COMPUTERSYSTEMSLABORATORY

DEPARTMENTS OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE
STANFORDUNIVERSITY · STANFORD, CA94305

EDT
A Syntax-Based Program Editor
Reference Manual

Ross S. Finlayson

Technical Report No. 83-245
PAV Report No. **21**

July 1983

This research was supported by the Advanced Research Projects Agency of the Dcpartmcnt of Dcfcnsc under Contracts MDA 903-80-C-0159 and NOO-039-82-C-0250.

EDT
A Syntax-Based Program Editor
Reference Manual

Ross S. Finlayson

Technical Report No. 83-245
Program Analysis & Verification Report NO. 21

July 1983

Computer Systems Laboratory
Departments of Electrical Engineering and Computer Science
Stanford University
Stanford, California 94305

Abstract

This report describes an experimental syntax-based editor that has recently been developed at Stanford. Syntax-based editors are unlike conventional text editors in that they use knowledge of the syntactic structure of the item (typically a program) being edited, to provide 'high-level' editing operations. The editor described in this report is currently being used as an editor for programs written in Ada. Other programming languages could also be handled, by replacing the appropriate language definition files by those for another language.

Key Words and Phrases: Ada, syntax-based program editor

EDT—A Syntax-Based Program Editor. Reference Manual

Ross S. Finlayson

Stanford University
Computer Science Department

May 20, 1983

1. Introduction.

Recently, there has been considerable interest in syntax-based program editors. Unlike standard text editors, syntax-based editors use knowledge of the syntactic structure of the programming language, to provide high-level language *specific* editing facilities. Accordingly, the user can be freed from much of the 'drudgery' that often accompanies program development (such as having to make sure that his program is correctly indented, and that no 'end's have been omitted). In addition, syntax errors can be eliminated entirely, since such editors will usually allow only syntactically legal constructs. Some systems also perform semantic checking. Probably the most well-known example of such an editor is the Cornell Program Synthesizer [2], an editor for programs written in PL/CS, an instructional dialect of PL/I. Much has been said about the desirability of a uniform programming environment, incorporating powerful language-specific tools to facilitate program development and maintenance. A syntax-based editor could be such a tool, and could be integrated with other tools such as a debugger, an interpreter/compiler and a verifier.

This report describes EDT-a syntax-based program editor developed for the Program Analysis and Verification Group at Stanford. This editor is written in MACLISP. It is unlike the Cornell editor, in that it is not 'hard-wired' to support any one programming language. Instead, the architecture of the EDT system (and indeed, the user commands for the system) do not presuppose the language to be edited. By incorporating files that describe the syntax of the programming language to be edited, EDT can be reconfigured to provide syntax-based editing for that particular language. This is described in more detail in section 3.

Currently, one such programming language is supported-the programming language Ada™ (although the complete Ada language is not yet supported). The resulting (Ada) editor is called ADAEDT. The remainder of this manual can be thought of as describing this particular editor. However, it should be remembered that most of the manual would still apply if the editor were reconfigured for another language. (The best languages for this editor would be those with a rich set of possible syntactic constructs—Ada and COBOL, for example. Languages such as LISP, which has a very parsiminous syntax, would be less suitable.)

Three of the major considerations that influenced the design of the system were:

(i) Naive users-It was anticipated that many of the users of the editor would be unfamiliar with the syntax of the language that is being used (Ada in this case). The editor provides *menus,* to help the user select the correct syntactic construct to use.

(ii) Modularity-The terminal, screen and operating system dependent portions of the editor are clearly separated from the rest of the system. The editor generates a terminal independent internal Lisp data structure to denote what is to be displayed on the screen. Low-level, terminal dependent display drivers then work from this data structure.

(iii) Interface with other tools-Internally, the editor uses a tree to represent what is being edited. The editor could be used as the 'front-end' to another tool (such as a compiler).

("Ada" is a registered traticmnrk of the U.S. Department of Defense.)

