key

This command is used in two different ways, the first is to DESCEND the program tree to the first possible alteration point. This is broadly speaking the opposite of the ASCEND command. The other use of the down-arrow is to force the generation of a subtree for an optional or list node. The use of this command is discussed below.

Examples Showing the Use of Cursor Movement Commands

```

PROGRAM demo ( <file_name> ) ;
[LABEL]
BEGIN
END .

```

```

Correct Symbols   : LABEL Nothing
FO=HELP | Syntax ::= [ LABEL <label> { , <label> } ; ]

```

Fig 2.7 - After Hitting RETURN while on <file name>

The "<file\_name>" placeholder is left in position to indicate that a filename is required and the cursor moves to the optional label declaration placeholder. If no labels were wanted, pressing RETURN again would skip to the next alteration point - the constant declarations (fig 2.8).

```

PROGRAM demo ( <file_name> ) ;
[CONST]
BEGIN
END .

```

```

Correct Symbols   : CONST Nothing
FO=HELP | Syntax ::= [CONST <CONST_defn> {;<CONST_defn>}
;]

```

Fig 2.8 - The Prompt for the Optional Constant Declarations

Entering "CONST" would cause the insertion of a template for Pascal constant declarations (fig 2.9). The terminal symbols "=" and ";" must be present in a constant declaration and are therefore provided by the editor. Identifiers, numbers and strings are treated as terminal symbols in the grammar but as they require further definition from the

user, the appropriate placeholder is left as a prompt.

```

PROGRAM demo ( <file_name> ) ;
CONST
    <constant_name> = <constant> ;

BEGIN
END .

Correct Symbols   : identifier
FO=HELP | Syntax ::= <constant_name>

```

Fig 2.9 - The display after requesting a CONST declaration

Entering a constant name "c1", in accordance with the help information, causes the name to be incorporated into the program. The cursor moves to the next possible alteration point (see fig 2.10). Note, in particular, the list of correct symbols.

```

PROGRAM demo ( <file_name> ) ;
CONST
    c1 = <constant> ;

BEGIN
END .

First Occurrence of Identifier : c1
Correct Symbols   : number identifier + - string
FO=HELP | Syntax ::= <constant>

```

Fig 2.10 - A List of Correct Start Symbols for <Constant>

Entering any of a number, an identifier or a string would cause the entry to replace the placeholder "<constant>". The entry of "+" would cause the "<constant>" placeholder to change to "+ <constant\_value>" and the help information to show that "<constant\_value>" could be either a "number" or an "identifier" (see fig 2.11).

```

PROGRAM demo ( <file_name> ) ;
CONST
    c1 = + <constant> ;

BEGIN
END .

Correct Symbols   : number identifier
FO=HELP | Syntax ::= <constant_value>

```

Fig 2.11 - The Development of a Constant Declaration

Notice that the "First Occurrence..." message has disappeared. The messages on the status line are transient and will disappear as soon as when the user presses any key.

#### The Use of the DOWN-ARROW to Force Subtree Generation

There are occasions when the occurrence of multiply nested list or optional node will cause sections of a program to be unreachable. For example, consider the top-level grammar for Snobol in fig 2.12.

```

<program> ::= { <statement> }

<statement> ::= [<label>] [<subject>] [<rest_of_Snobol>]

```

Fig 2.12 - Top Level Grammar for Snobol

The initial display will show "{ <statement> }" and the help information that a label is an identifier. There is no way to tell the editor that an identifier entered is to be used as the subject, not the label. Hitting RETURN doesn't help as the only alteration point is the

current node - the cursor doesn't move. A method of expanding the <statement> placeholder to "[<label>] [<subject>] [<rest of Snobol>]" is needed, with the cursor on <label>. Pressing RETURN would then skip to the next alteration point - "[ <subject> ]" - as required. When the cursor is on an optional or list node the DOWN-ARROW key will force the expansion of an optional or an iterated subtree to enable the cursor to be positioned on the required placeholder within the subtree. No confusion between the two uses of the DOWN-ARROW key should occur, as it functions as an EXPAND command only on optional or list nodes that have no expansion (in which case descending doesn't make sense). All other times it behaves as a DESCEND command.

### 2.7 Marking and Returning to Marked Nodes

In order to enable very rapid movement around the program, markers, labelled A to Z, are provided. Case differences are ignored. The effect of these markers is to associate a letter with the current node. To associate a marker with the current cursor position the user presses the SET-MARKER key (F8). The prompt - "Set Which Marker A-Z ? " is provided by the editor. A reply of any letter will set a marker, any other key will cause the command to be ignored - with an appropriate message. To return to that node the user gives a MOVE-TO-MARKER command (F9) followed by the name of the marker. In Pascal, marker "C" could be used for constants, "T" for Types and "V" for variables. The markers of the current node and the marked node are swapped, so giving the command again with the same marker name will restore the cursor to the original position. Attempting to move to an unset marker will cause the message "Marker has not be set" to appear on the status line,

but has no other effect.



### 2.8 The Delete Command - F5

The DELETE command deletes the section of program currently under the cursor. After DELETE has been invoked, the cursor moves to the parent of the deleted subtree. Therefore, repeated invocations of DELETE will delete successively larger sections of the program. The deleted section of the program is not irrecoverably lost but is copied to a file called "CLIPPED". This section of the program may be recovered either by using the INSERT command to reinsert this CLIPPED section of the program or by using the UNDO command (see later).

The delete command can also be used to change identifiers, numbers and strings. As illustrated above, if the cursor is on an identifier and the DELETE key pressed, the identifier is replaced by its appropriate placeholder. For example, if the cursor was moved to DEMO and the DELETE key pressed, the placeholder would revert to <program\_name> and a new name could be entered.

To delete templates, the cursor is moved to any node in the template and the UP-ARROW key is pressed repeatedly, until the cursor covers the construct to be deleted. The DELETE command will now remove the complete region under the cursor. It is still saved in the CLIPPED file. If too many ascend commands are given (repeated use of the UP-ARROW key from any initial position would eventually cause the cursor to encompass the entire program), the "undo" command should be used to restore the cursor to its previous position, not the DOWN-ARROW. The DESCEND command (down-arrow) does move the cursor down the tree, but to the first alteration point, which is usually much

further down the tree than intended.

A sequence of repeated ASCEND commands is illustrated in figs 2.13 to 2.15. Notice that the "correct symbols" and "syntax" fields in the help area change as the cursor ascends the syntax tree.

```
PROGRAM demo ( <file_name> ) ;
CONST
  cl = + 97 ;

BEGIN
  IF <factor> THEN
    BEGIN
      write ( <factor> ) ;
    END
  ELSE
    BEGIN
      b [<qualifier>] ;
    END ;
END .
```

```
Correct Symbols : identifier
FO-HELP | Syntax ::= <variable> ( [ := <expression> ] )
```

Fig 2.13 - The Start of the Assignment Statement is under the Cursor

```

PROGRAM demo ( <file_name> ) ;
CONST
  cl = + 97 ;

BEGIN
  IF <factor> THEN
    BEGIN
      write ( <factor> ) ;
    END
  ELSE
    BEGIN
      b ;
    END ;
  END .

```

Correct Symbols : identifier goto begin if case while repeat  
 for with reset rewrite read readln write writeln Nothing  
 FO=HELP|Syntax ::= [<simple\_statement>|<structured\_stmt>]

Fig 2.14 - Ascending the Syntax Tree from "b" to Compound Stmt

```

PROGRAM demo ( <file_name> ) ;
CONST
  cl = + 97 ;

BEGIN
  IF <factor> THEN
    BEGIN
      write ( <factor> ) ;
    END
  ELSE
    BEGIN
      b ;
    END ;
  END .
Correct Symbols : ELSE Nothing
FO=HELP | Syntax ::= [ELSE <statement>]

```

Fig 2.15 - The Optional ELSE part is under the Cursor

Entering the DELETE command at the stage shown in fig 2.15 would remove the ELSE part of the IF statement and replace it with the "else" placeholder "[ELSE]". Entering the UP-ARROW command again would cause the complete IF statement to be covered by the cursor. It could then

be deleted and possibly inserted elsewhere by being recalled from the clipped file. The contents of the CLIPPED file are never deleted, just overwritten by other DELETE commands. Therefore the contents of the CLIPPED file may be inserted repeatedly.

The ASCEND command stops on nodes that have productions as their names, on optional node, and list nodes. On occasions the cursor does not ascend as far as the user intends and the command must be repeated. This is dependent on the number of productions in the original input syntax - the more productions, the more places there are to stop.

### 2.9 The Insert Command - F6

This command inserts the most recently clipped section of program at the current cursor position. The editor attempts to incorporate all of the clipped subtree at the current cursor position, However any error will cause the insertion to be abandoned. The status line will, as before, indicate the erroneous symbol. The clipped section of the program may be inserted at more than one location as required.

### 2.10 Reading and Writing Files

The editor provides commands to write its current program and later, to read it back again. These commands are detailed below.

#### WRITE [filename] - F11

The current program is written to two files, a plain text file suitable for input to a compiler or interpreter, and a code file for reading

back into the editor. Both files have a suffix: the plain text file has the language name and the GED code file is suffixed by ".GED". For example, if the language was Pascal and the command "WRITE DEMO" given, the files "DEMO.PASCAL" and "DEMO.GED" would be written. If no filename is given the input filename is used. The lack of both an input and output filename is an error.

#### READ filename - F10

The GED code file of the given filename is read into the editor. For example, to continue work on the file "demo", the command "READ DEMO" is given. The editor appends its code suffix and reads the file "demo.ged". The named file is inserted at the current cursor position, without erasing the current program. To read in a complete program the display must be in its initial state - this can be achieved with the ASCEND and DELETE commands. This deletes the current program (if any) and could be done by the READ command itself, but in order to limit the number of commands, this is not done.

The input to the editor is designed to be essentially a program without compulsory terminal symbols, and this would seem to preclude the input of files containing complete programs or sections of program. However, the editor ignores all redundant symbols in the input stream and it is therefore possible to modify existing programs, to include useful subroutines or to continue writing a partially completed program using GED.

### 2.11 Undo Function - F12

The editor provides the ability to unwind previously entered commands. This provides a means to explore the editor commands without causing irreversible alterations to the program tree. Modifications made to the program during an INSERT operation are ignored by the UNDO command to avoid filling the undo stack. Therefore after an INSERT command, the UNDO may be used to restore the program to the state it was in prior to the insertion.

### 2.12 A Command Summary in Function Key Order

- F0 - Change the HELP display between the "Current Production/Correct Symbols" display and a brief summary of the Function key commands.
- F1 - Search Backwards for a user entered symbol.
- F2 - Search forwards for a user entered symbol.
- F3 - Move to previous required placeholder.
- F4 - Move to next required placeholder.
- F5 - Delete the region under the cursor (Use with Up-arrow).
- F6 - Insert the last deleted region at the cursor position.
- F7 - Go to first alteration point in the program.
- F8 - Set a marker at the current cursor position.
- F9 - Return to a previously set marker.

- F10 - Read a Ged-format file.
- F11 - Write both a Ged-format and a print file of the program.
- F12 - Undo the recent modifications to the program.
- F13 - Repeat the last command - Most useful for Searches.
- RETURN Key - Move to the next alteration position in the program.
- Left-Arrow - Move to the last alteration point before the cursor.
- Right-Arrow - Same as Return - for consistency.
- Up-Arrow - Move the cursor up the program tree to encompass more of program - Used in preparation for a DELETE command (F5).
- Down-Arrow - Move into the subtree of the current to first alteration point. If the current node is a optional node or list node without a subtree (it's unexpanded) then create one, and then move to the first alteration point in the new subtree.



ESC ESC - Redraw the screen - useful if it has been corrupted in some way (e.g. system messages).

### 2.13 Summary

The editor described in this chapter is designed to cater for a very wide variety of programming languages while preventing all syntactic errors. The editor is economical in terms of keystrokes required and while requiring a different approach to the construction of a program, is not difficult to use. The display of the current production under the cursor and the list of correct symbols, together with the undo facility provide a gentle introduction to the constructs of the language. This environment, while strange for those accustomed to the more conventional methods of program creation, may be especially suited to beginners who have no unlearning to do.

## Chapter 3

### GED - Its Internal Architecture

#### 3 GED - Its Internal Architecture

This chapter describes the data structures used within the editor to represent the syntax and the user's program, and the methods of manipulating and moving around these structures in order to provide the facilities described in chapter 2. The topics covered are: the syntax and its internal representation, the representation of the user program, and the implementation of user commands.

GED is written in PASCAL, using only the non-standard features of the "otherwise" option on a CASE statement and the ability to associate an external filename with an internal name inside the RESET and REWRITE statements.

#### 3.1 The Input Language Syntax

GED is intended to be a syntax-directed editor that reads the syntax of the desired user language as data. Therefore, a machine readable syntax notation must be used. The most common form of syntax notation is BNF [Backus 1959], but the use of recursion to provide repetition and the use of an explicit EMPTY symbol render the notation clumsy and obscure. Tramline (syntax) diagrams are another common form of syntax notation but the notation is graphical and therefore not suitable for direct entry into a machine. The syntactic notation TWIJI [Lyons 1983]

- a machine-readable form of the tramline diagram - could have been used but as BNF and its variants are more widely understood, a variant of Extended BNF has been chosen as the input for GED.

The major limitation of BNF is its clumsy method of handling repetition and optionality using recursion and the empty string. Wirth [1977] suggested a syntactic notation derived from BNF that avoids the use of an explicit symbol for the empty string by adding constructs for optionality and repetition - "[ ZZZ ]" to indicate that ZZZ is optional, and "{ zzz }" to indicate that zzz may occur zero or more times.

In terms of standard BNF:

$\langle D \rangle ::= [ ZZZ ]$  is equivalent to:  $\langle D \rangle ::= ZZZ \mid \langle \text{empty} \rangle$

$\langle D \rangle ::= \{ ZZZ \}$  is equivalent to:  $\langle D \rangle ::= ZZZ \langle D \rangle \mid \langle \text{empty} \rangle$

$\langle \text{empty} \rangle ::=$

However, the syntactic notation suggested by Wirth differs from standard BNF in its method of specifying terminal and non-terminal symbols. Instead of delimiting non-terminal symbols with angle brackets and letting terminal symbols represent themselves, his notation delimits terminal symbols with quotation marks and does not delimit non-terminals. For example, a simple "IF statement" would be defined as:

BNF : <if statement> ::= IF <expression> THEN <statement>

Wirth : ifstatement = "IF" expression "THEN" statement .

The use of quotation marks around terminal symbols in Wirth's notation has the advantage that no conflict arises between the use of a symbol in both the meta-notation and the language being defined.

GED uses the extensions of braces to indicate repetition, square brackets to indicate optionality, and parentheses to indicate grouping as suggested by Wirth, but leaves the remainder of the meta-notation intact - consistent with standard BNF. This is in accordance with the extensions suggested by Pagan [1981]. GED will therefore accept either standard BNF or this variant. Should it be desired to change the input format to conform completely to that suggested by Wirth, only minor changes to the syntax building procedures would be necessary.

The symbols "<", ">", "[", "]", "{", "}", "|", "(", ")", "\$", "\$\$" are part of the meta-language but may also be part of the language being defined. To enable the use of these symbols within the syntax definition, one of two escape characters - either " or ' - is used. The presence of either of these before another symbol removes any special significance that the symbol normally has in the meta-notation. For example, to indicate that parentheses may surround an expression the syntax definition would be:

`<expression> ::= '( <expression> ')`

OR

`<expression> ::= "( <expression> ")`

Although in these examples both single and both double quotes have been used, the choice is arbitrary, and they can be mixed.

3.2 Definition of the Extended BNF Accepted by GED

```

<syntax_definition> ::= <lexical_info>
                        <definition> { <definition> } '$$          $

<definition>         ::= <left_hand_side> " ::= <right_hand_side>
                        [ printformat <print format definition> ] '$ $

<lexical_info>       ::= [ IDENTIFIER_START_SET <set of characters> ]
                        [ IDENTIFIER_BODY_SET   <set of characters> ]
                        [ START_COMMENT        <character>           ]
                        [ END_COMMENT         <character>           ]
                        [ COMMENT_COLUMN      number >= 1 & =<132 ]
                        [ STRING_DELIMITER    <set of characters> ]
                        [ DELIMITER { :BLANK           |
                                      :END_OF_LINE     |
                                      <set of characters>
                                    }
                        ]
                        ]
                        $

<left_hand_side>     ::= '< <non_terminal_name> '>          $

<right_hand_side>   ::= <concatenated_rhs> { '|' <concatenated_rhs> } $

<concatenated_rhs> ::= <right_hand_options> { <right_hand_options> } $

```

```

<right_hand_options> ::= '{ <right_hand_side> ' } |
                       '[ <right_hand_side> ' ] |
                       '( <right_hand_side> ' ) |
                       <primitive_rhs>                                $

<primitive_rhs>      ::= '< <non_terminal_name> '> |
                       '" <token>' |
                       "' <token>' |
                       <token> |
                       IDENTIFIER |
                       NUMBER |
                       STRING |
                       COMMENT                                         $

<non_terminal_name> ::= Any character sequence excluding >          $
                       - Use ">" for >

<token>              ::= = | '<' | '<=' | '< | >' | '>=' | '$' | '$$' | '&' | '@' |
                       | '+' | '-' | '*' | '/' | ';' | ':' | ',' | '?' |
                       | '(' | ')' | '[' | ']' | '{' | '}' | '!' | '#' | '%' |
                       | '~' | '\' | '^' | '.' | '|' | ':='

<printformat>       Will be defined later.                            $ $$

```

Fig 3.1 - Definition of the Extended BNF accepted by GED

Note: The symbols ":BLANK" and ":END-OF-LINE" are used to include the blank and the end of line character in a set. The reasons for this will be discussed later.

The EBNF grammar may be augmented by print formatting information which is associated with each terminal or non-terminal in the syntax. This is to enable the implementor of an editor to specify how programs are to appear when they are printed. This function does not affect the actual form of the input grammar and may be omitted entirely if desired. The formatting commands will be described in detail after the internal representation of the user's program has been defined.

The symbols IDENTIFIER, NUMBER, STRING and COMMENT are unusual, as although they may represent a required terminal symbol, the editor cannot know which identifier, number, string, or comment will be entered by the user, and therefore cannot fill in the correct terminal symbol (in the way that is possible with a comma or BEGIN). Therefore these symbols are treated in the syntax as terminal symbols, but are known to the editor to be composite - the actual symbol to be entered by the user. As the form of each of these symbols differs between languages, their syntax is defined in the <lexical info> section of the syntax definition which must precede their first use.

```
% <- This is a comment, indicated by a "%" in column one
IDENTIFIER_START_SET  set of characters
IDENTIFIER_BODY_SET   set of characters
DELIMITERS            set of characters
.. other lexical definitions ...

<Root Node>          ::= <rest of productions>          $
<.....>              ::= rest of syntactic definitions  $ $$
```



Defining the Syntax of Identifiers

To accomodate the wide variety of keywords and reserved words in common use, the characters that may start an identifier and those that may occur after the first character are specified as part of the syntax definition. The keywords IDENTIFIER\_START\_SET and IDENTIFIER\_BODY\_SET denote the beginning of each set respectively. All the characters following the keyword (excluding blanks) become part of the set. The list is terminated by the end of the line.

The regular expression definition of an identifier is:

```
IDENTIFIER_START_CHAR { IDENTIFIER_BODY_CHARACTER }
```

where the braces mean "zero or more of".

For Pascal the definition of Identifier is:

```
IDENTIFIER_START_SET
```

```
abcdefghijklmnopqrstuvwxyzaBcDEFGHIJKLmNOPQRStUVWxyz
```

```
IDENTIFIER_BODY_SET
```

```
abcdefghijklmnopqrstuvwxyzaBcDEFGHIJKLmNOPQRStUVWxyz0123456789
```

These sets are used as the defaults if the definition of either of the identifier start or body sets (or both) are omitted. Note that the use of separate sets for the start and body of an identifier caters for the case of characters that can occur only at the start of an identifier (such as the "&" in SNOBOL), and characters that cannot start an identifier but are allowed in its body, such as the digits in many

languages, the underline in some Pascal implementations, the dot in SNOBOL and the dash in COBOL.

#### Defining String Delimiters

Strings are delimited in most languages by the single quote, but the use of double quotes is also common. The characters that delimit a string may be defined by the user with the `STRING_DELIMITER` pseudo-definition. As before, all following characters on the line (excluding blanks) will become string delimiters.

Once the editor has recognised the start of a string, all characters up to the matching string delimiter, or the end of the line will be included in the string. If no matching quote is found on the same line, one is provided - no warning is issued. The use of both single and double quotes as string delimiters (as in SNOBOL) permits the other type to be used as part of the string (e.g. "It's" or "hello"). For example:

PASCAL

STRING\_DELIMITER '

SNOBOL

STRING\_DELIMITER ' "

If no string delimiters are defined, the single and double quotes will be treated as tokens without any special significance.

### Defining the Comment Syntax

GED is capable of handling comment enclosed in a pair of bracketing symbols, comments preceded by a particular symbol terminating the logical record (at any position within it) and comment preceded by a particular symbol at a particular position in the record. As GED does not provide any mechanism for editing comments, to alter a comment it must be replaced. To prevent the user from creating an arbitrarily long comment, which would then be unalterable, the maximum comment length is one line. This means that long comments must be broken in many single line comments.

The presence of symbol in the START-COMMENT set (which is defined in the syntax) indicates the start of a comment. If an END-COMMENT symbol has been defined, all characters between the start and end of comment symbols become part of the comment. This caters for languages that bracket comments with special symbols, such as the use of left and right braces in Pascal, and the exclamation mark (as both the opening and closing symbol) as in PLZ-SYS [Snook 1978].

The definition of a START-COMMENT symbol, but not an END-COMMENT symbol indicates that the remainder of the line after the start symbol is a comment. This convention is used in Burroughs Extended ALGOL in which a "%" is used as a logical end of record. Some languages have the more restrictive convention that a certain symbol indicates the start of a comment, but only if it is in a particular position on a line. For example, Snobol uses an asterisk in column one as the comment flag. In this case, the corresponding column must be defined in a COMMENT\_COLUMN

declaration. If the COMMENT-COLUMN declaration is omitted, no fixed column is necessary for the START-COLUMN symbol.

The following declarations show the options for these different languages:

PASCAL	START_COMMENT	{	Comment surrounded by { & }
	END_COMMENT	}	
PLZ-SYS	START_COMMENT	!	Comment surrounded by :
	END_COMMENT	!	
Burroughs ALGOL	START_COMMENT	%	Comment is rest of line after %
SNOBOL	START_COMMENT	*	Comment is rest of line
	COMMENT_COLUMN	1	after "*" in column 1.

### Defining Delimiter Symbols

Most languages ignore certain characters, using them only for formatting purposes and to terminate tokens. The most common delimiters are the blank and the end-of-line character. However languages exist in which other symbols may be freely used for formatting purposes but are otherwise ignored. The language PLZ-SYS is unusual in this respect, as no punctuation is defined in the language - there are no specific statement, declaration or expression delimiters. The comma, semicolon, colon, blank, tab, line-feed, return and page-feed characters may be freely intermixed with the symbols of the

language. For GED to handle this type of language, there must be some method of specifying that certain characters are to be ignored. The DELIMITER set is used to do this. GED's pretty-printer will reformat the program when it is regenerated, so only the printable characters (comma, semicolon, colon), the blank, and the end-of-line character need to be specified. The inclusion of the blank and end-of-line characters in the set is awkward, as the blank is used for formatting purposes and is therefore ignored, and the END-OF-LINE indicates the end of a set. Therefore special symbols are necessary to represent these two characters within a set - the symbols :BLANK and :END-OF-LINE are used. These symbols, if present, must occur directly after the symbol DELIMITERS as otherwise multi-character lookahead would be required to determine that ":BLANK" meant the blank, and not the characters ":" "B" "L" "A" "N" and "K". All remaining printable characters on the line will be incorporated in the set.

For example:

PASCAL

DELIMITERS:BLANK:END-OF-LINE

PL2-SYS

DELIMITERS:BLANK:END-OF-LINE , ; :

### 3.3 Requirements of the Internal Syntactic Representation

When using a syntax-oriented editor, the user cannot be expected to enter the program in a continuous stream from start to finish. Mistakes and forgotten items will cause the user to edit different sections of the program in a more or less random order. For example, the user may request an IF statement, enter FOUND as the first part of the (IF) "<expression>", and then receive the message "First Occurrence of identifier : FOUND" - meaning it hasn't been declared. The user may then want to stop entering the partially complete <expression> and move back to the variable declarations to declare "FOUND". In the process, it may also be necessary to then add new TYPE and CONST declarations. There must be no requirement that suspended partially complete parses be resumed in the reverse of the order in which they were suspended - users have their attention distracted or forget. The parsing technique used in a syntax-oriented editor must be able to handle the suspension of a incomplete parse of one production (e.g. <if\_statement> is incomplete), and the resumption of any other partially complete production. With respect to their order in the final program, the input stream of tokens may be discontinuous (because of a jump from one production to another), and may not include all the symbols that will be present in the final program, as required terminals will be inserted by the editor. For example, the input symbols for the above example (omitting cursor movement commands) would resemble "IF FOUND VAR FOUND BOOLEAN".

The disconnected nature of program development greatly constrains the choice of parsing technique that may be used in a syntax-directed editing environment, if syntactic correctness of the program is to be guaranteed at all times. This goal could be attained by reparsing the complete program after the entry of every symbol, but this is too wasteful of processing power to be viable.

As the parse of productions may be suspended and resumed in an arbitrary order, the state of the parser - which production it's parsing and where it's up to within that production - must be accessible, so this information can be saved and restored. A parser that stores this information implicitly cannot be used as there is no way to access the current state. An example of this type of parser is the recursive descent parser, in which the parser state is distributed throughout the chain of return address and local variables on the stack - which is inaccessible. It is therefore impossible to save and restore the parser's current state. This saving and restoring of the current state is akin to a process swap, and could form the basis of an interesting research topic, a short description of which is given in chapter five.

A requirement of this implementation is that the syntax be regenerated for display purposes. The regenerated syntax is used for the placeholder prompts and to display the production currently being parsed, as a guide to the user. The editor is intended to read the language syntax as data, and so no information regarding the syntax may be implicit (written into the code) in the parser itself.

The mechanism chosen to represent the data structure is a network of trees. Each EBNF definition corresponds directly to one of the trees, and the nodes containing non-terminal symbols are linked to the tree defining their syntax. It is this interlinking of the trees that gives the structure its network aspect. The structure is based on one developed by Wirth [xxxx] for BNF, adapted to accommodate the loop and optional constructs, and to enable the syntax to be regenerated from the data structure. Each node in the structure (the syntax) may define either a terminal symbol, or be a pointer to other syntax nodes. Before describing the interconnection of these nodes and their fields, it is necessary to define the representation of terminal symbols within the editor.

#### 3.4 Representation of the Tokens of the Meta and User languages

A lexical analyser which breaks up the input character stream into tokens is used to scan both the input syntax and user's input. The output of the scanner is a sequence of tokens stored in three global variables, "token", "tokenvalue", and "string\_node". "Token" is an enumerated type and indicates the type of token. Some examples are: SEMICOLON, DOT, DOLLAR, TWO\_DOLLARS, STAR, PLUS, BEGIN, WHILE, IDENTIFIER, NUMBER, STRING, and COMMENT. IDENTIFIER, NUMBER, STRING and COMMENT require further information to identify which input symbol was entered. The variables "tokenvalue" and "string\_node" are used for this - these two variables are optional, unlike "token" which is always defined.



For identifiers, "tokenvalue" contains the symbol-table index of the particular identifier. To conserve space (because of Pascal's lack of strings), the spelling of identifiers and reserved words (e.g. IF) is not held in the symbol-table itself, but in a global string area. The symbol-table contains an index into the string area and the length of the identifier. Therefore once the symbol-table index of identifier is known, its spelling may be found.

In the case of NUMBER, "tokenvalue" contains the number's value. For STRINGS and COMMENTS, the global variable "string\_node" contains a pointer to a record containing the string (or comment), its length and its delimiting characters.

#### Token, Tokenvalue and String-node

Routines are provided within the scanner so that, given the triplet "token, tokenvalue, and string\_node" for a particular token, the scanner can regenerate its textual form. Therefore it is unnecessary to store a textual representation of a program - it can be reconstituted from its stream of tokens. Obviously all formatting information is lost when this is done. Although a triplet is always stored when it is necessary to identify a token uniquely, for brevity the triplet will be referred to as a "symbol". These symbols are stored in one variant of the nodes that make up the syntax - the "terminal" variant of the syntax node.

A variant record structure is used to represent the three different types of syntax record node. A tag field with the record indicates the variant applying to a particular node which may be one of "terminal", "header" and "non-terminal". These are used to represent terminals, the names of productions, and non-terminals productions respectively. They will be covered in turn.

#### The Representation of Terminal Symbols within the Syntax Tree

The terminal variant of the syntax node has three field to contain the "token", "tokenvalue" and "string\_node" fields of the token it represents. For example, nodes representing a semicolon, WHILE, an identifier, a number and a string would contain the following information.

Semicolon	TOKEN	= SEMICOLON
	TOKENVALUE	= unused
	STRING_NODE	= unused
WHILE	TOKEN	= WHILY
	TOKENVALUE	= unused
	STRING_NODE	= unused
found (identifier)	TOKEN	= IDTOKEN
	TOKENVALUE	= Symbol-table index of "found"
	STRING_NODE	= unused
19731	TOKEN	= NUMBER
	TOKENVALUE	= 19731
	STRING_NODE	= unused

'hello' (string)	TOKEN	=	STRING	
	TOKENVALUE	=	unused	
	STRING_NODE	=	0	
				Points at string node
			v	
	string		hello	
	length		5	
	start_char		'	The start and end chars
	end_char		'	may be different for
				comments.

All tokens stored within the editor may be traced back to terminal syntax nodes.

### 3.5 Describing the Names Of Productions

A node is associated with the left-hand-side of each production and is used to store the production's name, and point to its definition. This node is called a header (syntax) node. All the header nodes are linked together by the pointer field ALT (see diagram below) and so by following the links all productions (and their names) can be found. This enables a search for the appropriate definition to be undertaken when linking a non-terminal node into the rest of the structure.

Because the names of non-terminal productions are delimited by "<" and ">" in the input syntax, the non-terminal name may contain any printable character. Blanks may be present in the production's name, but are ignored when the name is stored. This is to enable the use of blanks to tidy the layout of the syntax, but avoid the problems that would occur if <withstatement>, <with statement>, < with statement> were deemed to be different. Note that the representation of non-terminal names is quite distinct from the representation of identifiers - any printable character may be used inside a non-terminal

name.

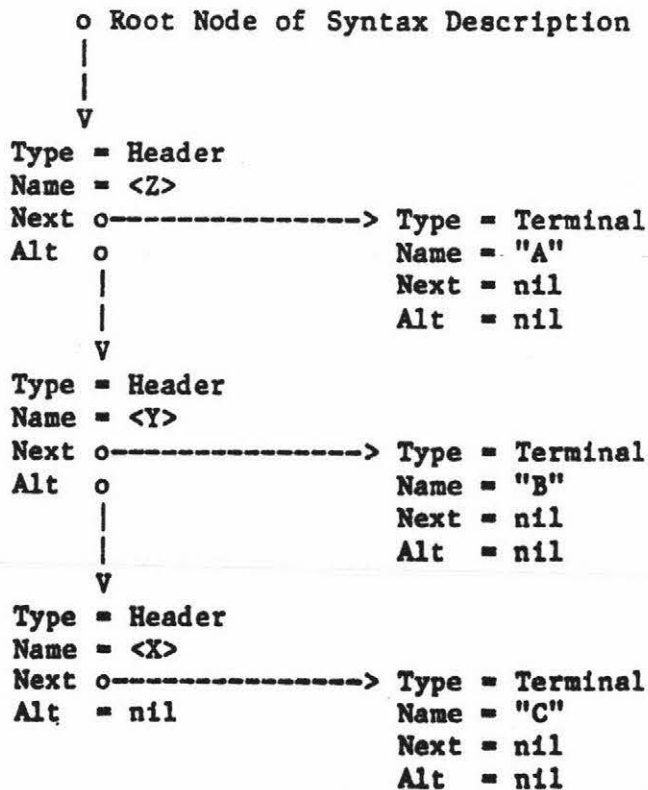
Header syntax nodes contain two pointer fields which point to the (the header node of) the next production and to the syntax nodes corresponding to the right hand side of the EBNF definition respectively.

For example, the productions:

<Z> ::= A

<Y> ::= B

<X> ::= C are represented as:



When the end of the grammar is encountered, any production with a null "next" field (no associated production) has its name printed with the message "No Definition for <undefined name>".

Now that terminal symbols and productions have been defined, some mechanism of describing sequences of these items is necessary. To accomplish this, a new form of node is used.

### 3.6 Non-terminal Syntax Nodes

The non-terminal node is used to construct sequences of nodes, to indicate alternative productions, and to provide a mechanism to represent the optional and list productions.

The "non-terminal" syntax node has a "next", an "alternative" and a "definition" field. These fields are to refer to the successor to the production pointed at by the current node or a possible alternative to it. However as the non-terminal node does not define a terminal symbol, some method of indicating which production must be parsed is necessary. The "definition" field is used for this and points to other syntax nodes, which may be terminal nodes, header nodes, or other non-terminal nodes. The non-terminal node provides the mechanism to build up the structures necessary to represent the constructs of extended BNF. The following examples illustrate the various constructs and their corresponding data structure.

### 3.7 Concatenation and Alternation of Productions

The presence of sequences or alternatives in a grammar always causes an extra level of syntax nodes to be constructed. Sequences of productions are represented in the data structure by a list of non-terminal nodes, their "next" pointers indicating the following non-terminal node in the sequence (fig 3.2). The non-terminal pointers may point to any item syntactic construct, including other sequences.

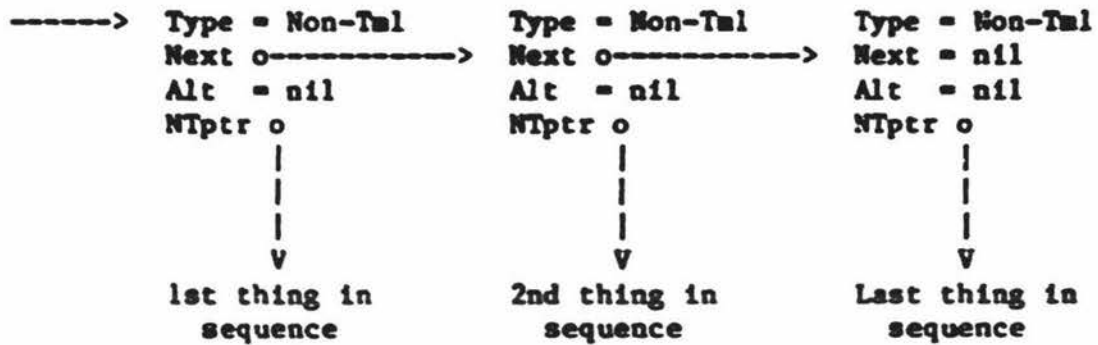


Fig 3.2 - Syntax Node Structure used to Represent Sequences

Alternatives in the grammar also cause the generation of another layer in the syntax structure (fig 3.3). This layer being distinct from the layer of nodes used to indicate concatenation. Keeping the layers for the different constructs separate simplifies the regeneration of the printable representation of the syntax.

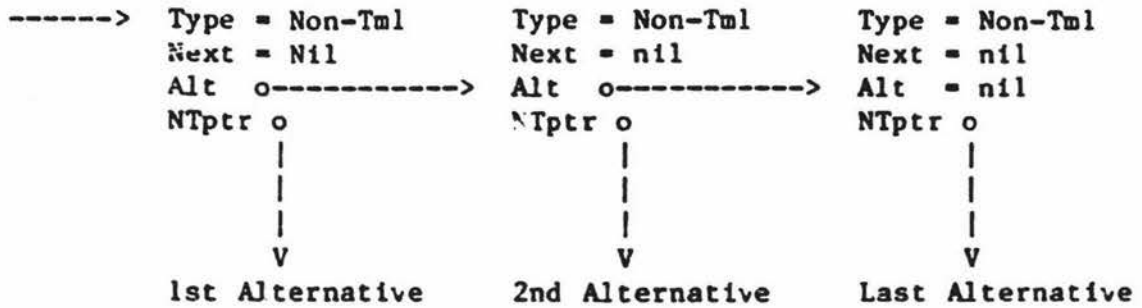
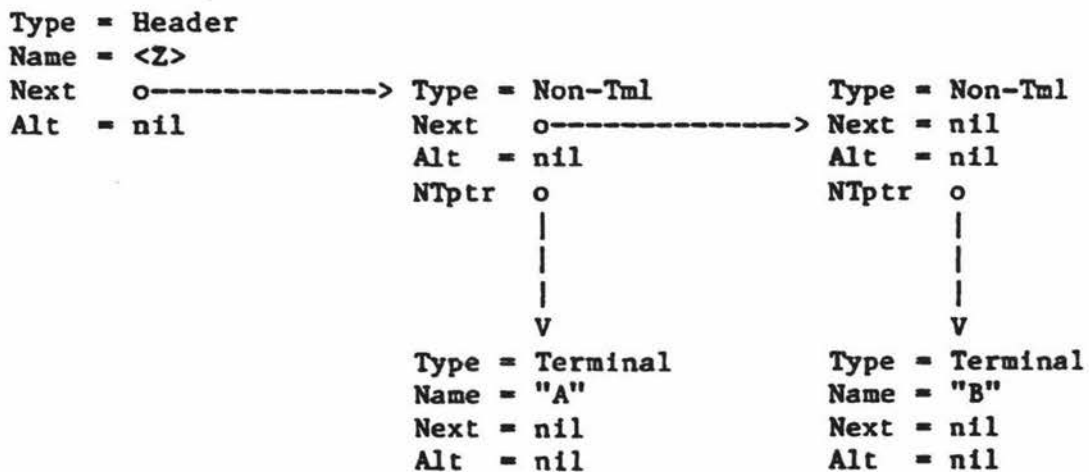


Fig 3.3 - Syntax Node Structure used to Represent Alternatives

Regenerating the printable representation of the syntax corresponds to treating the syntactic structure for each production as a tree, and then performing a depth-first scan over the tree, stopping whenever a header node is encountered. A pointer to a header node is not traced any further. The name of the production is printed instead.

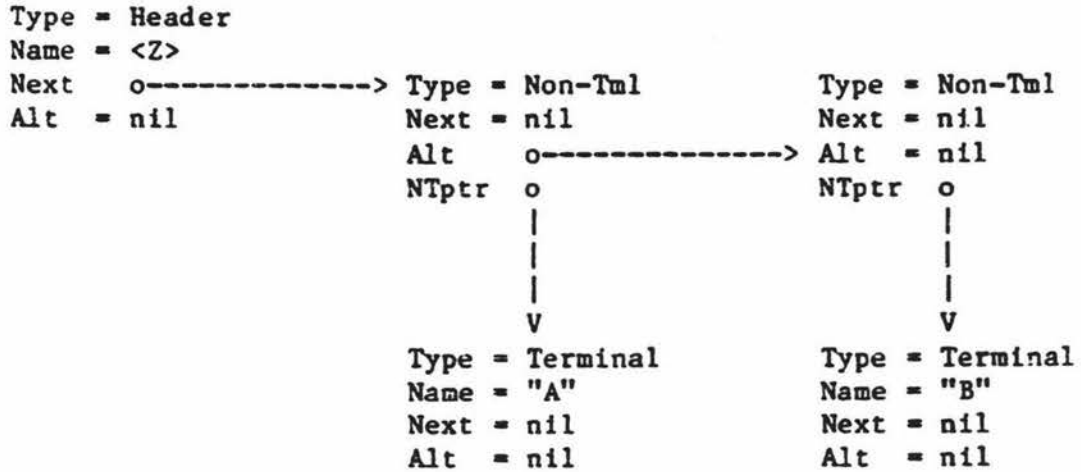
Examples of Simple Syntactic Productions and their Representation

<Z> := A B is represented as:



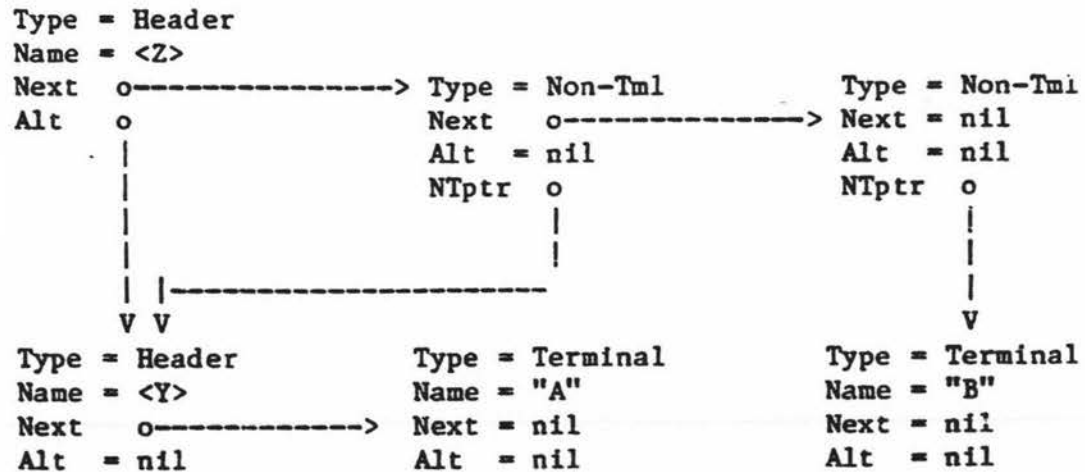
Alternatives to a Production

<Z> ::= A | B is represented as:



The Use of Non-terminal Names within Productions

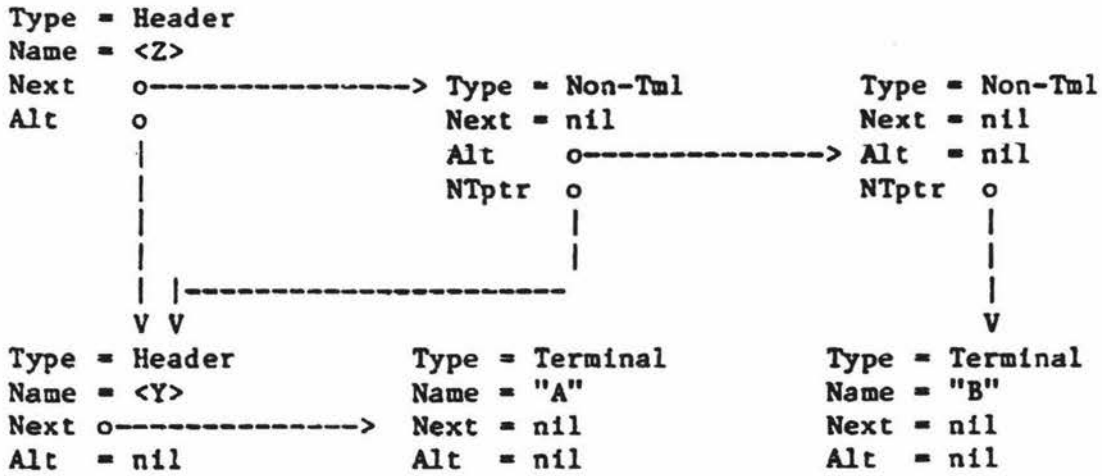
<Z> ::= <Y> B  
<Y> ::= A are represented as:





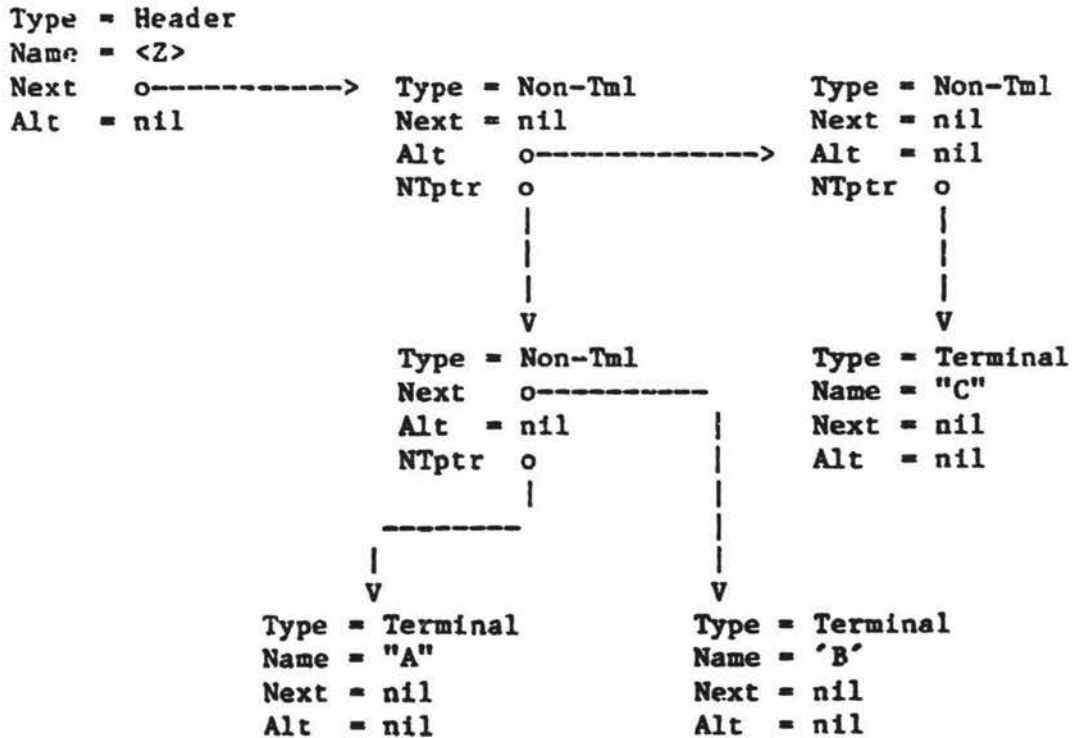
Example of Non-terminal Symbols, and Alternation

<Z> ::= <Y> | B  
<Y> ::= A are represented as:



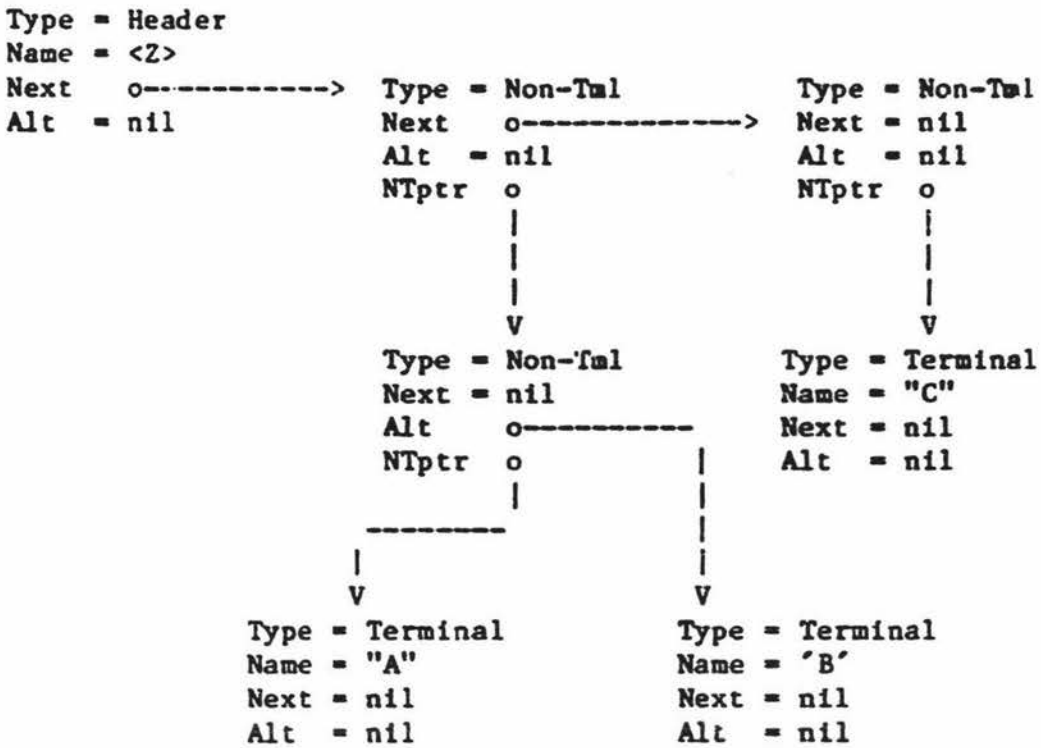
Example of Grouped Symbols, and Concatenation

<Z> ::= (A B) | C is represented as:



Example of Grouped Alternatives, and Concatenation

<Z> ::= (A | B) C is represented as:



Nested constructs form new subtrees in the data structure and therefore, when it is being used to guide a parser, the structure should be scanned depth-first. A parsing procedure designed to work with the constructs given so far is illustrated in fig 3.4.

```

function PARSE (start_node : syntax_node_ptr) : boolean;
var found : boolean;
begin
  case start_node^.node_type of
    terminal      : begin {See if input token same as in node}
                      found := (input_token = start_node^.token);
                      if found then Get_next_token; {into globals}
                    end;

    non_terminal  : begin {Try Depth-first, then alternatives}
                      found:= parse(start_node^.non_terminal_ptr);
                      if not found then {TRY ALTERNATIVE}
                        found := parse(start_node^.alternative);

                      if found then {Trace following productions}
                        found:= parse(start_node^.next);
                      end;

    header        : found := parse(start_node^.next);
  end; {case}
  parse := found;
end; {of PARSE}

```

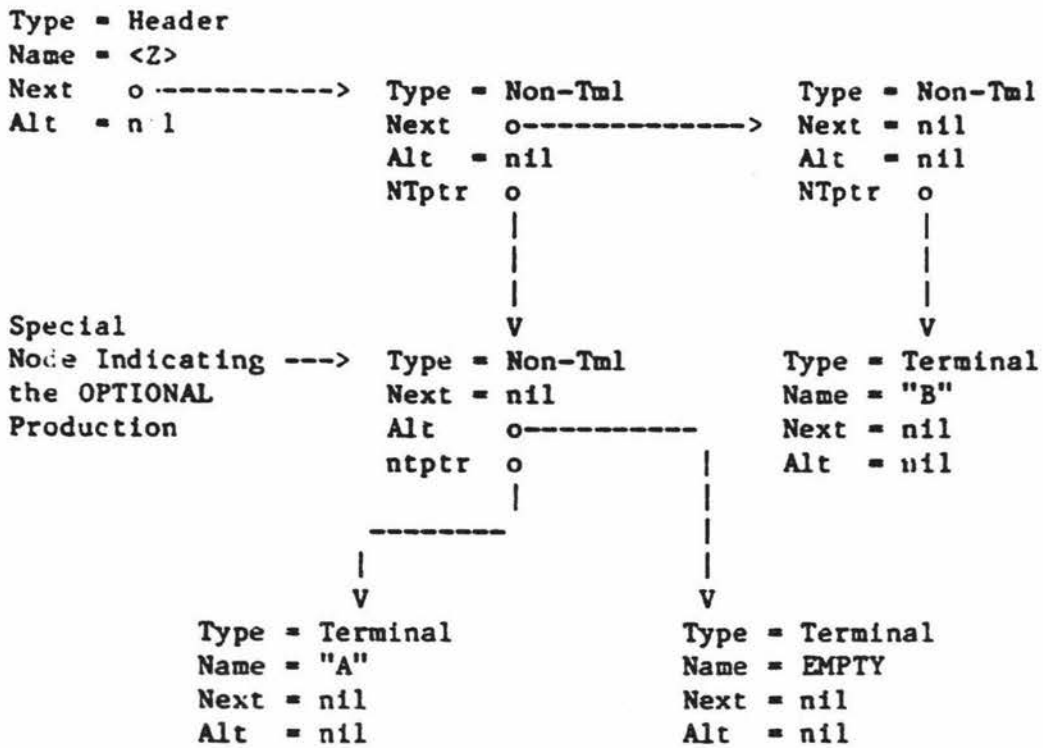
Fig 3.4 - Parsing Procedure to work with Syntactic Data Structure

### 3.8 The Data Structure used to Represent the Optional Symbol

In the constructs described so far there is no mechanism to describe the empty production, and therefore no method of defining a structure to represent the optional production (e.g. [Z]) or the iterated production (e.g. {Z}). There must be some method in the structure of indicating that if the current input symbol does not match the next symbol in the grammar, then that production may be skipped. This situation is covered in GED by defining a special terminal symbol named "EMPTY" that will match any input symbol, and therefore not cause a failure of the parse. "EMPTY" is special in that it does not consume the input symbol, which may then be matched against following productions. Therefore EMPTY does behave in the same manner as the production that derives the empty string.

Example of an Optional Production

For example,  $\langle Z \rangle ::= [A] B$  is represented as:



Optional Production

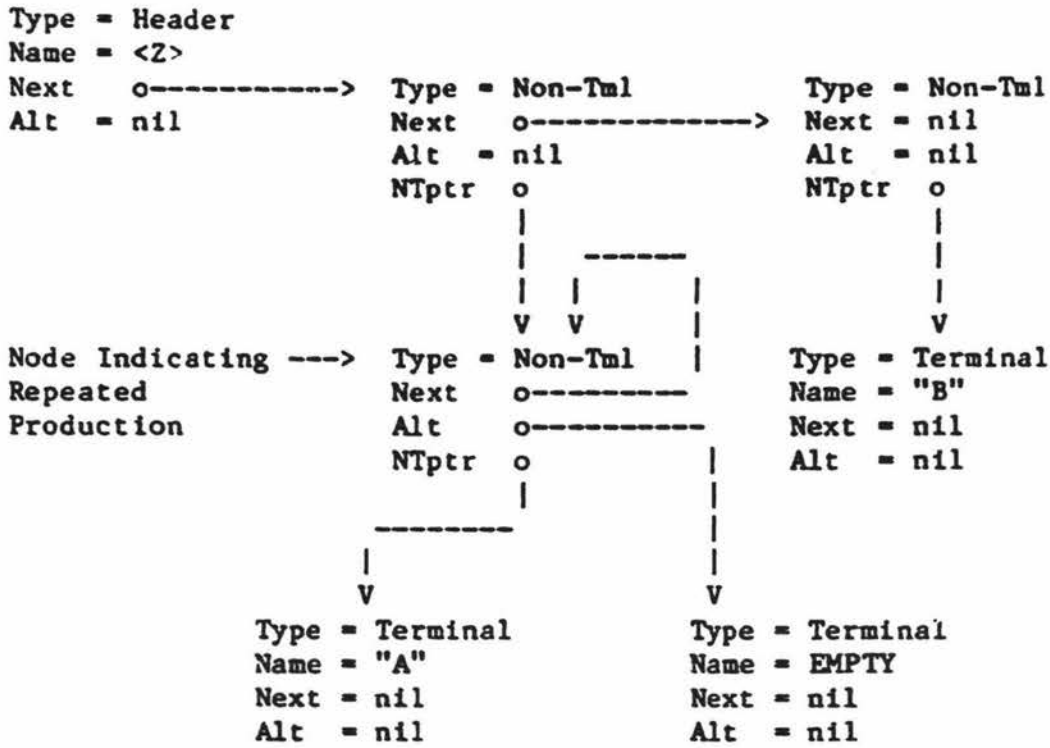
This node will match any input symbol without consuming it.

### 3.9 The Data Structure used to represent the List Construct

The list construct differs from the optional construct only in the number of times the production may be present. For an optional production, the production may be present once or not at all. This is represented by either a parse of the non-terminal pointer of the node indicating the optional production (indicated in the diagram above), or a match of the EMPTY production (represented by the terminal node EMPTY). If however, the "next" field of the special node pointed to itself, the non-terminal field could be parsed as long as the input symbol matched the optional production. This is illustrated below:

Example of an Iterated Production

For example,  $\langle Z \rangle ::= \langle A \rangle B$  is represented as:



Repeated Production

This node will match any input symbol without consuming it.

When eventually the input symbol is not in the start set of the iterated production, the EMPTY field will match, terminating the list. If the input symbol is not in the start set of the repeated production, the node EMPTY will match immediately, therefore the production pointed at by the "non-terminal" pointer of the special may be present zero, one or many times. The modified parsing procedure to handle the presence of the empty symbol (and therefore the optional and list productions) is shown in fig 5.

```

function PARSE (start_node : syntax_node_ptr) : boolean;
var found, token_match, empty_match : boolean;
begin
  case start_node^.node_type of
    terminal      : begin {See if input token same as in node}
                      token_match :=(start_node^.token=input_token);
                      empty_match :=(start_token^.node = EMPTY);
                      found:= (token_match OR empty_match);
                      {Don't consume token if EMPTY match}
                      if token_match then get_next_token;
                    end;

    non_terminal  : begin {Try Depth-first, then alternatives}
                      found:= parse(start_node^.non_terminal_ptr);
                      if not found then {TRY ALTERNATIVE}
                        found := parse(start_node^.alternative);

                      if found then {Trace following productions}
                        found:= parse(start_node^.next);
                      end;

    header        : found := parse(start_node^.next);
  end; {case}
  parse := found;
end; {of PARSE}

```

Fig 3.5 - Parsing Procedure including knowledge of EMPTY symbol

The EBNF grammar in fig 3.1 is LLI and may therefore be parsed by a recursive descent parser to build a graph structure representation of the syntax. The syntactic data structure is built by a recursive descent parser designed to parse the syntax given in fig 3.1. The first production in the syntactic definition is always taken to be the root node. This is arbitrary, but in practise does not cause any inconvenience.

An example of the data structure built for a small grammar (fig 3.6) is illustrated in fig 3.7. In order to enable the diagram to fit on one page, in fig 3.7 and in all future diagrams of the syntactic data structure, when terminal symbols occur in a sequence (e.g. PROGRAM is the first of a sequence), the terminal symbol will be drawn as though it were part of the parent node - the one used to link the items in a sequence together. This is simply to clarify the diagrams by eliminating a level from the structure. It is not to be construed as indicating a change in the syntactic structure from that previously defined.

```

<program>      ::= PROGRAM <program_name> [<output_file>] ; <block>      $
<program_name> ::= identifier                                           $
<output_file>  ::= identifier                                           $
<block>        ::= BEGIN <statement> ; { <statement> ; } END           $
<statement>    ::= IDENTIFIER := <expression>                          |
                  IF <expression> THEN <statement>                      $
<expression>  ::= IDENTIFIER | NUMBER                                  $
$$

```

Fig 3.6 - Syntax used to Illustrate Syntactic Data Structure

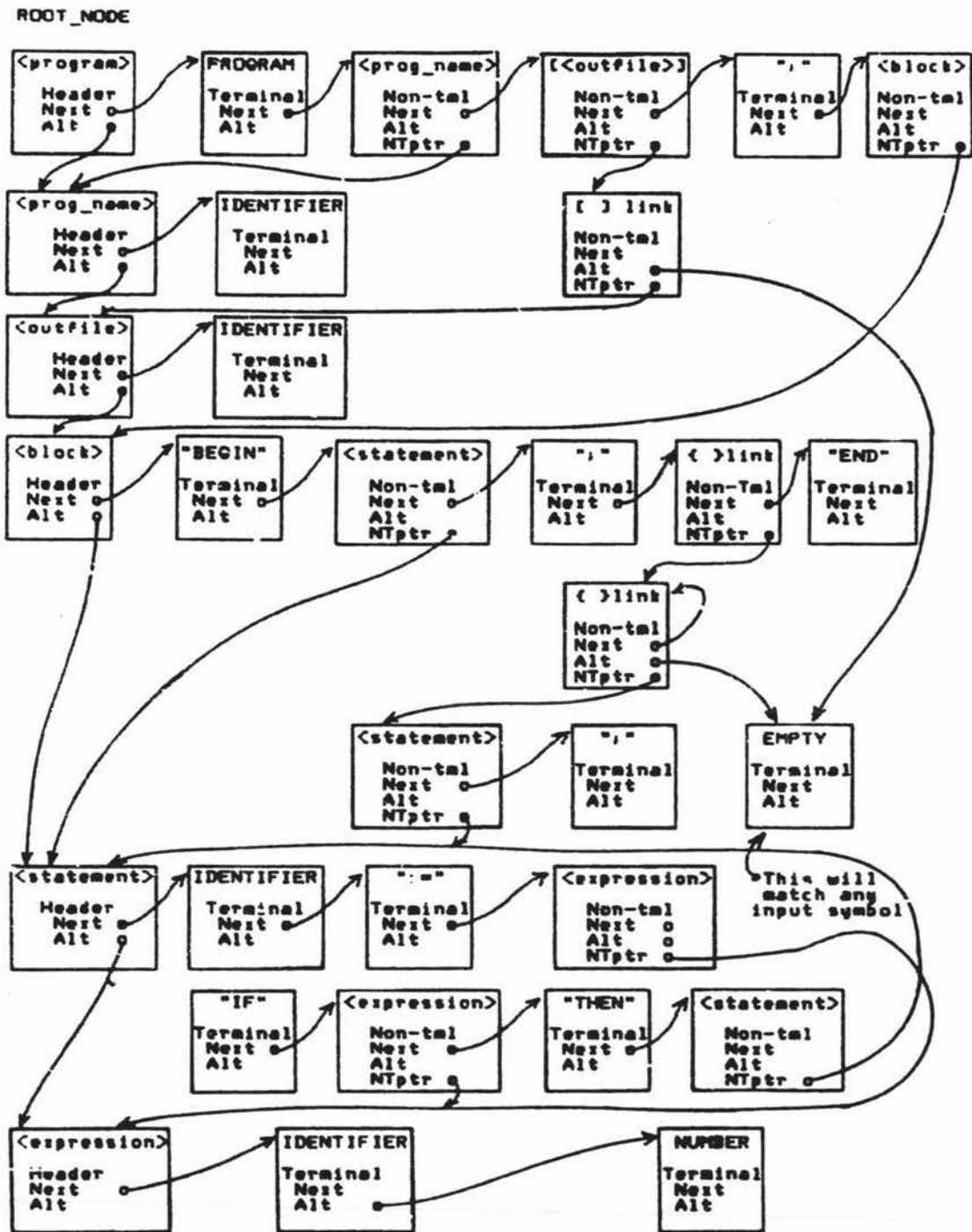


Fig3-7- Data Structure for the Grammar of Fig 2.



### The Syntax Node Variants - A Summary

Each syntax node defines either a terminal symbol or a pointer to other syntax nodes. A tag field in each node indicates its type, which can be either a terminal symbol, a non-terminal (a pointer to other nodes), or a "header" node - a node defining the start of a production. The header node corresponds to the left hand side of an extended BNF definition.

The use of a separate type of node for a header is because of the need to regenerate the syntax tree. The print procedures recursively scan the syntax tree, but they must stop when a header node is encountered. Only the non-terminal name must be printed, not a trace of the actual production. Otherwise, the regenerated syntax for "IF <expression> THEN <statement>" would not contain the names "<expression>" and "<statement>". Instead, all the possible options and alternatives that <expression> and <statement> may derive would be enumerated explicitly. In the case of the grammar in fig 3.6, <expression> would be expanded to "IDENTIFIER | NUMBER", and <statement> would be expanded to "IDENTIFIER := <expression> | IF <expression> THEN <statement>". Also it is possible to get into an infinite recursive loop, as would happen in this IF statement (as <statement> occurs within the IF statement).

### 3.10 Storing a Representation of the User's Program

The representation of a program must satisfy two primary requirements. The first is the ability to suspend a parse at any stage and carry on with another production, which may, itself, have been suspended. The second is to enable a printable representation of the program to be obtained, even if the program is incomplete.

Although the organisation of the data structure used to represent the syntax is obviously closely related to that of the user's program, it is not suitable for storing such a program. A syntactic item such as <expression> is defined only once in the syntax data structure, whereas many instances of it may occur in a program. Conversely, the syntax is capable of specifying an arbitrary number of repetitions of a construct, but has no way of recording the actual number of occurrences. This information must be recorded in the parsing procedures, either implicitly or explicitly.

The parsing function given in fig 3.5 will parse an input stream and return a verdict of success or failure (as true/false), but it is not directly suitable as the parser for a syntax-directed editor. This is because, like a recursive descent parser, it remembers which productions have yet to be completed in the trace of return addresses on the (implicit) return address stack. Therefore the parsing function is satisfactory for a continuous stream of input tokens, but not the disjoint segments of input (intended for different productions) found in a syntax-editing environment.

### 3.11 Recording the State of a Parser Without a Stack

To store the state of the user's program, GED uses a data structure with the same topography as the syntactic data structure, but which contains only the terminal symbols and non-terminal productions actually present in the user's program in its current state of refinement.

The nodes in this structure are called "program nodes" to distinguish them from the nodes used to represent the syntax (syntax nodes). Each program node contains a pointer into the syntactic data structure to define the syntax of the object (terminal or non-terminal) it represents.

Each instance of a syntactic construct (such as <statement> or <expression>) causes the creation of new program nodes that represent just that construct. Therefore no ambiguity can arise regarding the actual number of occurrences any one construct - only a specific number of program nodes pointing to it will be encountered in the program tree.

Although a grammar may specify an infinite number of viable strings, any particular program will contain only a small number. The pointer to the syntactic definition enables the editor to determine whether an input symbol is in the start set of the syntactic productions associated with a particular program node. For example, if a <statement> is possible at a particular place in a program, a program node is allocated to indicate this. The node's definition field

pointer to the header node for the production <statement>, and provides the necessary link between the current state of the program and all of its possible syntactically correct derivations.

A program node whose definition field points to a non-terminal production, such as <statement>, may not as yet have any terminal symbols associated with it. Such a program node is known as a placeholder - it is standing in for, as yet unspecified, terminal symbols. Placeholders occur when a non-terminal production has alternatives, such as the various types of statement, but the user has not indicated which option is wanted. A program that contains placeholders is obviously incomplete, as the very presence of placeholders means there are productions that do not produce terminal symbols. However, as the editor is designed to be interactive, a displayable representation for placeholders must be found.