Any such tool would work directly from this internal tree form. Thus, the internal format should be flexible enough to be able to incorporate any extra information that the 'back-end' tool may require.

Section 2 of this report can be thought of as the user manual proper, in that it describes how to use the editor, but does not discuss implementation details. Section 3 describes the 'architecture' of the system, display algorithms, internal formats etc. A separate manual-the EDT Implementation Guide, contains even more details of the implementation, including much of the source code.

This work was supported by the Advanced Research Projects Agency, U.S. Department of Defense, under contract NOO-039-82-C-0250.

2. User Guide.

Running the Editor.

The editor currently runs only on SAIL, supporting a (moderately large) subset of the programming language Ada. At present, two kinds of display terminal can be used to run the editor-DataDiscs and Datamedias. To use the editor on a DataDisc terminal, type R ADAEDD. If you are on a Datamedia terminal (or an equivalent terminal, such as an Alto running TALK or DMCHAT), type R ADAEDM. This (Datamedia) version of the editor is the preferred version at present, because the (low-level) display routines for DataDisc terminals are considerably slower than the corresponding Datamedia routines. On the other hand, the DataDisc has the advantage of having the SAIL keyboard (which makes the editor easier to use), and of being able to display more lines of text than Datamedias.

After you start up the editor, the screen will be cleared and partitioned in two, and the editor will display READY. The top portion of the screen is the editor's special display window. The editor is built on top of LISP, but does not completely hide the underlying LISP interpreter. The bottom portion of the screen is, in fact, set aside for the LISP interpreter, and behaves just as if you were talking to LISP alone (including printing normal LISP error messages if something goes wrong!).

The Editor's Internal Tree.

Any program (or program fragment) that the user edits is represented internally as a tree. Each editable part of the program is a node of this tree. The root of the tree represents the entire program (fragment) that is currently being edited. The structure of this tree is intended to reflect the structure of the program fragment. For example, a node representing an Ada **for** statement has three 'sons'. The first son represents the index variable of the for statement. The second son represents the discrete range over which this index variable will range. The third son represents the statement list that forms the body of the **for** statement. Note that purely syntactic items such as reserved words and semicolons are not represented internally as nodes, although such things will, of course, be displayed on the user's screen.

The leaves of the tree (i.e. those nodes that have no sons) represent items that can be considered 'atomic', in that they have no editable subparts. Examples of such nodes are identifiers, numbers and nonterminals (discussed below).

The nodes of the tree fall into three classes. Firstly, there are the 'ordinary' nodes, which represent actual programming language structures. Most nodes will usually be of this type-the for statement node mentioned above would be such a node, for instance.

Secondly, there are *nonterminal* nodes. When displayed, they are delimited by left and right angled brackets, for example "<statement>". These nodes can be thought of as 'unexpanded placeholders', which can later be filled in (i.e. replaced) by an 'ordinary' node. Nontcrminals serve two, related, purposes. As mentioned above, they serve as

placeholders for nodes not yet expanded. In addition, they help ensure that the program as a whole remains syntactically consistent. For example, a **for** statement node could replace the nonterminal node <statement>, but could not replace the nonterminal node <expression>, since a **for** statement is a statement, but not an expression.

Thirdly, there are pattern variable nodes. These nodes are used for searching and replacement, which are described later.

The following diagram illustrates (the top branches of) the internal tree form for one particular Ada for statement.

The editor's internal tree structure is described in more detail in section 3.

The Cursor.

At any one time, there is one particular node in the tree that can be edited by the user. This (distinguished) node is the cursor node. Virtually all of the editor commands apply to this one node, either to change it in some way ("manipulating the cursor"), or to make some new node the cursor node ("moving the cursor").

On the user's screen, the text represented by the cursor node (and all descendants of this node) is emphasized, relative to other (surrounding) text. The exact nature of this emphasis is terminal-dependent. On DataDisc terminals, reverse *video* is used. On Datamedia terminals, the pixels of the cursor text characters are are dispiayed brighter than those of the characters of surrounding text. (An Alto running TALK displays the cursor in reverse video, however.)