The obvious solution, and the one used in GED, is to display the syntactic derivation of placeholders instead. It is therefore crucial that a printable representation of the syntax be obtainable from the syntactic data structure. To make sense, the display should not trace any header nodes encountered (i.e. "<statement>" should be displayed, not the options of IF <expr> THEN... & WHILE <expr> DO ... & ...).

### 3.12 The Initial Form of the Program Node Tree

The initial form of the program node tree is simply that of the top level syntax definition (fig 3.8).

o Root Node of Program Node Tree

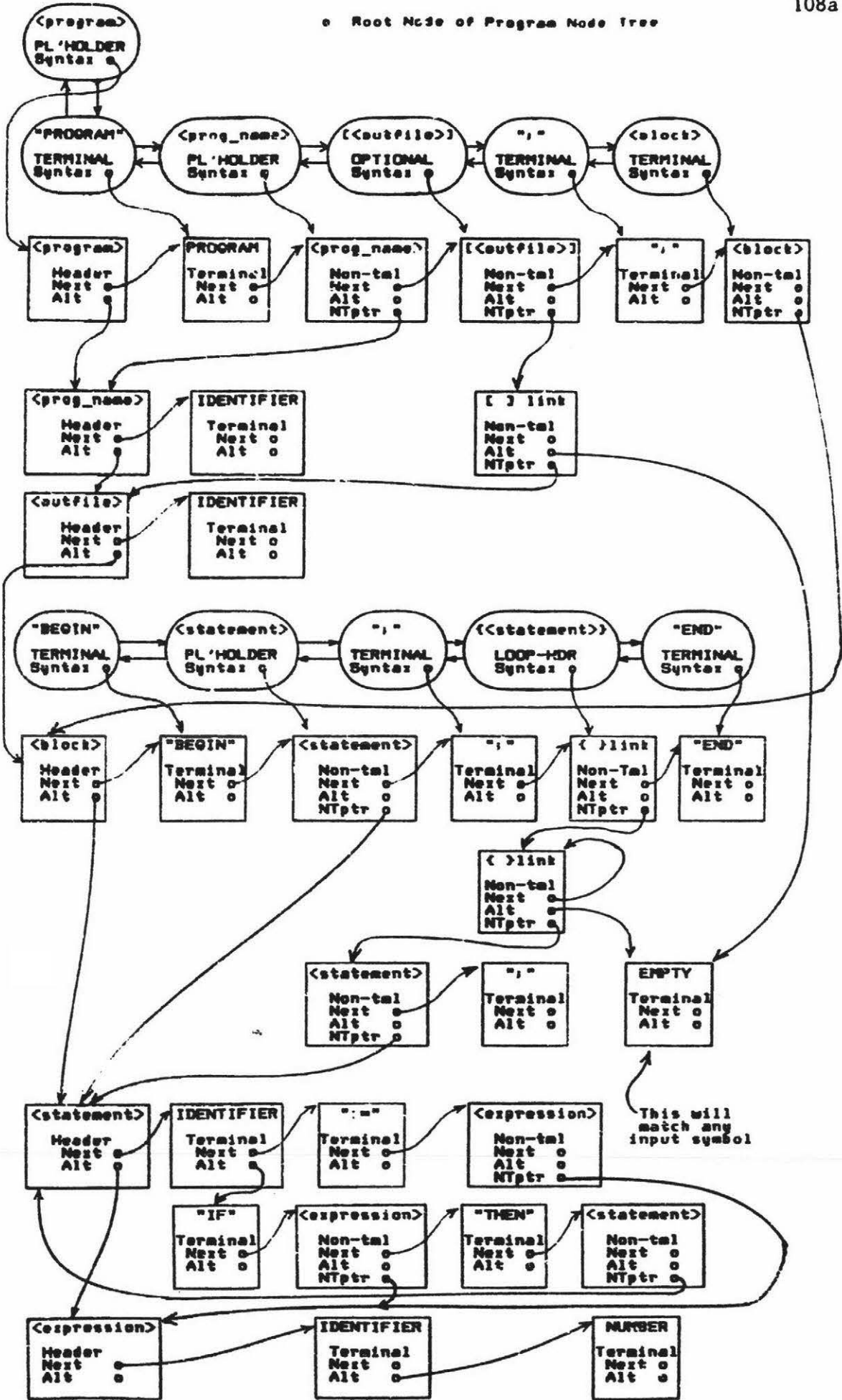


Fig 3-g - Combination of Program Node Tree and Syntax Tree

A program node is allocated for all productions, all possible productions and all terminal symbols present that are directly derivable from the root node and are essential to form a complete program. This is why <block> has been expanded to "BEGIN <statements> ... END".

The program node tree lays out the order of the productions (and pointers to their definitions) that must be present for the program to be valid. It is at this point that this parser differs from the more usual table-driven parser.

The expansion of a placeholder causes GED to create new program nodes. This is illustrated by the expansion of <statement> to an IF statement in fig 3.9. This is distinct from the use of a data structure to guide a recursive parser. Rather than remembering which productions have yet to be completed on a stack, the parser stores this information is stored explicitly in the newly created layer in program node tree. Therefore the nesting of one <statement> within another does not cause any information to be saved implicitly. The expansion of the newly created statement placeholder to another IF statement is shown in fig 3.10. All information concerning the state of the parse is encoded in the state of the program node tree.

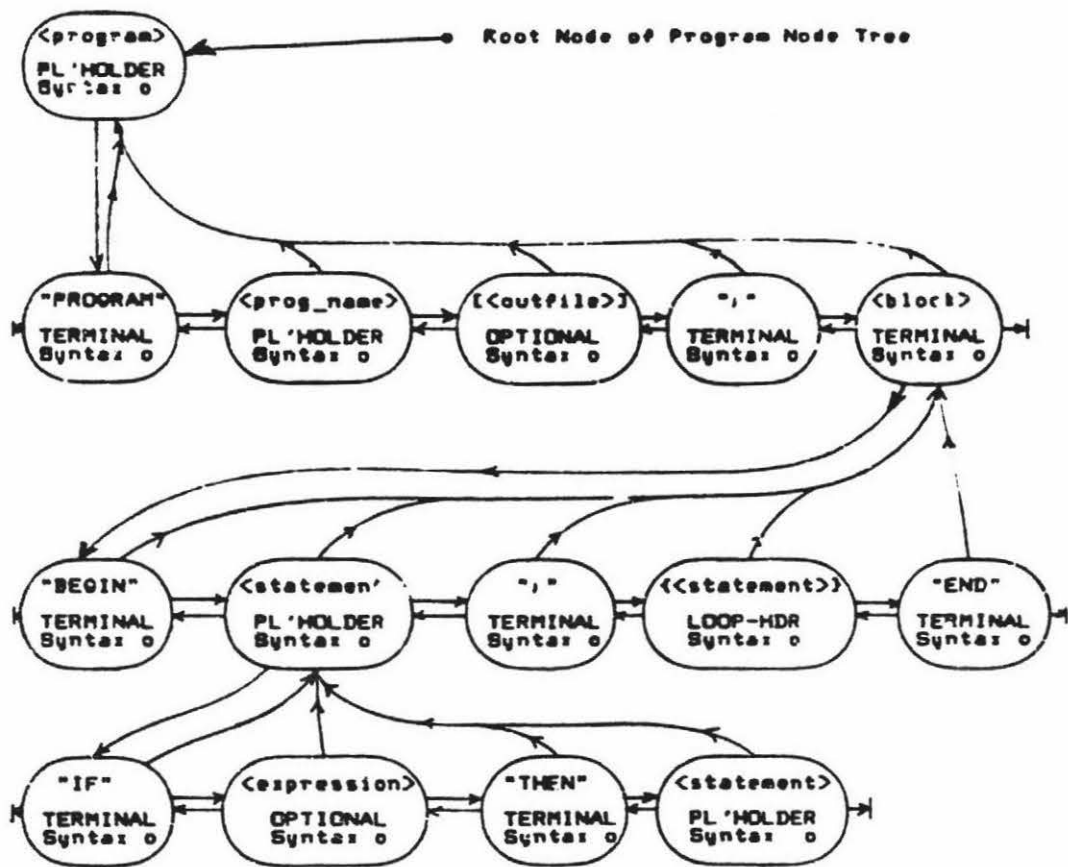


Fig 3.9 - The Expansion of statement to If statement



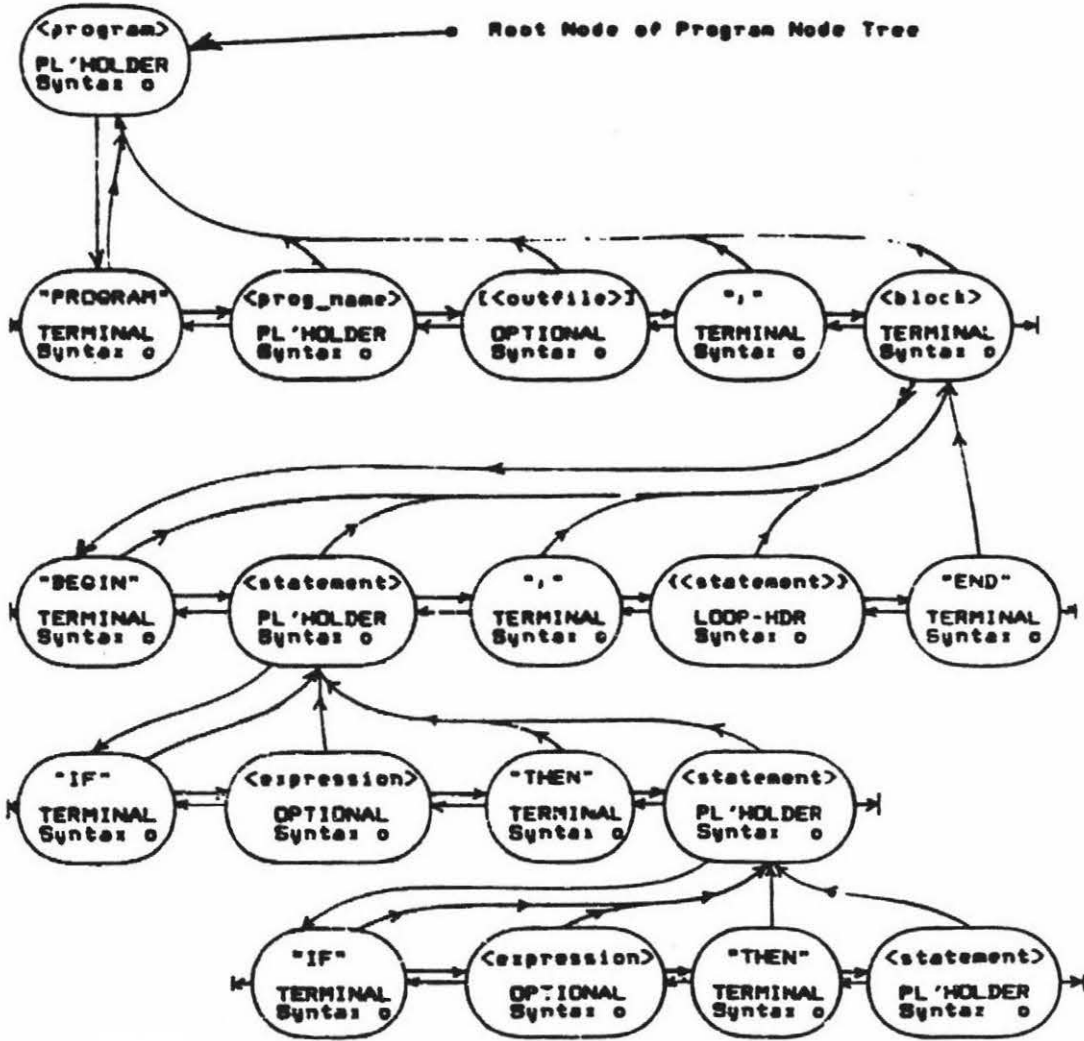


Fig 3.10 - The expansion of If statement to an If statement

### 3.13 The Program Node Field Definitions

#### The Program node PNODE TYPE field

This is a tag field indicating which variant of a program node is represented by this node. There are two major variants, "terminal symbol" and "placeholder".

For "terminal symbol" program nodes, the actual symbol represented is stored in the program node itself. Sufficient information is saved in the node to enable the symbol to be regenerated for display purposes.

A program node that has not been expanded is called a "placeholder". For "placeholders" program nodes, the definition of the node in terms of the syntax is indicated by the "definition" field.

The two other variants, "loop node" and "optional" program nodes indicate the "zero or more" ( $\{A\}$ ) and "zero or one" ( $[A]$ ) constructs respectively. This tag field is, strictly speaking, redundant. The same information could always be obtained by following the "definition" field pointer to the syntax every time the program node type is needed - which is often. For clarity during programming, and run-time efficiency this field has been included.

#### The Program node DEFINITION field

This field always points into the syntax definition. It indicates which syntactic production must be satisfied to completely expand the current node. For example, if a statement was necessary at a

particular point, this would be denoted by a program node with its definition field pointing to the header for <statement> in the syntax tree (figs 3.7, 3.8). As the definition field is always present, a printable representation is always available for all program nodes. For placeholders, the syntactic derivation is printed and for terminal symbols, the actual symbol.

#### The Program node EXPANSION field

This link is a pointer to the expanded (more detailed) definition of the syntax definition pointed at by the current node. This expansion is in terms of other program nodes. An example would be a node that pointed to the syntactic production for <statement>. If the expansion field was not null, it would point to the possible expansions of <statement> in terms of terminal symbols and placeholders. One expansion could be an IF statement (fig 3.9). If however, the expansion pointer is currently nil, then no more detail is available about a particular derivation.

When a program is complete all the expansion fields, with two exceptions, will be non-nil. The exceptions are for terminal symbols and for optional productions (loop nodes and optional nodes). For terminal program nodes, the node contains the definition of the symbol it represents and therefore no further expansion is possible. In the case of optional productions ([A] or {A}), the program is complete without further expansion, and so the expansion field may be nil.

The Program node CONTRACTION field

This is a pointer to the ancestor of this program node. It is the opposite of the "expansion" link. Following the contractions links will eventually lead to the root node. The root node is the only program node that may have a null contraction pointer. The root program node will have its definition field pointing to the root node of the syntax definition - always the first production. This field enables the user to ascend to program tree and is used to encompass sections of the program in preparation for a delete command. This will be explained later.

The Program node NEXT field

This is the pointer to the nodes at the same logical level. The "next" field provides the links necessary to indicate sequential productions as in "PROGRAM <program-name> ; ". The last node in a list has a next field of NIL.

The Program node PREVIOUS field

This is the opposite of the "next field". The first program node in a list has a "previous" field of nil. As will be explained later, this field is used to repair pointers when performing an UNDO operation.

### Suspending a Parse

The existence of the program node tree permits the parse to be suspended or resumed at any stage, as the current state of the parse is stored explicitly in the program nodes themselves. This is illustrated in figs 3.9 & 3.10, where the placeholder for <statement> has been expanded before those for <program name> and [<output file>]. The need for an expansion of <program name> is indicated by the presence of the "placeholder" program node with a null expansion field. Note that the syntactic definition is available through the pointer to <program name> in the syntax. As the syntax for <program name> cannot derive EMPTY, an expansion is required. However, this does not apply to the parent node of the <output file> as this node may derive EMPTY and therefore need not be expanded.

### 3.14 Automatic Inclusion of Necessary Terminal Symbols

The editor will automatically include all non-optional terminal symbols. This is illustrated by the inclusion of the END to match BEGIN, and a THEN when the IF of an if statement is entered. There are however, many other symbols that must be present, some examples from Pascal being the colon in a type definition, and the dot at the end of a program. These are also included.

The location of terminal symbols for automatic inclusion is aided by the parallel nature of the program and syntax node trees. If a program node has as its definition a syntax node which defines a terminal symbol and has an "alternative field" of nil, then that symbol must be present in the final program and is provided without user intervention.

This automatic inclusion is necessary not only at the top level but must be applied recursively as any productions included may contain other required productions and terminal symbols. A single level of this is illustrated in fig 3.7 with the production "<block>", which is necessary. Therefore as the BEGIN and END that occur within <block> have no alternatives, they must be included also. This automatic inclusion is propagated as far as possible to include all non-optional terminal symbols in all non-optional productions.

The automatic inclusion of symbols stops when an choice of directions is indicated by the syntax (the "alternative" field is not nil). Further development of the program node being built is abandoned, but its definition field still points to the production with alternatives.

When the user indicates by entering a symbol which alternative is wanted, that alternative together with all necessary sub-productions and non-optional terminal symbols will be included.

An exception is made in the case of any placeholders that derive the terminal symbols "identifier", "number" and "string" as the actual symbol must be provided by the user. However, if the automatic inclusion of non-optional expansions is carried to the limit, the information provided by the upper level placeholders can be lost as placeholders are always reduced to "identifier", "number" or "string". This is illustrated in fig 3.11, where the placeholder <prog\_name> has been expanded to "identifier".

This is syntactically correct, but from a user's point of view, it is

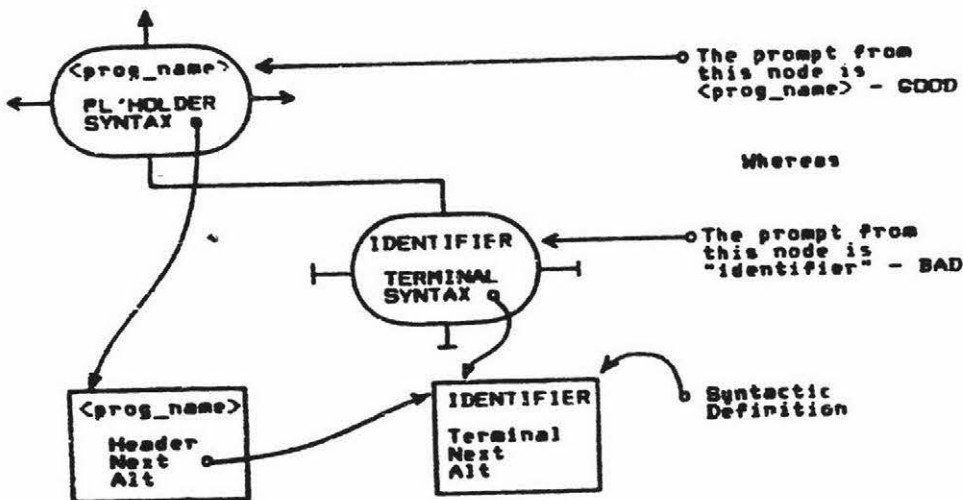


Fig 11 - Leaving <Prog name> Node Unexpanded Gives a Better Prompt

much more informative to have the placeholder "<prog\_name>" instead of "identifier". The "help" information will still show that an

identifier is a correct choice. In order to inhibit the development of these undesirable expansions, the editor checks its syntax before building the expansion of a subtree, to see if it eventually produces just one of "identifier", "number" or "string". If so, no further expansion is done. This leaves the upper level placeholder unexpanded and therefore its name is used as the prompt.



### 3.15 The Cursor - the Concept of a "Current Node"

At any stage, a single program node must be selected as the target for any alterations to the program tree by the user. The "cursor" is a pointer to that node and through its "definition" field to the syntactic definition of that node. This is the production that can be parsed if the user enters a symbol. The cursor position will be changed by one of two actions. The first is the entry of a correct input symbol, in which case the cursor will move to the next possible insertion point. The second is the entry of a cursor movement command.

### 3.16 Where does the Cursor Stop?

The cursor can, by various commands, be made to stop on all unexpanded placeholders and optional nodes, on all loop nodes and on specific user-entered symbols (otherwise they couldn't be changed). These are its primary stopping points. During the creation of subtrees (by expanding a placeholder, optional or loop node) the cursor will stop sequentially on each unexpanded placeholder, optional or loop node in the subtree. If the original node is a loop node, then once all the nodes in the subtree have been expanded (or skipped) the cursor will again stop on the loop node to permit another subtree. The cursor will then move to the next insertion point in the program, regardless of which subtree it is in. This order is illustrated for a small program in fig 3.12.

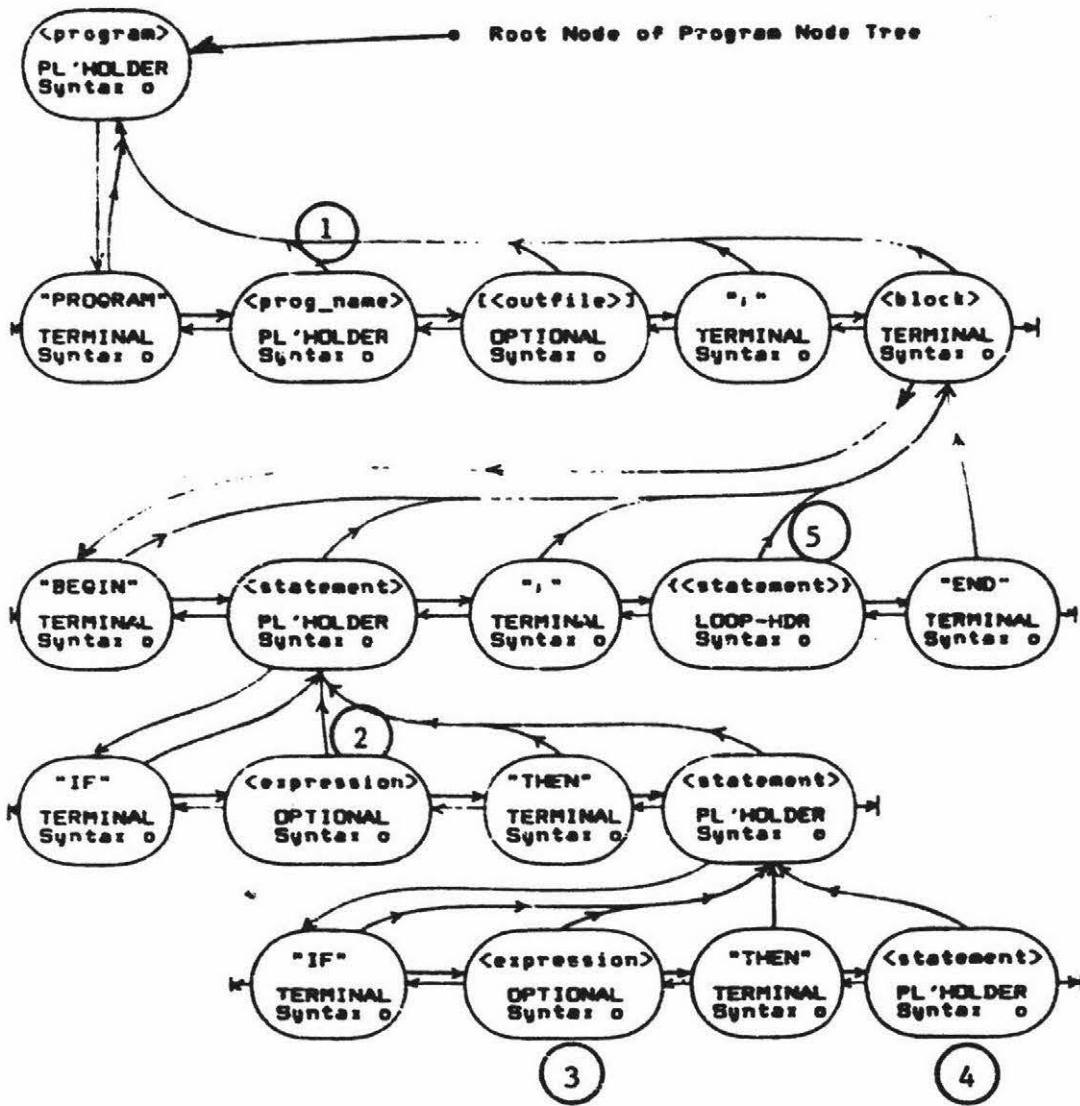


Fig 3.12 - Stopping Nodes from from Beginning to End

In addition, for purposes of deleting specific subtrees, the cursor may be made to ascend the tree (to encompass more and more of the program). It will stop only on program nodes that correspond to complete syntactic productions (i.e. the definition field points at a header node), optional nodes and loop nodes. This means that only subtrees corresponding to syntactic units may be clipped or deleted. The stopping nodes while ascending the program are shown in fig 3.13. The "ascend" command (Up-arrow) only alters the cursor position. It is non-destructive as distinct from the "delete" command which removes the subtree below the current node.

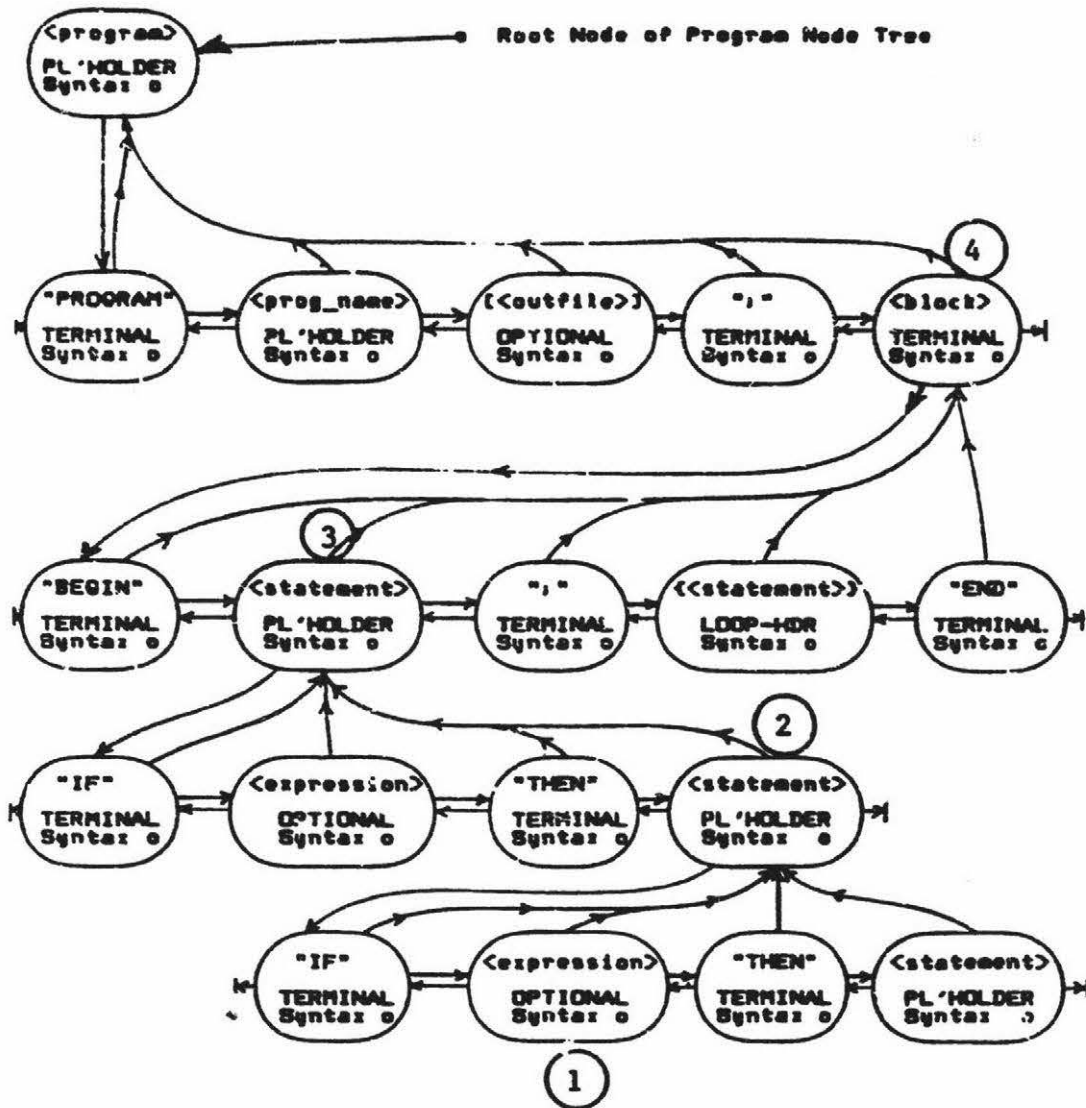


Fig 3.13 - Stopping Nodes while Ascending Program Tree

### 3.17 The Inclusion of User Symbols into the Program Tree

Any symbol that may start the syntactic productions directly derivable from the definition field of the current program node is acceptable at the cursor position. For example, in fig 3.7, the <statement> placeholder has as alternatives either the assignment statement or the IF statement. Therefore the only acceptable symbols are either an IDENTIFIER or an IF. The entry of any symbol by the user will cause the editor to attempt to find a match among the start symbols of all the alternative productions derivable from the current node. One of three things can now happen, depending on whether or not the symbol is in the current node's start set and if not, whether the current node can derive the empty production.

#### The Symbol is Not in the Start Set of the Current Node

If the input symbol is not the start symbols for the any of the productions and no production can derive the empty symbol, then the symbol is incorrect in the current context. The user is notified of an error.

If the production may derive the empty symbol, then following program nodes are checked to see if the input symbol is in their start sets. If not, then the symbol is incorrect (at the current position) and the user is notified. If a program node with the symbol in its start set is found, it is treated as though it were the current node, and the symbol used to expand it. An example of this would be the entry of "IF" with the cursor on the optional node "[<output\_name>]" (fig 3.8). "IF" is a reserved word used in <statement> and is therefore not an

identifier and so not in the start set of <output\_name>. However, <output\_name> is optional and therefore following program nodes are examined. The reserved word IF may start a <statement> and so the placeholder for <statement> is expanded to "IF <expression> THEN <statement>". The effect is as if the cursor was on the <statement> node. If the expansion selected by the lookahead is as the user intended, all is well. However if not, the sudden change in the position of the cursor and the incorporation of an unexpected construct at an unexpected location can be confusing. Although the lookahead can produce unexpected results, it is useful, as it avoids the need to locate the specific node for a known input symbol accurately. Note that this lookahead will only skip over optional nodes - the occurrence of a required placeholder will cause the search to be abandoned. An unexpected expansion caused by the lookahead can be removed with the undo command, which will also restore the cursor to its previous position.

#### If a Symbol is in the Start Set of the Current Node

If the current node points at "identifier", "number" or "string" and the input symbol is one of these classes of symbols, then a terminal symbol program node is created and the actual symbol stored in it. The expansion field of the current node is changed to point to this new program node. These are the only pseudo-terminals that may be expanded to actual terminal symbols.

All other terminal symbols are used to guide the editor. The automatic inclusion of terminal symbols and subtrees (i.e. productions) can only proceed while no ambiguity exists regarding the possible next symbol.

In other words, until an alternative is encountered in the syntax. An input symbol provided by the user indicates which alternative is wanted and enables the editor to continue its construction of the program tree. There are two forms the alternatives can take, they can be either terminal symbols or pointers to other productions.

The first case, when the alternatives are terminal symbols, requires no special treatment. The entry of one of the correct terminal symbols will cause that symbol to be incorporated into the program. This is done by creating a new program node (of type "terminal symbol"), saving the new symbol in the node, and linking the new node into the program tree, as the expansion of the current node (fig 3.14).

The second case, that of non-terminal alternatives, is potentially much

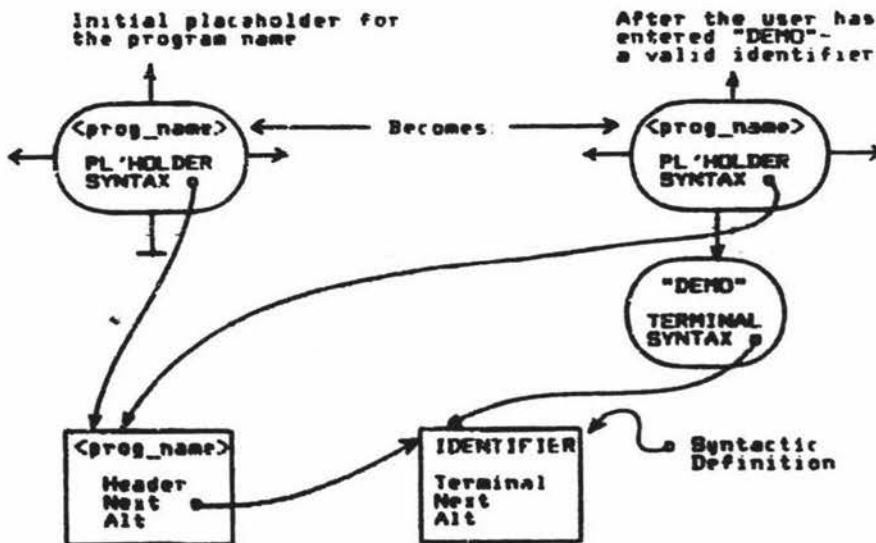


Fig 3.14 - The Incorporation of a Terminal Symbol into Tree

more complex. The complexity arises because a non-terminal production may point to other non-terminal productions to an arbitrary depth. Care must be taken in this case to avoid losing intermediate

**productions (see next section).**



### Building Program Nodes on Ascent

The problems with non-terminal alternatives arise because it is no longer sufficient to simply to identify the input symbol as being one of the valid symbols and to change the expansion field of the current node to point to a new program node incorporating this symbol. If this approach was adopted it is possible to skip some productions completely, as is illustrated by the omission of the intermediate production "<middle>" in fig 3.15. This is avoided by building the necessary program nodes at the lowest level (where input symbol matches) and then as the recursion unwinds, for any nodes whose "next" field is not nil, building a level of program nodes at this intermediate level. The lower level nodes are then linked in as the expansion field of the first node (fig 3.16).

Because all essential non-terminals are automatically included in the structure. The entry of a single keyword can cause the generation of multiple layers of program nodes.

This scheme does not lose productions, but does have side effects in an apparently unrelated section of the editor. As will be explained presently, it is possible to associate formatting commands with any syntax node. These commands are executed during the scan of the program node tree (via the derivation pointer of each node) in order to pretty-print the regenerated program. If only those syntax nodes with non-null "next" fields are included while building the program during ascent, any formatting commands associated with the omitted nodes will be ignored. For this reason, when tracing the ascent of the syntax

tree, a program node is created for all syntax nodes.

```

<top>      = <middle>  |  B
<middle>   = <bottom> MISSED
<bottom>   = F
    
```

Fig 15a - Grammar used to illustrate Shipped Productions

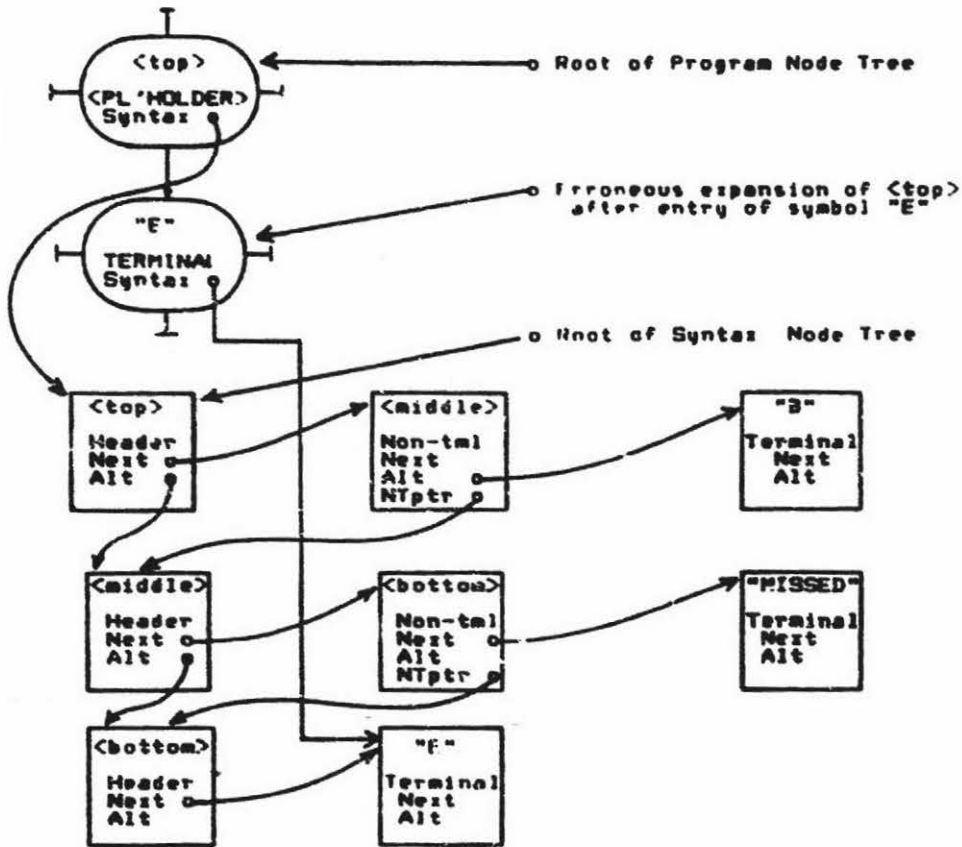


Fig 15b - Initial Data Structure for Syntax of Fig 15a

<top> = <middle> | B  
 <middle> = <bottom> MISSED  
 <bottom> = E

Fig 16a - Grammar used to Illustrate Shipped Productions

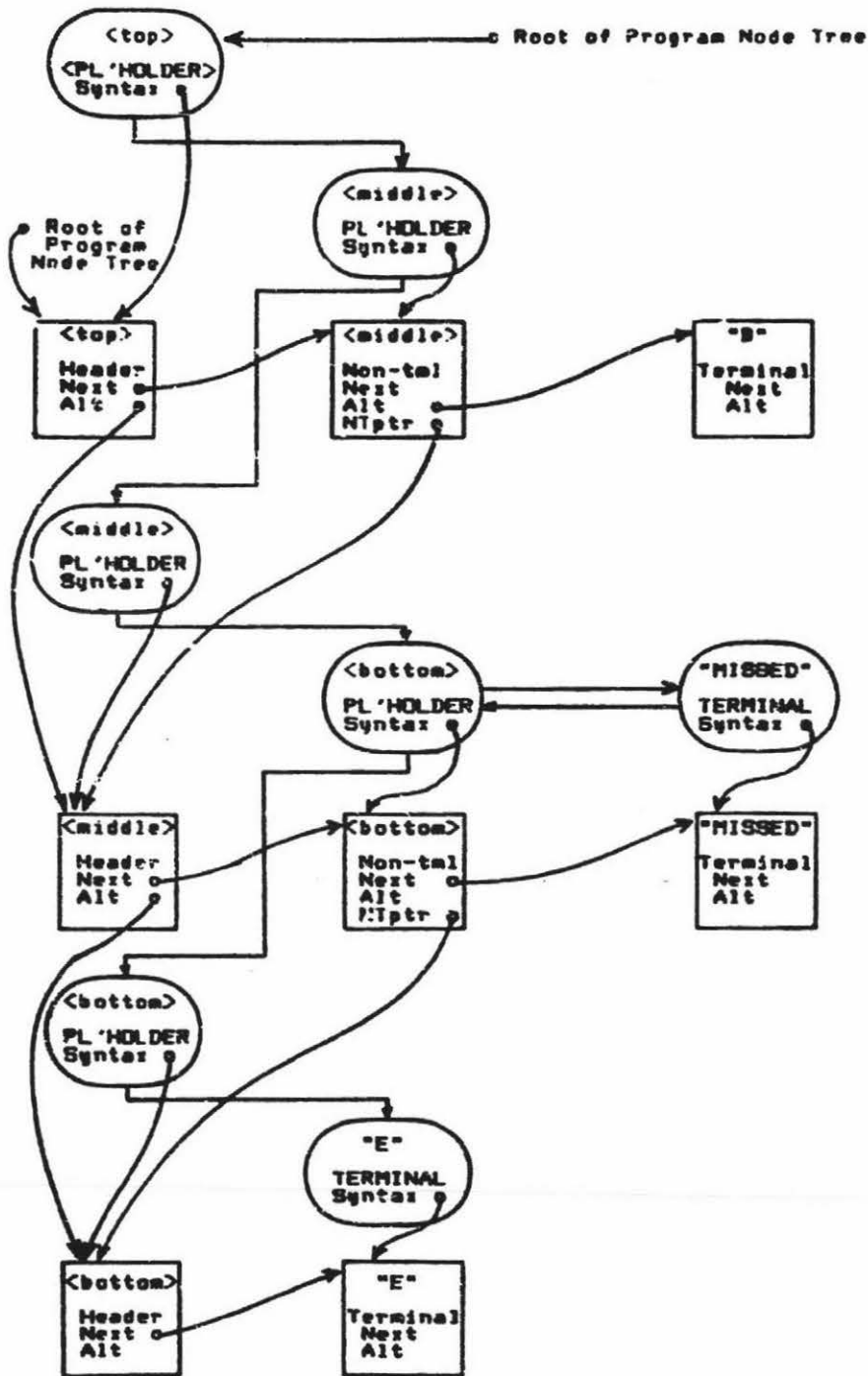


Fig 16b - Correct Expansion of <top> after Entry of "E"

3.18 The Structure Created by the Expansion of Loop Nodes

The expansion of a loop node must be treated specially as, unlike all other program nodes, it may be expanded repeatedly. "A" is a valid expansion of {A}, but the node may still be considered as unexpanded, as an indefinite number of "A"s are valid input symbols derivable directly from the placeholder for {A}. This is different from most placeholders which are initially unexpanded and the once expanded, are no longer considered when searching for unexpanded nodes.

The initial form of a loop program node is identical to a placeholder node - it has a null expansion field, and as usual, a pointer to its syntactic definition (fig 3.17).

If a valid input symbol is entered, then an expansion subtree will be

Syntax            <statements>        = { <statement> . }

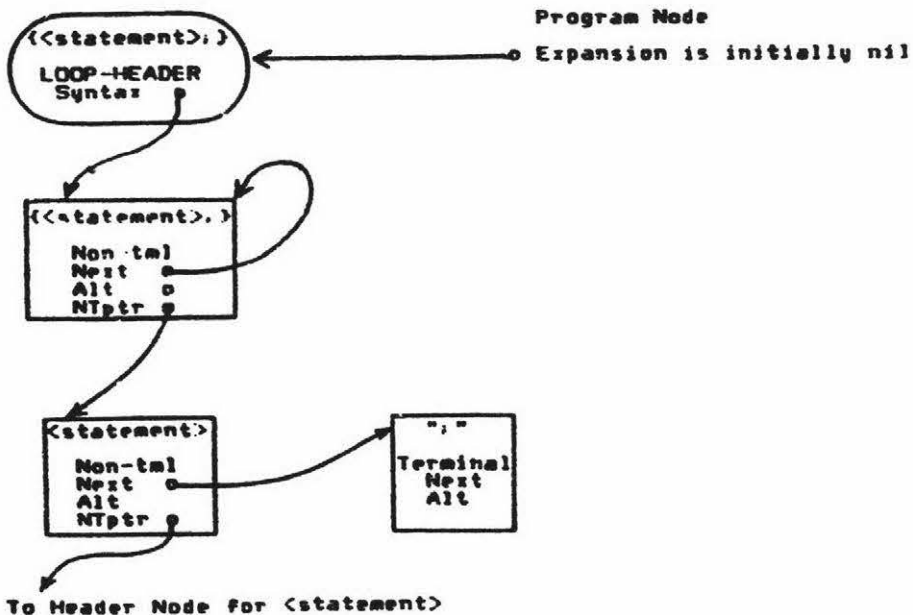


Fig 3.17 - Initial form of Loop Program Node

produced as with any placeholder program node. At that stage the

cursor could descend into the subtree to any possible modification points (i.e. any placeholder, loop or optional nodes). However, the loop node, while it has one expansion, is still a valid candidate for further expansion. The entry of the same symbol causes another instantiation of the subtree. The original loop node has already been expanded and therefore its expansion field is in use. Therefore, there is no attachment point for the newly created and any subsequent subtrees. In this case, a new instance of the loop node is created as the current node's neighbour (i.e. "next" of current node points at the new loop node) (fig 3.18). All further instances of the loop node subtree are handled in the same manner. This method has the desirable property that, by delinking the newly created loop node, it and its complete subtree may be removed from the program tree as a single unit - in the complementary manner to its creation. Note that the newly created loop node has the same ancestor as the original loop node - it is at the same logical level, as it should be.