Throughout this report, the term cursor is used to refer both to the (internally represented) cursor node (mentioned above), and to the text represented by this node, which appears (highlighted) on the user's screen. Which of these two meanings is intended will usually be clear from the context in which the term appears.

Commands to the Editor.

The commands to the editor are each in the form of control characters. That is, you hold down the <CONTROL> key and type another key, depending upon the command. In this respect the editor's command interface is rather primitive. It will probably be improved sometime in the future. Case is significant-for example the **<ctrl>i** command is different from the **<ctrl>I** command. Often, after typing the appropriate control character, you will be prompted for further input, such as a file name. In such cases you type the required input, followed by **<cr>**.

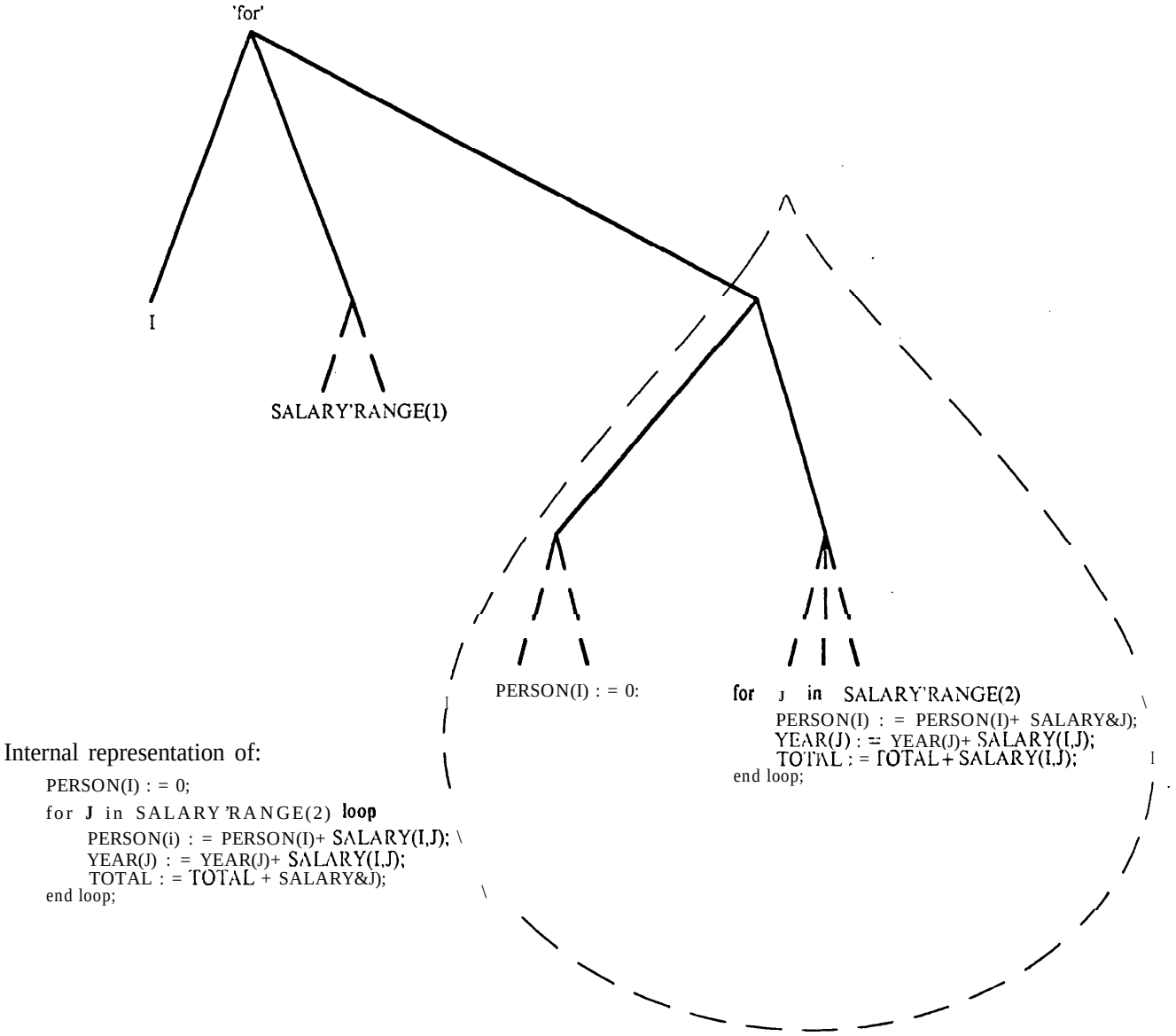


Figure 1: Internal representation of:

```
for I in SALAKY'RANGE(1) loop
    PERSON(I) : = 0;
    for J in SALARY'RANGE(2) loop
        PERSON(I) : = PERSON(I)+ SALARY(I,J);
        YEAR(J) : = YEAR(J)+SALARY(I,J);
        TOTAL : = TOTAL+SALARY(I,J);
    end loop;
end loop;
```

File Input Commands.

Having started up the editor, you'll need something to edit! One way to get something to edit is to input it from a previously written file. The <ctrl>i and <ctrl>I commands enable you to do this.

If you have an Ada compilation unit (e.g. a package or a subprogram) on file in text form, you can input it to the editor using the <ctrl>I command. You will be asked for the name of the file, and also for a name to give to the new tree that will be constructed as a result of the input. (Whenever you create a completely new tree, the editor asks you to give a name for it. This is done in order to avoid the risk of this tree becoming 'orphaned' at some later stage.)

Since the input file is in text form, it is first read by the parser, and checked for syntactic correctness. (Note that at present the editor has no knowledge about *semantic* correctness.) If a *syntax* error is reported, there will be one of three possible causes:

(i) There may be a genuine Ada syntax error.

(ii) The piece of text in question may (strictly speaking) be legal Ada, but a part of Ada of which the editor (in particular, the parser) is not currently aware. Appendix B describes those parts of Ada which the editor does not currently support.

(iii) The complete piece of text may be legal Ada, but may not be an Ada compilation unit. For example, it may instead be only an Ada statement. For the parser to be used, it must know exactly what kind of syntactic (nonterminal) element it is expected to check for. For the <ctrl>I command, 'compilation-unit is assumed. If your text file contains 'less' than a compilation-unit, you must first create a new 'bare' node of the appropriate nonterminal type, using the <ctrl>W command, and then 'parse into' this new nonterminal node, using the <ctrl>P command. These commands are described later.

If no syntax error is found, the input program will be displayed on the screen. The entire compilation unit will be highlighted, indicating that the cursor is now at the root of this new tree.

There are several files of examples on which you can apply the <ctrl>I command. The files are TEST1.ADA[EDT,RSF] . . . TEST8 .ADA [EDT ,RSF] on SAIL. Most of these examples are taken from the CACM Ada self-assessment test [3]. (Note that the editor does not assume a default file extension with any of its file I/O commands-you have to type in the ".ADA".)

While using the editor, it is possible to file away (using a command to be described later) the node that you are currently editing, so that it can be restored during a later session. If you do this, you are actually filing the internal form of the node, rather than the text that the node represents. To input a node that was previously saved in internal form, use the <ctrl>i command. As with <ctrl>I, you will be asked for a file name and a new node name. Unlike the <ctrl>I command, however, a tree input using the <ctrl>i command need not be an entire compilation unit.

The internal tree forms of the compilation units given in the example test files above have been filed on TEST1.TRE[EDT,RSF]...TEST8.TRE[EDT,RSF]. The <ctrl>i command can be tried out on these files.