Syntax <statements> ::= { <statement> ; }

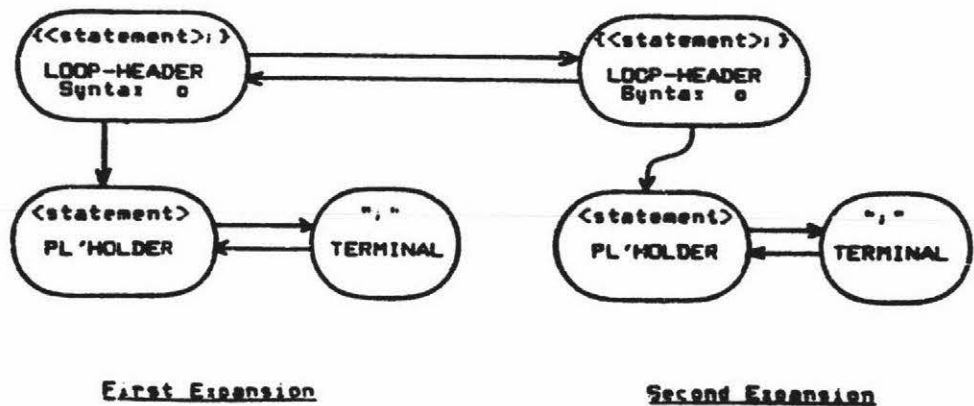


Fig 3.18 - Attach Point of Second Expansion of Loop Node

### 3.19 Unparsing the Program - Deriving a Display from the Program Tree

The current state of the user's program is stored in the program node tree and it is solely from this representation that a listing of the program is generated. No form of text representation is saved with the exception of the "spelling" of the user-defined terminal symbols. To regenerate the program, each program node is visited in turn, with the "expansion" field of a node unparsed recursively before that node's "next" field - a depth-first scan. However, not all the program nodes need have their names printed. Only the leaf nodes of the tree - terminal symbol and unexpanded placeholder nodes - should be printed. The print representation of a terminal program node is simply the terminal symbol that it represents. Unexpanded placeholder program nodes must also have a printable representation, but this can't be in terms of terminal symbols - there aren't any (yet). Instead, the name of the requisite non-terminal syntactic production is used. This is always available as every program node contains a pointer to its syntactic definition.

Unexpanded optional and loop nodes are only displayed when they lie within the subtree of the current node. To always show all the optional parts of a program is confusing - it clutters the screen with extraneous detail.

The display of a loop node is treated in a special fashion, as it may be expanded many times. If it has been expanded but does not lie under the cursor then only its subtree is printed, as with any other expanded placeholder or optional node. For example, if {<statement>} had been

expanded to "Z := 1", it would be displayed as "Z := 1". If however, the loop node is within the subtree of the current node (i.e. Under the cursor), then after unparsing its subtree, the name of that node is printed again, to indicate that another instantiation is possible. Therefore if the above loop node "{ <statement> }" was under the cursor, it would be displayed not as "Z := 1" but as "Z := 1 {<statement>}". This clearly indicates the possibility of another <statement>.

Note that there has been no mention of formatting the regenerated program in any way. If formatting of the regenerated program is required, the EBNF definition of the grammar is augmented with pretty-printing instructions, as the syntax itself contains no information about program layout.

### 3.20 Defining the Program Layout - A Table-Driven Pretty-Printer

In a syntax-directed editor that is intended to be language independent, it is essential that the user be able to define the screen layout of the resultant program. This information cannot, in general, be found from the syntax specification, as there are many different ways to format the same syntactic production. One possible method would be to derive the formatting information from the layout of the syntax specification. In other words, mimic the layout of the syntax when regenerating the program. However, this method has severe limitations, some of which are:

- 1) If a production starts in certain column, does this mean that it must always start in that column?
- 2) If a non-terminal name is longer than "n" characters, but the following syntax item must start in column "n", how is this handled?
- 3) Line skips in the syntax definition are ambiguous. Are they: to make it (the syntax definition) easier to read, to try and get a syntax item into its correct column, or to indicate a line-feed in the displayed program?



This is too restrictive for general use. To overcome these limitations, the use of explicit formatting commands is necessary. A study of current programming languages provides a guide as to which formatting commands should be provided. There are four major styles of program layout:

- 1) Fixed column oriented - as in FORTRAN and COBOL.
- 2) Semi-column oriented as in Snobol - the first column has special significance, usually for labels. The remainder of the line is free format. This format is also common in assemblers.
- 3) Free format but line-oriented as in BASIC.
- 4) Free format with a nested structure as in PASCAL, ALGOL, and PL/I.  
This format also clarifies the structure of Lisp programs.

The first two require absolute column-oriented commands. A production must be able to be placed in, or not placed in, a certain column. In Snobol and Fortran a statement label, if present, must start in column one. The remaining parts of Snobol statements may occur in any of the other columns. Therefore a "TAB column-no" command is required. The availability of a NEW LINE command is assumed.

The free-format line-oriented layout requires only that the adjacent productions be printed adjacent to one another - no new commands are required.

The fourth class, that of nested construct, free-format languages also require tabbing but relative to the indentation level of a previous construct, not a fixed column. These relative tabbing levels are referred to as the indentation level. To cater for these languages, INDENT and OUTDENT commands are provided. These increase and decrease the current indentation level by a fixed number of columns.

To control the layout of the regenerated program, GED allows a list of print formatting commands to be associated with any terminal symbol or non-terminal in the syntax. The formatting information is optional and if omitted, the program tree will be printed without any regard to the number of characters that will fit on one screen line. Consequently, print formatting commands should as least define which productions start on a new line.

### 3.21 GED Print Formatting Commands

The print formatting commands provided by GED are listed in table 1. The actual commands are those starting with an "@" character. The uppercase text is to provide a English word to associate with each one. The use of the English is of course possible, but as several commands are usually necessary for each terminal symbol or non-terminal, the formatting definitions would become long and unwieldy.

@?      PRINT-ME : Print the terminal symbol or non-terminal associated with this node. This is necessary as a separate command as there are occasions when the current node should be displayed only after the execution of other formatting commands (e.g. skipping to a new line). Therefore the obvious default action of always printing the current program name first (or last) is not always satisfactory.

@n  
~      NEWLINE : Skip to a new line. Whether or not the next character is printed in column one will depend on the current indentation level. If it is not zero the appropriate number of spaces will be skipped first.

@l      MARGIN : Set the indentation level to zero, to take effect on the next newline.

@>        INDENT    : Increase the indentation level, to take effect on the next line skip.

@<        UNIDENT   : Decrease the indentation level, to take effect on the next line skip. If the indentation level is already zero, this command is ignored.

@t num    TAB        : Tab to column "num". This command takes effect immediately, unlike "@l", "@<" and "@>". If the tab column is not between 1 and 132 an error message given.

Table 1 - GED Print Formatting Commands

3.22 The Method of Associating Formatting Commands with the Syntax

In order to associate print formatting information with each production it is necessary to augment the EBNF definition. One possibility was to interleave print formatting commands with the EBNF. The augmented EBNF definition necessary to lay out an IF statement is shown in fig 3.19. In the following examples, layout is used only for presentation and is ignored by GED. In human terms however, it clearly indicates the intended layout and therefore tends to reduce errors when deriving the formatting commands.

```

<if stmt> ::= IF @? <expression> @? THEN @?@>@n
                <statement>                @? @< @n
[ELSE                @? @> @n
                <statement>                @?@<@n
]

```

Fig 3.19 - Format Information for an IF Statement

This looks somewhat cryptic, but is read as follows:

For IF - Print itself (i.e. "IF").

For <expression> - If the expression has been expanded then print its expansion, otherwise print the non-terminal name (i.e. <expression>). The line so far would consist of "IF <expression>" or (simply as an example) "IF value>15 + x".

For THEN - Print itself, increase the indentation level (to take effect on the next line) and then skip to a new line.

For <statement> - If <statement> has been expanded then print its expansion. If not, then print its name (i.e. "<statement>"), decrease the indentation level and skip to a new line. This will leave the indentation

level the same as when the IF was encountered. Therefore as long as "[ELSE <statement>]" maintains the current indentation level, all following constructs will be correctly indented.

For ELSE - If the ELSE option had been requested, then "ELSE" is printed at the current indentation level (i.e. in alignment with the IF). Then the indentation level is increased (to indent the following statement) and a new line started.

For <statement> - Print "<statement>" or its expansion, decrease the indentation level and skip to a new line.

Notice that any nested constructs, such as <statement>, must preserve the current indentation level. In the above example, if the "THEN <statement>" altered the indentation level, the ELSE and its following <statement> would be out of alignment with the IF. The print-formatting commands are executed in order to allow some actions to precede others, such as skipping to a new line before (or after) printing the derivation (or name) of the current node.

The method chosen to incorporate the formatting information is similar to that used in the ALOE system and consists of a print formatting definition that follows, and has the same structure as, the EBNF definition. This method (fig 3.20) is marginally more complex to use

than interleaving the formatting information with the syntactic definition (fig 3.19) as extra information must be included to indicate which formatting commands are to be associated with each syntactic item. It does however have the advantage that it doesn't alter the existing EBNF definition and doesn't introduce extraneous symbols that aren't part of the syntactic EBNF definition. If the formatting commands follow the syntactic definition, they may then be added after the grammar has been written without altering the existing definitions. Also the problem of distinguishing between symbols, syntactic meta-symbols and formatting meta-symbols no longer arises.

The keyword PRINTFORMAT terminates the current EBNF syntax definition and signals the start of the format definition. In order to associate the appropriate formatting commands with each terminal or non-terminal, the flag character "&" is used. It indicates the start of formatting information for the next syntactic item (terminal or non-terminal). The commands to display the IF statement using the trailing print formatting definition are shown in fig 3.20. (The reason for the digits after each ampersand will be explained shortly.)

```

<if statement> ::= IF <expression> THEN
                <statement>
                [ELSE
                <statement>
                ]
printf format  &1@?  &2@?  &3 @? @> @n
                &4 @? @< @n
                [&4 @? @> @n
                &6 @? @< @n
                ]

```

Fig 3.20 - The Use of a Trailing Print Format Definition

The formatting commands given in fig 3.20 are the same as those given in the interleaved example with the addition of "& <number>" and square brackets. The ampersand is necessary to associate the formatting commands with the different syntax items, but the number and the square brackets are redundant. Both are added to provide some consistency checks while adding the print format information to the previously constructed syntax tree. The number after the ampersand should always be the same as an internal counter which starts at one and is incremented whenever a new syntax item is started - on every "&", "[", "{" and "(" . A discrepancy signals an error. The syntactic meta-symbols of braces, parentheses, square brackets, and the alternation bar must all be present in the format definition and in the same relative position as in the syntax definition. An error here also causes an appropriate message.



The Default Print Format

The default action is to print the name or derivation of each node, for unexpanded and expanded nodes respectively. This is equivalent to "@" (i.e. PRINT-ME) being associated with every node. For a large percentage of the symbols in a grammar, and as a result most definitions do not have any formatting commands. Those productions that require specific formatting action must have a PRINTFORMAT definition, and must define the layout of every part of the current production. It is not possible to define half of a production and default the rest.

3.23 Generating the Screen Display

As mentioned previously, the current display of the user's program is found from a depth first scan of the program node tree. During this scan, the "print format" field of the syntax node associated with each program node is examined, to locate the formatting commands (if any) that should be executed before and after displaying the program node. The obscure phrase - displaying the program node - is necessary, as there are seven different types of program node which require different display formats (independent of the user defined formatting commands). These are listed in table 2.

<u>Type of Program node</u>		<u>Displays</u>
Terminal program node	prints as	The actual terminal symbol.
Expanded placeholder	prints as	Trace of the expansion subtree
Unexpanded placeholder	prints as	The name of its non-terminal

Expanded optional node	prints as	Trace of the expansion subtree
Unexpanded optional node	prints as	[ <name> or 1st symbol ] e.g. [ <filename> ] or [ELSE]
Unexpanded loop node	prints as	{ <name> or 1st symbol } e.g. { <statement> } or {,}
Expanded loop node	prints as	Trace of expansion subtree followed by { <name> or 1st symbol }

Table 2 - Print Formats of Different Program Nodes Types

The output from the display procedure could be displayed directly but this would involve rewriting the complete screen after most user input. Given that often the current and next screens are similar, some form of optimisation is possible.

### 3.24 Optimising the Rewriting of the Screen Display

The output from the program tree display procedure is all directed through a procedure that handles single character output. Any output destined for the screen is buffered into a circular buffer of lines until either: the region under the cursor occupies the entire screen; the cursor region is centered on the screen; or the program is exhausted. The characters are buffered in order to record the current image on the screen and so avoid rewriting any positions that have not

altered. The start and end of the highlighted cursor region are found by recording the position of the buffer pointer before and after unparsing the subtree of the current node. The display is normally in half intensity with only the region (subtree) under the cursor intensified. The current image on the screen (saved in an "old\_image" buffer) is then compared with that in the current buffer and only those characters that differ, or differ in intensity are redrawn. This is done in a straightforward manner using only the terminal commands of cursor x/y addressing, erasing the remainder of a line and erasing the screen. The ability of the terminal to display in two visually different modes (e.g. full and half intensity) is necessary to clearly delimit the extent of the subtree under the cursor. For terminals that support the operations of inserting and deleting both lines and characters, the redisplay algorithm described by Gosling [1981] would probably result in superior performance although the algorithm would have to be adapted to handle the use of dual intensities. The display routines, while not set up for a variety of terminals, have all the terminal control functions localised into a set of procedures (e.g. "erase\_screen", "position\_cursor(x,y)", "bright", "subdued") which are called when necessary. This clarifies the code and facilitates the adaption of GED to terminals other than the Visual 200 by localising the terminal dependencies. The ability to display half and full intensity is desirable, although normal and inverted video would suffice.

### 3.25 The Implementation of User Commands

### 3.26 Primary Cursor Movement Commands

The variants of the cursor movement commands are all handled by two tree-walking procedures (to handle forward and backward searches), and a boolean function to indicate when a valid stop node has been found. The use of a single function to test whether a program node satisfies the search conditions enables the use of only two tree-walking procedures (for forward and backward) to handle searches for many different types of program nodes. The function, "is\_stop\_node", has three boolean parameters: "searching", "stop\_on\_optional\_nodes" and "stop\_on\_user\_nodes" which are set up by whichever procedure calls the function.

The parameter "searching" indicates that the stop node must match the current token and is the method of implementing the forward and reverse symbol searches. The parameter "stop\_on\_optional\_nodes" will cause the procedure to flag all unexpanded optional, and all loop nodes, whereas "stop\_on\_user\_nodes" will only stop on symbols entered by the user. Unless a specific token is being searched for (i.e. searching = true), any unexpanded placeholders are treated as stop nodes by default.

For example, the forward search for the next modification point, (The "->" key) has both "stop\_on\_user\_nodes:" and "stop\_on\_optional\_nodes" set to true. Therefore the cursor will stop on any user entered nodes, any optional nodes, and by default, any unexpanded placeholders.

### The Use of a Default Argument for the Symbol Search Functions

If either the "repetition" command is given, or an empty line is entered as the reply to the prompt for the search symbol, the last symbol searched for in either a forward or a reverse search is used. In this case the actual symbol being searched for is displayed after the query as a confirmation that the symbol is as the user remembered. If the symbol cannot be found, the prompt message is overlaid with "NOT FOUND ---> ", leaving the symbol itself intact.

Therefore "Forward Search for : hello"

becomes "NOT FOUND ----> : hello"

This, like all status line messages, disappears when the next key is pressed.

### 3.27 Reading and Writing the Program and Clipped Subtrees to Disk

To save the program on disk as a listing file is straightforward - the routine that handles all single character output redirects it to a disk file. The list file will be an exact duplicate of the non-optional items in the user program as seen on the screen. All required placeholders remain but any currently visible optional placeholders are totally suppressed, as they are not required in a complete program.

To save the current state of the program node tree in a format suitable for recreating the tree is more awkward. Ideally a memory image would be saved. This would enable the exact state of the editor in its current state to be preserved. However, as Pascal does not provide any

way to save arbitrary data structures in a file, there can be no standard way of saving the current state of a program in this fashion. Therefore, the writing and subsequent reading of the data structure must be handled explicitly.

It would be possible to write out a trace of the program node tree, referring to each node by its id-number (which is unique) and to which nodes each of its fields pointed. This approach could be made to work but is very complex as it entails preserving the complete tree together with all the program node tag fields and for terminal nodes, the terminal symbol stored in the node. The user would have to ensure that the syntax did not alter between one run and the next, as the program nodes refer to the syntax tree. Either the names of the productions or syntax node id-numbers would have to be preserved, depending on how the program nodes referred to the syntax (by name or number). These limitations are too severe to be acceptable.

The use of the regenerated program text is another method of saving the program, or any subtree. Unfortunately, the list format is not suitable as input for GED as it contains many redundant symbols - those automatically included by the editor - and possible placeholders, which may be in some contexts indistinguishable from the user program. Consider a language that had a construct to starting with a left angle bracket followed by an identifier - for example the array specification in Snobol - would this represent a placeholder or two user-entered symbols?

These problems are only a distraction - given that all placeholders were originally inserted by the editor, why save them? Only those symbols originally entered by the user need be saved, together with sufficient information to ensure that they are used to expand the correct subtrees. Any subtree (including the complete program) may be saved by writing a text file consisting of only those symbols entered by the user, any necessary commands to skip over optional and loop placeholders. This format is concise and easily read back into the editor - the input stream is taken from disk instead of from the keyboard. An example of the two files written by the "Save Program" command for a small program are shown in figs 3.21 & 3.22. Fig 3.21 is the program listing file and fig 3.22 is the corresponding symbol file.

```

PROGRAM disk_io_demo ;
CONST
    line_length = 80 ;
    backspace = '?' ;
TYPE
    line_type = ARRAY [ 1 .. 80 ] OF char ;
VAR
    line_buffer : line_type ;
PROCEDURE getline ( VAR length : integer ) ;
VAR
    count : integer ;
    ch : char ;

BEGIN
    count := 0 ;
    WHILE ( NOT EOLN ) AND ( count < line_length ) DO
        BEGIN
            read ( ch ) ;
            IF ch = backspace THEN
                BEGIN
                    IF count > 0 THEN
                        count := count - 1 ;
                    END
                ELSE
                    BEGIN
                        count := count + 1 ;
                        line_buffer [ count ] := ch ;
                    END ;
                END ;
        END ;
    END ;

BEGIN
END .

```

Fig.3.21 - A complete Pascal program as listed by GED

```

disk_io_demo -> -> CONST line_length 80 -> ; backspace '?' -> -> TYPE
line_type ARRAY 1 80 -> char -> -> VAR line_buffer -> line_type -> ->
PROCEDURE getline ( VAR length -> integer -> -> -> -> VAR count ->
integer -> ; ch -> char -> -> -> count -> := 0 -> WHILE ( NOT EOLN ->
-> AND ( count -> < line_length -> -> -> BEGIN read -> ch -> -> IF ch
-> = backspace -> -> BEGIN IF count -> > 0 -> count -> := count -> - 1
-> -> -> ELSE BEGIN count -> := count -> + 1 -> line_buffer [ count ->
-> -> -> := ch -> -> -> -> -> ->

```

Fig 3.22 - The Ged code file Corresponding to Program of Fig 3.21



Note - Apart from replacing the nonprintable escape sequence for moving the cursor with "->" (for display purposes), these listings are as output by GED.

The symbol file also has the advantage that it is readable for diagnostic purposes. With this mode of program I/O, the syntax checks are implicit. The program or subtree, when read back into the editor is subjected to all the usual checks on user input. If the file contains any errors then the user is notified of the erroneous symbol and the insertion of the file is abandoned. This is consistent with inserting only complete and correct syntactic constructs; however the checking is implicitly done while reading the program - there is no header on the file indicating the type of production to follow.

### 3.28 The Clip/Delete and Insert Commands

Given that the program node tree is strictly hierarchical and that all program constructs are represented as subtrees, any construct can be removed in its entirety by deleting the pointer to it from the node above - that is, deleting the expansion pointer of its parent node. This is perfectly satisfactory as the action of a "delete" command, but some method of "clipping" a subtree and moving it elsewhere is also desirable. If the "deleted" subtree is saved in some form, it can form the basis of a composite "move" command (i.e. delete, move cursor and insert).

To insert this deleted subtree elsewhere, the expansion field of another program node is altered to point the delinked subtree and the contraction pointer of the subtree adjusted to point to its new parent. Note that if the only nodes whose subtrees may be deleted are restricted to those whose syntactic definitions are complete productions and the expansions of optional and loop nodes, this will restrict deletions and insertions to complete syntactic constructs. To retain the syntactic integrity of the program, the attach point must have the same syntactic derivation as the clipped subtree. Therefore, a statement that had been clipped out at the level of "<structured\_statement>" would not be an immediately acceptable expansion for "<statement>" - their non-terminal derivations are different. A search of the definition for <statement> would be needed to establish this equivalence. Often a section of code is not to be moved but copied. The direct manipulation of the tree in this manner will work for moving a single subtree, but not if the subtree is to be replicated. In that case, in order to prevent unexpected side effects if the subtree is altered, or even worse, moved, any replication of the subtree must cause a new copy to be created.

The method of saving the program to disk, by writing a compressed symbol file, can be considered a special case of saving an arbitrary subtree. It can therefore be used to save deleted subtrees also. In order to insert these subtrees elsewhere, the input stream can be taken from the file, as it done when reading a complete program. No problems exist with replicating the subtree - it is, as if the user had re-entered the same symbols as were used to create the original

subtree. Also no problems exist with the syntactic equivalence - the acceptability of each symbol is checked individually, rather than checking the syntactic equivalence of the complete structure.