Note that you can use commands such as <ctrl>i and <ctrl>I even if you are editing another tree at the time.

Moving the Cursor.

Remember that when you edit a program using this editor, you are not editing a piece of program text, but rather, you are editing the syntactic structure of the program. In particular, you are editing the tree-structured representation that was mentioned earlier. Cursor movements, for instance, should be thought of as movements inside this syntactic structure, rather than as movements in any particular direction on the display. If you have previously used only standard text editors to compose and edit programs, then this difference of emphasis could prove to be difficult at first. I believe, however, that once this concept is grasped, the advantages (or potential advantages) of this editor (and other syntax-based editors) should become apparent.

The command <ctrl>/ is used to move the cursor to the leftmost son (if any) of the original cursor. For example, if the cursor is at a **for** statement, then <ctrl>/ will move it to the **for** statement's index variable. If the cursor is at an **if** statement, <ctrl>/ will move it to the if statement's conditional expression. As a third example, if the cursor is at (the top of) a list of two or more statements (e.g. in a loop), then <ctrl>/ will move it to the first statement in that list. Note that reserved words (such as **for** and **if** in the examples above) are skipped over, since they are not represented as explicit nodes.

Similarly, the <ctrl>\ command moves the cursor to the rightmost son of the original cursor. As with all cursor movement commands, it has no effect if there is no such node to move to.

<ctrl>↑ can be thought of as the 'inverse' of <ctrl>/ and <ctrl>\. It is used to move to the 'parent' of the cursor. <ctrl>∩ extends this--it moves the cursor to the root of the tree (i.e. the complete item being edited).

<ctrl>→ moves the cursor to the right brother of the original cursor. If there is no immediate right brother, then the editor looks for an ancestor that has an immediate right brother, and then moves the cursor to that right brother. This feature can be very useful. As an example, suppose that the cursor is at the "3" in:

```
FOO := BAR + 3;
BAR := BAR + 1;
```

Then, <ctrl>→ will move the cursor to the statement: BAR := BAR + 1;. <ctrl>← moves to the left brother of the original cursor. It behaves analogously to <ctrl>→. <ctrl>ↄ is used to move to rightmost (i.e. the farthest right) brother of the cursor. <ctrl>ↄ moves to the leftmost brother. Unlike <ctrl>→ and <ctrl>←, these commands only consider immediate brothers.

Note that the application of any of these cursor movement commands does not change the structure of the internal tree in any way. All that is done is that a new node of this tree becomes known as the cursor. Commands which actually change the node being edited will be described later.

Naming Nodes.

All commands which create (and move the cursor to) new trees require you to give a name for (the root of) that new tree. In fact, you can give a name to any node-the <ctrl>n command is used to name the cursor. You should note that such names are used only for editing purposes, and do not appear on the screen. In particular, they have no relation to any program *labels* that may be present. Anything that would be a valid LISP atom can be used as a name. The editor will not let you use a name that is already in use, and you cannot use the <ctrl>n command if the cursor already has a name. If the cursor already has a name, and you really want to rename it, then use the <ctrl>N command. The <ctrl>? command tells you whether or not the cursor currently has a name, and if so, what that name is.

To move the cursor to a previously named node, use the <ctrl>t command. You will be asked for the name of the node to go to, of course.

Reverting to the Previous Cursor Position.

The <ctrl>v command will move the cursor back to where it was just before the most recent (major move'-i.e. back to where it was before the most recent (successful) <ctrl>t, <ctrl>∩, <ctrl>v, or the creation of a new tree (e.g. with <ctrl>i, <ctrl>I or <ctrl>W (described later)). 'Basic' movements such as <ctrl>/ and <ctrl>↑ are not taken into consideration. This command is useful if, for instance, you go to another tree (with <ctrl>t), edit for a while, and then want to go back to editing the original tree exactly where you left off. This command only works 'at one level'-the previous cursor positions are not 'stacked'.