When deleting a loop node from a list, the previous and next nodes are altered to skip the deleted loop node. This has the visible effect of closing up the elements of list, to eliminate the deleted subtree. (fig 3.23)

As the deleted symbol sequences are not wanted after the completion of

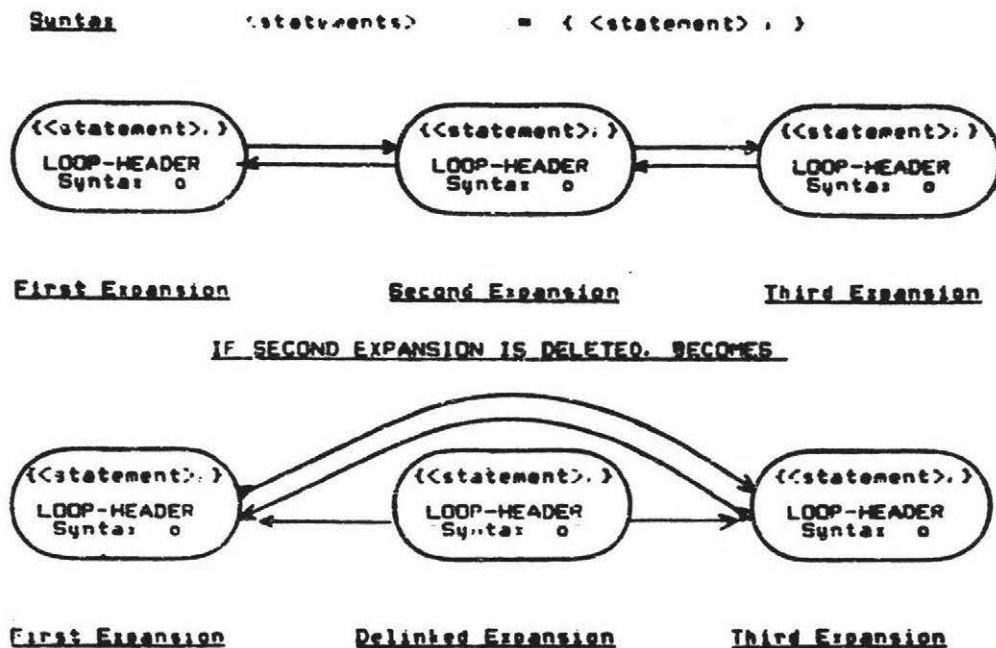


Fig 3.23 - The Effect of Deleting One in a List of Loop Nodes

the editing session, these could be saved in memory rather than on disk. However, as the same procedure will write both program and subtrees, the speed of disk I/O is satisfactory, and there are effectively no size constraints or memory management problems with the disk based system, the use of a memory based mechanism for preserving

subtrees has not been attempted.

### Ascending the Program tree to Locate Deletion Subnodes

In order to clip/delete sections of program larger than individual symbols, it is essential to be able to ascend the program tree until the required section of the program is under the cursor. On the screen, this subtree will be highlighted to clearly delimit the extent of cursor. When ascending the program tree, the cursor must not stop at every program node. Many apparently redundant nodes are created by the editor to keep in alignment with the syntactic data structure. These are normally transparent to the user, and should remain so. Therefore, when using the "ascend" command the cursor will only stop on program nodes that correspond to complete syntactic productions (i.e. the definition field points to a "header" syntax node), on optional and on loop nodes.

### 3.29 Marking, and Moving to, Specific Nodes in the Program Node Tree

To enable rapid cursor movement to specific user-defined nodes in the tree, a list of markers is provided. The markers are referred to by a letter and simply associate the current position of the cursor with a letter. The case distinctions are ignored. If a command is given to move to an uninitialised marker, an error message is given.

The markers are most useful for skipping over large sections of the program, enabling actions such as returning to the type or variable declarations in one command. The "move to marker" command swaps the value of the current node and that associated with the letter. If the

end of the declarations have been marked, the first move command will move the cursor to the current end of the declaration list. A declaration may then be added and the move command given again, returning the cursor to its original position and updating the marker position to the new end of the list. This simplifies adding elements to a list from arbitrary points in the program.

### 3.30 The Implementation of the "Undo" Command

The undo command is implemented by keeping a stack of the position of the cursor before each command was executed, and which node, if any, was altered during its execution. The number of commands that can be undone is limited by the size of undo stack which is currently set at thirty. Some commands, such as "delete" when removing elements from a list, alter not one but two nodes. However, this is not a problem as each program node is doubly linked to its contraction, its expansion, and each of its neighbours. Therefore, knowing the links in one direction is sufficient to enable the others to be re-established.

## Chapter 4

### Language Implementation Considerations

#### 4 The Implementation of Syntax-Editors for New Languages using GED

The implementation of a syntax-directing editing environment using GED is a sizable project, even if the language grammar is available in an extended BNF format. This chapter will discuss the problems which were encountered in building editors for Pascal, Snobol and Lisp. These grammars cover a wide variety of programming styles and types of languages, and are sufficiently different to indicate the strengths and weaknesses in the design of the editor.

There are two distinct stages in the implementation of an editor for a new language. The first is the preparation of the language syntax in EBNF and the second is adding the print formatting information to the syntax. Although both of these are well defined, in practice the syntactic definition is usually modified iteratively to provide the most useful placeholder prompts and a pleasing layout. The modifications made to the grammars of several languages and the addition of formatting information will be discussed in turn.

#### 4.1 Preparing the Extended BNF Grammar

The syntax of the programming language must be definable in extended BNF and must conform to the requirements of an LLI grammar (i.e. in each production there may be no left recursion and no replicated start symbols). A grammar that is free of left recursion but does not conform to the second condition will be accepted by GED, but some productions will never be parsed. For example, in the production " $\langle Z \rangle ::= A \mid A B$ ", the input symbol "A" will always match the first alternative and therefore "A B" will never be parsed. The reason for this is evident from the manner in which GED searches the syntax tree - the first production that matches the input symbol is the one parsed. The removal of multiple start symbols from a grammar can be accomplished by factoring the productions (e.g  $\langle Z \rangle ::= A [B]$ ).

The actual definitions of programming languages are rarely LLI as is illustrated by the excerpt from the Pascal grammar in fig 4.1 (taken from the "Pascal User Manual and Report" [Jensen 1974]), and later in the grammar of Snobol.

```

<simple statement>      ::= <assignment statement>      |
                        <procedure statement>          |
                        <go to statement>              |
                        <empty>                          |

<assignment statement> ::= <variable> := <expression>   |
                        <function identifier> := <expression>

```

```
<procedure statement> ::= <procedure identifier> |  
                           <procedure identifier>  
                           (<actual parameter> {,<actual parameter>})
```

Fig 4.1 - Standard Grammar for Pascal is Not LLI

The problems with the grammar in fig 4.1 are these: at the level of <simple statement>, both <assignment statement> and <procedure statement> produce <identifier>. Within <assignment statement> and <procedure statement> both alternatives also produce <identifier>. Therefore if a parser was at the <simple statement> node and the input symbol was an identifier, it has no way of determining which alternative to parse. The input symbol would obviously be correct, but as part of which production? The grammar must be rewritten to remove the ambiguity. If the information obtained from the variable and procedure declarations was available (from a symbol table), no ambiguity would exist. The symbol table would indicate whether the identifier was a variable, a function name, or a procedure that had, or did not have, parameters. GED is designed to work solely from the syntactic definition and therefore has no symbol table. One of the aims of this thesis was to investigate the viability of this approach.



#### 4.2 A Case Study - The Implementation of a Snobol Editor

The starting point for the implementation of a syntax editor for a new language is its syntax definition. The grammar used here is from "The SNOBOL4 Programming Language" [Griswold 1971]. The notation used is similar to EBNF, but differs in the way of defining optional productions and lists of productions. The grammar shown in fig 2 is a transliteration into EBNF, but is otherwise unaltered. This is only the preliminary step as the grammar is still not in a form that is suitable for input to GED.

Unmodified Definition of Snobol in Extended BNF

```

<digit> ::= 1|2|3|4|5|6|7|8|9
<letter> ::= 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
<alphanumeric> ::= <letter> | <digit>
<identifier> ::= <letter> { <alphanumeric> | . | _ }
<blanks> ::= one or more blank characters
<integer> ::= <digit> { <digit> }
<real> ::= <integer> . [ <integer> ]
<operator> ::= ^ | ? | $ | . | ! |
              + | - | @ | " | &
<unary> ::= <operator>
<string> ::= one or more EBCDIC characters
<sliteral> ::= ' <string> '
<dliteral> ::= " <string> "
<literal> ::= <sliteral> | <dliteral> | integer | real
<element> ::= {<unary>} ( <identifier> |
                       <literal> |
                       <function.call> |
                       <reference> |
                       ( <expression> )
                       )
<operation> ::= <element> <binary> (<element> | <expression>)
<expression> ::= [<blanks>] [<element> | <operation>] [<blanks>]
<arg.list> ::= <expression> { , <expression> }
<function.call> ::= <identifier> "(" <arg.list> ")"
<reference> ::= <identifier> "<" <arg.list> ">"
<label> ::= <alphanumeric> <string>
<subject.field> ::= <blanks> <element>
<pattern.field> ::= <blanks> <expression>
<object.field> ::= <blanks> <expression>

```

```

<equal> ::= <blanks> =
<goto> ::= "(" <expression> ")" | "<" <expression> ">"
<goto.field> ::= <blanks> : [<blanks>]
                ( goto
                  |
                  S <goto> <blanks> [F <goto>] |
                  F <goto> <blanks> [S <goto>]
                )
<eol> ::= END-OF-LINE
<eos> ::= [<blanks>] ( ; | <eol> )
<assign.stmt> ::= [<label>] <subject.field> <equal>
                 [<object.field>] [<goto.field>] <eos>
<match.stmt> ::= [<label>] <subject.field> <pattern.field>
                 [<goto.field>] <eos>
<repl.stmt> ::= [<label>] <subject.field> <pattern.field> <equal>
                 [<object.field>] [<goto.field>] <eos>
<degen.stmt> ::= [<label>][<subject.field>][<goto.field>]<eos>
<end.stmt> ::= END [<blanks> [<label> | END ]] <eos>
<statement> ::= <assign.stmt> | <match.stmt> | <repl.stmt> |
                 <degen.stmt> | <end.stmt>
<comment.line> ::= * <string> END-OF-LINE

```

Fig 4.2 - Official Syntax of Snobol4 in EBNF

The grammar must be modified in the following ways:

- a) The delimiters "\$" and "\$\$" must be appended to each production and the last production respectively.
- b) The definition for identifiers must be converted to start and continue sets, because otherwise the editor would assume that the individual characters were separate tokens and would separate them with spaces.
- c) Strings must be defined in terms of their delimiters.
- d) GED requires that comments be defined in terms of either their start character or their delimiters. Snobol uses an asterisk in column one as a comment flag, and therefore both the comment start character and the comment column must be defined.
- e) The grammar contains a specific symbol for an End-of-line condition, a symbol unknown to EBNF.
- f) The grammar has a specific representation for blanks - a lexical item unknown to the editor.
- g) Several productions in the grammar contain alternatives that begin with the same symbol. The grammar must be factored to remove the multiple start symbols. An obvious example is <statement> in

which all the options start with [`<label>`], but a more subtle example is the production for `<element>`. It may start with either an `<identifier>`, a `<function.call>` or a `<reference>`, however both `<function.call>` and `<reference>` also produce `<identifier>`.

- h) Print formatting information must be added to nine of the total of 24 productions.



```

    printfmat      &l@s@?                                $
<goto>           ::= '( <expression> ' ) | '< <expression> '>    $
<goto.field>    ::= : ( <goto>                               |
                        S <goto> [F <goto>]                   |
                        F <goto> [S <goto>]                   |
                        )
% Tab to column 50 before printing the colon

    printfmat      &l@s@t50@? (&2@?                          |
                        &3@? &4@?@s [&5@? &6@?]              |
                        &7@? &8@?@s [&9@? &10@?]             |
                        )                                        $

% The end.of.line inserted by the pretty.printer forms the end
% of statement if ";" missing. ie instead of <eos> ::= ; | end_of_line

<eos>            ::= [ ; ]                                    $
<end.stmt>       ::= END [ [<label> | END ] ]
    printfmat     &l@s@? [ [ &2@s@? | &3@s@? ] ]              $
<statement>     ::= [<label>] <stmt, goto, or. end> <eos>
    printfmat     [&l@? ] &2@? &3@?@n                      $
<stmt, goto, or. end> ::= <stmt_body> | <goto.field> | <end.stmt> $
<stmt_body>     ::= <subject.field>
                    ( [ = [ <object.field> ] ] |
                      <pattern.field> [ = [ <object.field> ] ]
                    )
                    [<goto.field>]                          $
$$

```

Fig 4.3 - The Grammar of Snobol ready for input to GED

### 4.3 Areas of Alteration in the Snobol Grammar

With the exception of the productions delimiters ("\$" & "\$\$"), the addition of formatting information, and the removal of multiple start symbols, all the modifications to the grammar result from Snobol's incorporation of complete lexical information into the syntax. Examples of these occur in the definition of <identifier>, <string> and

<blank>, and format-sensitive constructs. These constructs include the use of the end-of-line character, and the requirement that certain constructs, such as comments and labels, must start in the first column.

#### 4.4 Are Identifiers, Numbers, Strings and Comments Productions?

Syntactically these items can be defined as any other production in the EBNF syntax, and could therefore be handled in the same fashion. In traditional compilers, for reasons of efficiency, this is not done. Instead, a lexical analyser (a scanner) is used to collect the input stream into the basic symbols of the language before any parsing is done. GED uses a scanner, not for efficiency, but for the pragmatic reasons discussed below. In the following discussion, the identifier is used as an example, although the comments apply equally to numbers, strings, and comments.

The editor should provide for convenient entry of identifiers as single entities in the usual fashion. That is, no prompting should be needed while entering an identifier. If however, the production <identifier> was implemented using the same technique as the remainder of the syntactic definition (i.e. as <letter> {<alphanumeric>|.|\_}), the optional part "{<alphanumeric>|.|\_}" would reappear every time the cursor moved past the identifier. This is unnatural in use as identifiers, numbers and strings are usually treated as composite (multi-character) items only at the time of their initial entry. They are not subject to incremental modification (i.e. the addition of new characters) at some later time. This can be contrasted with



"<statement> {<statement>}", in which the later addition of extra statements is possible. Notice that the boundary between the lexical and syntactic constructs must be determined by the writer of the grammar - it is not evident from the syntax itself.

#### 4.5 Hiding Optional Placeholders

The above example highlights one point that must be considered by the writer of a new input grammar for GED - it is essential to minimize the number and appearance of optional placeholders. If this is not done, many unexpected placeholders appear, which is disconcerting to the user and makes movement around the program clumsy. Optional placeholders represent optional productions in the syntax and so they cannot be removed - this would alter the language. They can only be hidden.

There are two methods of hiding optional placeholders. The first is to use the fact that GED displays only the first option in a list of alternatives as a prompt. For example, every statement in Pascal may be prefixed by a label. Therefore, the definition of <statement> could be written as "[<label>] <statement>". However, this would cause the "[<label>]" prompt to appear before every <statement> prompt, when in practice it is rarely used. By rewriting the grammar as "<statement> | <label> <statement>", this is avoided. The first option, "<statement>" is used as the prompt, but the syntax help display still shows the the complete production (including the optional label), and the start symbols include that for <label>.

Alternatively, productions related to program format (such as the "<blanks>" production in Snobol) may be omitted and their function taken over by the print formatter. The pretty-printer is used to insert a space when one is required.

It may be argued that this is modifying the syntax, but EBNF cannot represent blanks anyway - their description is in English. More importantly, if the productions regarding optional blanks were not removed from the syntax, the placeholder [<blanks>] would reappear sufficiently often to become annoying. Also, the required placeholder <blanks> would have to be represented as "<blank> {<blank>}". This would be another source of irritation as the cursor stopped on {<blank>} each time each time it was encountered.

It is not possible to state categorically that optional placeholders should, or should not, be displayed. A subjective decision on the part of the language implementor is necessary in order to determine their relevance, and only those judged relevant should be displayed.

#### Definition of Snobol Identifiers

The syntax of identifiers is defined in GED by enumerating the characters that may start and continue an identifier. For example, in the definition of fig 4.2, an identifier must start with a letter and may have any number of following letters, digits, dots and underscores. The definition of these sets in a form compatible with GED is illustrated in fig 4.4.

```

IDENTIFIER_START_SET &abcdefghijklmnopq:rstuvwxyz
                    ABCDEFGHIJKLMNOPQ:STUVWXYZ

IDENTIFIER_BODY     abcdefghijklmnopqrstuvwxyz
                    ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789._

```

Fig 4.4 - The Lexical Specification of Snobol Identifiers

Note - An ampersand has been added to the identifier start set in order to allow it to be used at the start of reserved words.

#### Defining the Strings in Snobol

The original grammar defines strings as arbitrary sequences of characters surrounded by either single or double quotes, whereas GED defines strings in terms of their delimiting characters. Therefore the definition which was:

```

<literal>      ::= <sliteral> | <dliteral>
<sliteral>     ::= ' <string> '
<dliteral>     ::= " <string> "
<string>       ::= Any sequence of characters

```

becomes:

```

STRING_DELIMITER " '
<literal>       ::= STRING

```

#### 4.6 Removing the Production for <BLANKS> from the Snobol Syntax

The Snobol syntax uses productions to specify strings of blanks. These productions are used either as separators (one or more blanks) or for formatting purposes (zero or more). GED uses a pretty-printer to format regenerated programs and therefore the use of blanks for formatting is redundant. Therefore, the production "[<blanks>]" can be omitted. However, single blanks are still needed as separators. These could be provided by retaining the production <blanks> which is defined as "one or more blanks" but this would require the definition of a new lexical item BLANK. This has not been done. The production "<blanks>" is also omitted and the print-formatter is used to insert a space where one is required. This is illustrated below.

```

<blanks>          ::= one or more blank characters
<subject.field>   ::= <blanks> <element>

                BECOMES

<subject.field>   ::= <element>

printfomat        &l @s @?          Print space (@s) before
                                   printing <element> (@?)

```

#### 4.7 Rewriting the Productions to Remove Common Start Symbols

Productions must often be factored to remove common start symbols. The production <statement> in Snobol is an example of this. Every type of statement may start with a label and have a <goto> field. Also most variants have a subject field. The syntax of <statement> (with <blanks> removed) is reproduced below.

```

<assign.stmt> ::= [<label>] <subject.field> <equal>
                [<object.field>] [<goto.field>] <eos>

<match.stmt>  ::= [<label>] <subject.field> <pattern.field>
                [<goto.field>] <eos>

<repl.stmt>   ::= [<label>] <subject.field> <pattern.field> <equal>
                [<object.field>] [<goto.field>] <eos>

<degen.stmt>  ::= [<label>] [<subject.field>] [<goto.field>] <eos>

<end.stmt>    ::= END [<blank>] [<label> | END ] <eos>

<statement>   ::= <assign.stmt> | <match.stmt> | <repl.stmt> |
                <degen.stmt> | <end.stmt>

```

This format is too complex to see clearly the form of each variant, and so each non-terminal name is abbreviated here to enable each production to fit on one line. With the exception of the equal sign, all the names represent non-terminals and so the angle brackets may also be omitted.

```

assign.smt    ::= [l] s      = [o]   [gto] <eos>
match.smt     ::= [l] s p                    [gto] <eos>
replace.smt   ::= [l] s p      = [o]   [gto] <eos>
degen.smt     ::= [l] [s]                    [gto] <eos>
statement     ::= assign.smt | match.smt | replace.smt | degen.smt

```

All the above productions start with an optional label, and most have a subject. The <end.statement> is omitted here as its start symbols differ from those in the above statements, and so it doesn't enter into the factorisation. It simplifies the factorisation if the degenerate statement is replaced by two productions, one with a subject and one without. By reordering the resultant list, the following list of productions is obtained.

```

assgn.smt      ::= [l] s      = [o]   [gto] <eos>
degen1.smt     ::= [l] s                               [gto] <eos>
match.smt      ::= [l] s p                               [gto] <eos>
replace.smt    ::= [l] s      = [o]   [gto] <eos>
degen2.smt     ::= [l]                               [gto] <eos>
statement      ::= assgn.smt | match.smt | replace.smt | degen.smt

```

These factor neatly into a production that starts with an optional label, and is followed by two alternatives, one with a subject part and one without.

```

<stmt>        ::= [l] [ s [p] [ = [o]] ] [gto] <eos>

```

Substituting the production names:

```

<stmt> ::= [<label>] [<subject> [<pattern> [= [<object>]]] [goto] <eos>

```

This will always show the possibility of a pattern replacement when in practise, direct assignment is more common. The productions can be altered to have the optional pattern as a alternative. The productions below, copied from the syntax used for GED, show the alterations.

```

<end.stmt>      ::= END      [ [<label> | END ] ]

<statement>    ::= [<label>] <stmt,goto,or.end> <eos>

<stmt,goto,or.end> ::= <stmt_body> | <goto.field> | <end.stmt>

<stmt_body>    ::= <subject.field>
                  ( [ = [ <object.field> ] ]
                    <pattern.field> [ = [ <object.field> ] ]
                  )
                  [<goto.field>]

```

Some productions have been relabelled as non-terminals so the non-terminal names will be used as the placeholder prompts.

#### 4.8 Defining the Print Formatting Commands

Minimal print formatting is needed to lay out a regenerated Snobol program - each statement or comment is printed on a separate line. The only other formatting commands are those needed to insert a blank where a <blanks> production has been removed. As GED, by default, prints all productions side-by-side, most productions do not need formatting commands.

The print formatting commands associated with <statement> and <comment> simply print either the prompt or the expansion of their placeholder, and then skip to a new line. Syntactically, the <goto.field> could be printed immediately after the earlier parts of the statement but it is much easier to read a Snobol program if the goto part is aligned on the right-hand side of the page. This is done by preceding the printing of the <goto.field> with a "tab to column 50" (@t50) command. This will cause all the <goto.field> expansions to be aligned at column 50. If however the cursor is already beyond column 50, no gap would be inserted before printing the ":" that starts the <goto> part. This would violate the syntax which specifies that a leading blank is necessary. Therefore the print formatting commands associated with the colon are:

```
@s @t50 @?
```

"meaning print a space", "tab to column 50", and print symbols (":")

Although the complete list of changes necessary to implement the Snobol editor seems long, the total time taken was only about two man days. A custom built system would, of course, take much longer to implement.

#### 4.9 The Implementation of Pascal and Lisp Editors

The major problems associated with implementing a Pascal editor were the same as those that occurred during the implementation of Snobol. The grammar had to be factored to remove replicated start symbols and extra productions added to cause the non-terminal names (such as <list of files>) to be used as a prompt. For example, the production



```
<heading> ::= PROGRAM <program_name> [ "( <name> {, <name>} )" ] ;
```

would cause the left parenthesis to be used as the prompt for the optional list of files. The prompt "[(" is much less informative than the "[<list\_of\_files>]" prompt given if the above construct is written as:

```
<heading> ::= PROGRAM <program_name> [<list_of_files>] ;
```

```
<list_of_files> ::= "( <name> {, <name>} )"
```

These alterations become easier to predict with practice. Although rewriting the grammar to remove left recursion and replicated start symbols could be mechanised, the complete process of the grammar preparation cannot. This is because many of the decisions, such as factoring out the part of productions (e.g. <list\_of\_files> above) into a separate production, are based on reasons of style and the relative occurrence of certain constructs. Another example would be the definition of the labelled statement in Pascal as "<statement>|<label><statement>" instead of the more obvious "[<label>]<statement>", in order to hide the rarely used label.

As GED will not expand a placeholder that leads to alternatives, differing intermediate level productions can be provided to act as prompts. This is most clearly illustrated by the syntax of a minimal subset of LISP, in which virtually everything produces an S-expression. This is illustrated in fig 4.5.

```
<Lisp Program> ::= <s-expression> { <s-expression> }
  printfmat    &1@?@n@n      { &2@?@n@n      }      $

<s-expression> ::= < atom > | <list>      $

<list>        ::= "( ( <lisp-function> |
  printfmat    &1@? ( &2@? <s-expression> { <s-expression> } )" |
  &3@?          ( &4@?   ) &5@? |
  )                                                    $

<lisp-function> ::= COND <pred_&_result>      " |
  CAR <s-expression>      " |
  CDR <s-expression>     " |
  DEFUN <function-name> <parameter list> <function-body> " |
  CONS <new-head> <old-list> "

printfmat      &1@?@>@n &2@?@< &3@n@? |
  &4@? &5@? &6 @? |
  &7@? &8@? &9 @? |
  &10@? &11@? &12@?@>@n &13@?@< &14@n@? |
  &15@? &16@? &17@? &18 @? |
  $

<pred_&_result> ::= "( <predicate> <result> )"
  printfmat      {"( <predicate> <result> ")}
  &1@? &2@? &3@? &4@?@n
  {&5@? &6@? &7@? &8@?@n} $

<predicate > ::= <s-expression> $

<result> ::= <s-expression> $

<function-name> ::= identifier $

<function-body> ::= <list> $

<parameter list> ::= "( <parameter> {<parameter>} )" $

<parameter> ::= identifier $

<new-head> ::= <s-expression> $

<old-list> ::= <s-expression> $

<atom> ::= identifier | number $ $$
```

**Fig 4.5 - Syntax of LISP Subset**

For example, the COND construct is displayed as

```
( COND
  ( <predicate> <result> )
)
```

which is much more informative than:

```
( COND
  ( <s-exp> <s-exp> )
)
```

The same applies to the parameter lists and body of function definitions. The distinction is important as it removes a source of ambiguity. The editor has the knowledge of which part of the syntax is currently being expanded and this information should be constantly available to the user.

The layout defined by the print formatting commands in the Lisp grammar above is illustrated by the following function (which is to look up atom X in a list Y) as output by GED.

```
(DEFUN lookup (x y )
  (COND
    ((eq y nil )nil )
    ((eq x (CAR y ))(CAR (CDR y )))
    (t (lookup x (CDR (CDR y ))))
  )
)
```

When defining Lisp one must decide whether a list of s-expressions should be printed on the same line, or be separated and indented. For example, a parameter list such as "(a b c)" should be printed on one line but the COND construct, with its multiplicity of predicates and results is much easier to read if the predicate and result are indented from the COND and displayed one (pair) per line, as illustrated above. Syntactically, there is no difference, as COND is an atom. However, unless a distinction is made the resulting format is unacceptable. To handle the two different cases the grammar is factored and the leading left parenthesis of made part of the production <list>. Until it is

known whether the contents of the list are going to be a COND or a DEFUN it is impossible to determine if a line-feed is necessary before printing the right parenthesis. Therefore the right parenthesis is made part of each alternative.

#### 4.10 Problems Encountered in the Addition of Formatting commands

Constructs that require indentation for part of a production but do not have a terminating symbol, such as the list of constant definitions in a constant declaration, can present problems. For example, if the `<const_declaration>` production was defined as

```
<const_declaration> ::= CONST    <const_defn> ;
                          { <const_defn> ; }

      printfmat    &1@?@>@n  &2@?    &3@?@n
                          { &4@?    &5@?@n}
```

then the output would be formatted as

```
CONST
  c1 = 1;
  c2 = 2;
  VAR ...    <--- Indentation level is incorrect
```

The CONST is printed, the indentation level increased and then a new line is started. The constant definitions are printed on separate lines, all indented one level, which is correct. However there is no place to put an "undent" one level (`@<`) command to reset the indentation level after the last definition in the list. This can be handled in several ways. If the construct always starts in a specific column (in this case, column one), then the tab rather than the indent command may be used to align the constant definitions. For example:

```

<const_declaration> ::= CONST      <const_defn> ;
                          { <const_defn> ; }

      printfmt      &1@?      &2@t7@?      &3@?@n
                          {&4@t7@?      &5@?@n}

```

will format the output as

```

CONST  c1 = 1;
        c2 = 2;
VAR...          <--- Indentation is Correct

```

This technique is only applicable if the construct is at a known column. More generally, the production can be rewritten as:

```

<constant_definitions> ::= CONST      <const_list>
                          printfmt      &1@?@>@n      &2@?@<@n

<const_list> ::= <constant_definition> ; { <constant_definition>; }
      printfmt      &1@?      &2@? { &3@n@?      &4@?}

```

This will format the output in the same manner as the previous example. Fortunately, it is rarely necessary to rewrite the grammar simply to preserve the indentation level, as this problem only occurs when the indented construct is an explicit list. Another example of a production with an indented production (without a balancing symbol on which to place an "indent" command) is the "while" statement. However, as the indented production (<statement>) is not a explicit list, no problems occur. The production and its formatting commands are:

```

<while statement> ::= WHILE <expression> DO
                          <statement>
      printfmt      &1@?      &2@?      &3@?@>@n
                          &4@?@<@n

```

This will format a while statement correctly, with <statement> being indented with respect to the WHILE and it will also reset the indentation level.

Lisp is intermediate in formatting complexity between Pascal, whose wealth of indented constructs renders it by far the most complex, and Snobol, which is the simplest. Apart from inserting blanks and tabbing to the right-hand side of the page for the destination parts of a statement, all Snobol productions use the default formatting (side-by-side).

#### 4.11 Summary

The implementation of a syntax-directed editor for a new language appears to be a major undertaking, but is much easier than is apparent from the written description of the problems. The removal of replicated start symbols is the major area in which the syntax must be structurally altered. Extracting segments of a grammar and turning them into new productions to provide descriptive prompts may be performed iteratively once the grammar is LLI. The largest part of the syntax preparation is in defining the print formatting information associated with each syntactic item. However, this is not because of any deficiency in GED - it is a necessary prerequisite whenever a pretty-printer is being defined for a new language.

While the amount of work necessary to implement a syntax-directed editor using GED is still significant, the time taken to bring up an editor for a new language is much less than that required to implement

such a system from the beginning. For flexibility, in both adding new language constructs and altering the presentation of the program, the use of a data-directed rather than a purpose written editor has much to recommend it.

## Chapter 5

### Conclusions

#### 5.1 A Short Description of the System

This thesis has traced the development of GED, a full-screen syntax-directed editor that is language-independent. The editor initially reads in as data a language syntax augmented with lexical and pretty-printing information, and is subsequently capable of syntax-oriented editing of programs in that language.

To aid the end-user, GED provides a skeletal program and prompts for insertions and alterations. It continuously displays the current production and all possible correct input symbols. As incorrect symbols are not accepted, no incorrect constructs can be incorporated in the program being built.

#### 5.2 The Realisation of Design Goals

During the editor's development, three main goals were pursued. These were:

5

1. It should be general. That is, it should be able to be set up to edit any (or nearly any) programming language.
2. It should be easy to set the editor up for a new language. It was considered that an editor which accepted a BNF language syntax as



its only controlling input would be maximally easy to set up, and that any information which needed to be specified in addition to the BNF detracted from this.

3. It should be easy for even the naive user to edit programs with GED. Colloquially, it should be user-friendly.

The following three sections of this chapter investigate the extent to which these goals have been achieved.

### 5.3 Generality of the Editor

GED is capable of handling any language which can be defined in BNF (i.e. it deals with context-free languages.) Many programming languages are defined in BNF (or a variant thereof) but are, in fact, context-sensitive, because they require identifiers to be declared. GED handles this case by flagging the first case of each user-defined symbol. This permits the use of the editor to alter declarations (or spellings) as necessary to avoid "syntactic" error-messages from the compiler. In this way, languages with declarations are included in the set which can be handled, and a handy spelling-checker is available when editing any type of language.

It is considered that the goal of generality has been achieved.

#### 5.4 Ease of Setting-up

In initialising the editor, the major task is the input of the language syntax. As this is represented in BNF, the most widely used syntax specification language, it is straightforward in most cases. However, there are two types of information which the editor needs, but which are not present in BNF.

The first is the lexical grammar of the user-defined symbols in the language. It is possible (cf. Snobol) to specify such a grammar in BNF, but this is rarely done. It has therefore been necessary to incorporate a facility for the analysis of lexical syntax into GED, and to preface each EBNF grammar with such a syntax. Although simple, the lexical analyser is general and will suffice for a wide variety of languages.

The second type of information missing from a BNF syntax specification concerns program layout. It is widely acknowledged that layout can be used as a powerful aid to program comprehensibility, but a syntax specification contains no information about layout. A simple notation for specifying indentation and other prettyprinting features, which needs only to be applied to some productions (e.g. those whose components can be expected to extend over several lines), and which can be incorporated into the BNF input syntax, has been devised.

Although the amount of work necessary to set up GED for a particular language is certainly non-trivial (amounting to an average of slightly less than one man-week for the languages used as examples), it is enormously less than would be required for the implementation of a

custom-designed syntax-oriented editor. Changes in language syntax can be incorporated with ease into GED, whereas a custom-designed editor would, in most cases, need major rewriting.

### 5.5 Ease of Use

No major public trials of the editor have been attempted, because of the difficulty of fitting in with the academic year, as GED was completed towards the end of one academic year and this thesis was due at about the beginning of the next. However, informal trials have indicated that it is an easy system to use, more particularly for new users who do not have preconceived ideas of how an editor "ought" to act on their program.

An area of difficulty encountered by some users concerns ascending and descending the program tree. Language definitions are not designed to facilitate this process, and often the number of commands necessary to reach a particular location in the tree is excessive. However, judicious massaging of the syntax by the implementor can alleviate this problem to some extent.

A similar problem relates to the mnemonic value of the names used for the various productions within the language. They are often obscure, but, again, the implementor can easily substitute more meaningful names.

The system is easy to edit with. It removes one of the bugbears of new users - the program with more error messages than statements. In an environment with many languages, it guarantees consistency of operation

of syntax-oriented editors.

### 5.6 Future Developments

This work has suggested two significant areas for future research. The first is a different technique for implementing the editor, while the second is a major increase in its functional capabilities.

As we have seen, the ordinary recursive descent parser's method of storing the state of a parse implicitly in its stack of return addresses is impractical for a syntax-directed editor. However, it should be possible to implement one as a multi-process recursive descent parser which forks a new process for each production being parsed. As each parse would have its own stack, no information would be lost when a process was suspended. Only the parser corresponding to the production currently under the cursor would receive input symbols, and movement of the cursor would thus automatically suspend one parse and resume another.

As the syntax-oriented editor incorporates a significant portion of a compiler, it seems natural to speculate about eliminating the gap between the two; i.e. developing a syntax-oriented editor which can execute the programs which it is used to build. This would require each syntactic production to be followed by a specification of its associated semantics in some interpretable language (e.g. Lisp), and would also require the inclusion of generalised symbol-table manipulations. It is difficult to see how this could all be accomplished without loss of generality, but it is an intriguing

problem nevertheless.

### 5.7 Final Thought

A syntax-directed editor is to programming what a word-processor is to English. Both are designed to simplify the task of document preparation - the differences occur only in the entities being manipulated.

## Acknowledgements

I would like to acknowledge the assistance given by my supervisor, Mr Paul Lyons, in the preparation of this thesis. His pertinent questions and critical comments contributed much to this thesis and its writeup.

My wife Anne contributed in a different way - through her support and understanding during many afternoons and evenings spent alone.

## Bibliography

- [Achugbue 81] Achugbue J O. "On the Line Breaking Problem in Text Formatting", Sigplan Notices, Vol 16, No 6, June 1981, p117.
- [Ada 80] United States Department of Defence. "Reference Manual for the Ada programming Language", 80 Proposed Standard Document.
- [Allen 81] Allen T, Nix R, Perlis A. "PEN: A Hierarchical Document Editor", Sigplan Notices, Vol 16, No 6, June 1981, p74.
- [Archer 79] Archer J (Jnr), Shore A. "A Program Development System Execution Supervisor", Report # TR 79-397. Department of Computer Science, Cornell Unive
- [Archer 81] Archer J (Jnr), Conway R. "COPE: A Cooperative Programming Environment", Report # TR 81-459. Department of Computer Science, Cornell University
- [Atkinson 81] Atkinson L V, North S D. "COPAS - Conversational Pascal System", Software Practice and Experience, Vol 11, 1981, pp 819-829.
- [Barach 81] Barach D R, Taenzer D H, Wells R E. "The Design of the PEN Video Editor Display Module", Sigplan Notices, Vol 16, No 6, June 1981, p130.
- [Brown 81] Brown P J. "Dynamic Program Building", Software Practice and Experience, Vol 11, 1981, pp 831-843.
- [Chamberlin 81] Chamberlin D D, King J C, Slutz D R, Todd S J P, Wade B W. "JANUS, An Interactive System for Document Composition", Sigplan Notices, Vol 16, No 6, June 1981. p82.
- [Cherry 81] Cherry L. "Computer Aids for Writers", Sigplan Notices, Vol 16, No 6, June 1981, p61.
- [Cohen 70] Cohen D J, Gottlieb C C. "A List Structure Form of Grammars for Syntactic Analysis", Computing Surveys, Vol 2, No 1, March 1970.
- [Demers 81] Demers A, Reps T, Teitelbaum T. "Incremental evaluation for attribute grammars with application to syntax-directed editors", 8th POPL Conference. p105.

- [Deutsch 81] Deutsch M S. "Software Project Verification and Validation (Tutorial)", IEEE Computer Magazine, April 1981, p54.
- [Elliot 82] Elliot B. "The Design of a Simple Screen Editor", Software Practice and Experience, Vol 12, 1982, pp 375-384.
- [Englund 81] Englund R M. "The Coming Decade of Innovation - A Workshop Report", IEEE Computer Magazine, April 1981, p77.
- [Feiler 81] Feiler P H, Medina-Mora R. "An Incremental Programming Environment", IEEE SW Engineering 1981, p44.
- [Fischer 81] Fischer C, Johnson G, Mauney J. "An Introduction to Release 1 of Editor Allan Poe", CS Tech Report #451, University of Wisconsin, Madison.
- [Fraser 81] Fraser C W. "Syntax Directed Editing of General Data Structures", Sigplan Notices, Vol 16, No 6, June 1981, p17.
- [Goldfarb 81] Goldfarb C F. "A Generalised Approach To Document Markup", Sigplan Notices, Vol 16, No 6, June 1981, p68.
- [Good 81] Good M. "Etude and the Folklore of User Interface Design", Sigplan Notices, Vol 16, No 6, June 1981, p34.
- [Gosling 81] Gosling J. "A Redisplay Algorithm", Sigplan Notices, Vol 16, No 6, June 1981, p123.
- [Gutz 81] Gutz S, Wasserman A I, Spier M J. "Professional Development Systems for the Professional Programmer", IEEE Computer Magazine, April 1981, p45.
- [Habermann 82] Habermann A N. "System Development Environments" in "Tools & Notions for Program Construction" - An Advanced Course, edited by D. Neel. Cambridge University Press (1982). p247.
- [Hammer 81] Hammer M, Ilson R, Anderson T, Gilbert E, Good M et al. "An Implementation of ETUDE, An Integrated and Interactive Document Production System". Sigplan Notices, Vol 16, No6, June 1981. p137.
- [Hansen 71] Hansen W J. "Creation of Hierachic Text With a Computer Display", PhD Thesis - Stanford, Report # ANL-7818. , Argonne National Laboratory, Argonne, Illinois.



- [Ivie 77] Ivie E L. "The Programmer's Workbench - A Machine for Software Development", Communications of ACM, October 1977, Vol 20, No 10, p746.
- [Jensen 74] Jensen, K and Wirth, N. "Pascal User Manual and Report", Springer Verlag, 1974
- [Kernighan 81] Kernighan B W. "PIC -- A Language for Typesetting Graphics", Sigplan Notices, Vol 16, No 6, June 1981, p92.
- [Kernighan 80] Kernighan, B.W. and Ritchie, D.M. Prentice Hall Software Series: "The C Programming Language" Prentice-Hall 1978.
- [Lakos 82] Lakos C A, McDermott T S. "Interfacing with the User of a Syntax Directed Editor", Report #R 82-3 Department of Information Science, University of Tasmania, Hobart.
- [Lesk 75] Lesk, M.E. "LEX - a lexical analyser generator", CSTR 39, Bell Laboratories, Murray Hill, New Jersey.
- [Lyons 83] "TWIJI - A Written Version of the Tramline Syntax Notation", Personal communication from P.J.Lyons, Computer Science Department, Massey University, Palmerston North, New Zealand.
- [Medina-Mora 81] Medina-Mora R, Notkin D S. "ALOE Users' and Implementors' Guide", Report #CMU-CS-81-145. Department of Computer Science, Carnegie-Mellon University.
- [Mikelsons 81] Mikelsons M. "Prettyprinting in an Interactive Environment", Sigplan Notices, Vol 16, No 6, June 1981, p108.
- [Morris 81] Morris J M, Schwartz M D. "The Design of a Language-Directed Editor for Block Structured Languages", Sigplan Notices, Vol 16, No 6, June 1981, p28.
- [Osterweil 81] Osterweil L. "Software Environment Research: Directions for the Next Five Years", IEEE Computer Magazine, April 1981, p35.
- [Pagan 81] Pagan, F G. "Formal Specification of Programming Languages: A Panoramic Primer" Prentice-Hall Inc (1981) Englewood Cliffs, New Jersey, pp 21-22
- [Reid 81] Reid B K, Hanson D. "An Annotated Bibliography of Background Material on Text Manipulation", Sigplan Notices, Vol 16, No 6, June 1981, p157.

- [Reps 81] Reps T. "Optimal-time Incremental semantic analysis for syntax directed editors.", Cornell University, TR 81-453.
- [Snook 78] Snook T, Bass C, Roberts J, Nahapetian A, Fay M. "Report on the Programming Language PLZ/SYS", Springer-Verlag, New York, 1978.
- [Stallman 81] Stallman R M. "EMACS - The Extensible Customisable Self-Documenting Display Editor", Sigplan Notices, Vol 16, No 6, June 1981, p147.
- [Stromfors 81] Stromfors O, Jonesjo L. "The Implementation and Experiences of a Structure Oriented Editor", Sigplan Notices, Vol 16, No 6, June 1981, p22.
- [Tee 83] Personal communication from G.J.Tee, Computer Science Dept, University of Auckland, New Zealand.
- [Teitelbaum 80] Teitelbaum T. "The Cornell Program Synthesizer: A Tutorial Introduction", Report # TR 79-381 (revised 1980). Department of Computer Science, Co
- [Teitelbaum 80] Teitelbaum T, Reps T. "The Cornell Program Synthesizer: A Syntax-Directed Programming Environment", Report # TR 80-421. Department of Computer Science. Cornell University.
- [Teitelbaum 81] Teitelbaum T, Reps T, Horwitz S. "The Why and Wherefore of the Cornell Program Synthesizer.", Sigplan Notices, Vol 16, No 6, June 1981, p8.
- [Teitelbaum 81] Teitelbaum W, Masinter L. "The Interlisp Programming Environment" IEEE Computer Magazine, April 1981, p25.
- [Teitelbaum 81] Teitelbaum T, Reps T. "The Cornell Program Synthesizer: A Syntax Directed Programming Environment", Communications of the ACM, September 1981, Vol 24. No 9. p563.
- [Turba 81] Turba T N. "Checking for Spelling and Typographical Errors in Computer-Based Text", Sigplan Notices, Vol 16, No 6, June 1981, p51.
- [Van Wyk 81] Van Wyk C J. "A Typesetting Language", Sigplan Notices, Vol 16, No 6, June 1981, p99.
- [Vickers 80] Vickers, S. "ZX81 Basic Programming", Sinclair Research Limited, Cambridge England (1980).
- [Vickers 82] Vickers, S. "ZX Spectrum Basic Programming", Sinclair Research Limited, Cambridge England (1982).

- [Walker 81] Walker J H. "The Document Editor: A Support Environment for Preparing Technical Documents", Sigplan Notices, Vol 16, No 6, June 1981, p44.
- [Wasserman ] Wasserman A I. "Automated Development Environments" University of California, San Francisco
- [Waters 82] Waters R C. "The Programmers Apprentice: Knowledge Based Program Editing", IEEE Transactions on Software Engineering, Vol SE- 8, No 1, January 1982.
- [Wirth 77] Wirth N. "What Can we do about the Unnecessary Diversity of Notation for Syntactic definitions?", CACM November 1977, Vol 20, no 11 pp 822-823.
- [Wood 81] Wood S R. "Z -- The 95% Program Editor", Sigplan Notices, Vol 16, No 6, June 1981, pl.