Inserting Copies of Previously Named Nodes.

If the cursor is at a nonterminal node, then you can fill it in by inserting a copy of any node (of a compatible nonterminal type) that you have previously named. For example, if the cursor is at a <statement> nonterminal, and if you have previously named another (expanded) statement node *foo,* then by typing <ctrl>c foo <cr>, a copy of foo will be inserted in place of <statement>. Note that it will be a copy of *foo,* and not *foo* itself. Thus, if you go back and change *foo,* you won't alter the new node. (The editor never shares the same node at different places in the tree-it always produces a copy.)

Often, a node will be 'compatible' with a large number of different nonterminals. For example, the node FOO + BAR could be inserted (using <ctrl>c) in place of any of: <expression>, <parameter> (e.g. as a parameter in a procedure or function invocation, or as an array index) and <choice> (e.g. in a branch of a case statement).

Deletion and Undeletion.

<ctrl>d will delete the cursor node, replacing it by an appropriate nonterminal node. This can be thought of as a 'soft' deletion, since the original node being edited does not vanish completely, but instead a nonterminal node is left in its place. Exactly which nonterminal replaces the original cursor will often depend upon the context in which the cursor appears. For instance (considering the example given above), if the node FOO + BAR were (soft) deleted using <ctrl>d, it could be replaced by <expression>, <parameter> or <choice>, depending upon the context in which the node appeared.

If the cursor node is part of a list, for example a procedure/function parameter, array index or a statement or declaration forming part of a statement or declaration list, then it can be deleted entirely, by typing <ctrl>D. This can be thought of as a 'hard' deletion, since nothing is left in place of the original node. After the deletion, the cursor will move to the right brother of the original node, or to the left brother, if the original node had been the final element of the list. (In the special case where the (original) cursor had been the only element of the list, the cursor will move up to this (now empty) list.) The editor will not let you perform a hard deletion if such a deletion would make the enclosing list smaller than it is allowed to be.

The reason that hard deletions are allowed in lists is, of course, because they preserve the syntactic integrity of the program (fragment) being edited. This would not in general be true if hard deletions were allowed elsewhere. However, there are circumstances where it would be desirable to be able to perform hard deletions elsewhere than inside a list, but still preserve syntactic correctness. Two examples would be the deletion of the closing identifier of a subprogram or package, and the deletion of the private part of a package specification. Unfortunately, the editor does not allow you to perform this kind of hard deletion with a single command. Such deletions can still be performed, though, by performing an appropriate coercion on the parent of the node in question. (Coercions will be described later .)

If a node that was previously named is (soft or hard) deleted, then the deleted node retains the name that it was previously given, and, in the case of a soft deletion, the replacement nonterminal node is (initially) unnamed. This allows you to name a subnode of the tree being edited, 'chop it off ', and then come back to it at some later stage (as well, the 'undeletion' facility mentioned in the next paragraph allows you to restore the node that was *most recently* deleted). There is an exception to this rule, however. If the root of the tree being edited is deleted, then it is the resulting nontcrminal node that retains the original root's name. The reason for this exception is that the entire tree will usually be deleted only to correct an error. In such circumstances it would be most desirable for the root's name to be retained by the nonterminal (and thus, by any correction that follows).

The editor remembers the last node that was deleted (by either <ctrl>d or <ctrl>D). At any later stage, you can 'undelete' this node, if you are at a nonterminal node of compatible type (in particular, you can *immediately* undo a soft deletion). You do this by typing <ctrl>u. As with <ctrl>c, this inserts a copy of the node that was most recently deleted. This means that you can USC <ctrl>u several times, at different instances of the compatible nonterminals. In each case a copy of the (same) last deleted node will be inserted in place. You cannot undo the effect of a *hard* deletion immediately, although

you can still. undo a hard deletion by inserting a new nonterminal node to the left or right of the resulting cursor position (using the <ctrl>{ or <ctrl>} commands, to be described later), and then using <ctrl>u.

Finally, suppose that *you* wish to *simultaneously* delete a node and undelete a previously deleted node in the same place. You cannot do this with any of the commands so far described. The <ctrl>U command allows you to do this. This command deletes the node that is currently at the cursor position, and inserts the most recently deleted node in its place. (The original node is then remembered as being the node "most recently" deleted.)

Adding to a List.

If the cursor is at an element of a list (for example, at a statement inside a subprogram body, or at a parameter inside a subprogram call), then you can add to the list by using the commands <ctrl>{ and <ctrl>}. <ctrl>{ inserts a new (nonterminal) node just to the left of the cursor, and then moves the cursor there. Similarly, <ctrl>} inserts a new (nonterminal) node just to the right of the (original) cursor.

Note that every element of a list is of the same nonterminal type.

Creating New (Nonterminal) Nodes.

If you wish to build a tree 'from scratch', then you will need to obtain a 'bare' root node to start from. The <ctrl>W command enables you to do this. You will be asked for a name to give the new node, and also for the nonterminal type of the new node. The cursor will move to the resulting nonterminal node. Three nonterminal types that will be commonly used are compilation-unit, statement and expression (a complete list is given in Appendix B). Note that you do not type any angled brackets around the nonterminal name.

Invoking the Parser.

Suppose the cursor is at a nonterminal node, which you wish to 'fill in'. So far, we have described how this could be done by using <ctrl>c to insert a copy of some previously named node, or by using <ctrl>u to 'undelete' the last node that was deleted. However, neither of these commands is of any help to us if we want to expand the nonterminal into something new. You can do this by using *productions* (described later), by reading a piece of program text from an input file, or by entering text from the keyboard.

The last two of these require that the text be. checked by the *parser,* since the editor must maintain syntactic correctness at all times. If you wish to fill in a nonterminal node by typing text from keyboard, use the <ctrl>p command. You will be prompted by a

"**>**" (at the bottom of the screen). Type in the text (it can be over more than one line if you wish), and then type **<altmode> <cr>**. Admittedly this is a bit of a nuisance, but the parser needs a symbol to represent 'end-of-file'. **<altmode>** serves this purpose. If the parser detects a syntax error in your input, then an error message will appear, and no action is taken. Otherwise, the nonterminal will be filled in by (a node representing) the text that you typed in.

It is also possible to fill in a nonterminal node with text from an input file, using the **<ctrl>P** command. You will be asked for the name of the input file. The parser will check the text in the file, as it does with **<ctrl>p**. Note how this command differs from the **<ctrl>I** command, which assumes that the text in the input file is a compilation unit, and which creates a completely new (isolated) node, rather than filling in an already existing node.

It should be noted that the parser does not recognize nonterminals in any text that it reads (whether or not they are surrounded by angled brackets). In particular, the program text in a file that is read using the **<ctrl>I** or **<ctrl>P** commands cannot contain (unexpanded) nonterminals. (The parser does not recognize pattern variables either.)

Productions (on Nonterminal Nodes).

Most nonterminals can also be filled in using productions. These allow you to choose a 'skeleton' of an expanded node from a menu, and use this skeleton to fill in the nonterminal.

If the cursor is at a nonterminal node, then the production *menu* can be displayed by typing either **<ctrl>E** or **<ctrl>Q**. If the menu is empty, you cannot use productions. This only applies to nonterminals such as <identifier> which can be filled in only by typing at the keyboard (using **<ctrl>p**). (Note that the editor will display identifiers in upper case, regardless of how they are typed in. Ada reserved words are displayed in lower case.)

Each entry in the production menu represents a possible substitute for the nonterminal that appears at the top of the menu. To the left of each entry is a number and a mnemonic. To select an item from the menu, type **<ctrl>e** mnemonic **<cr>**, or **<ctrl>q** mnemonic **<cr>**, where mnemonic is the mnemonic to the left of the appropriate entry in the menu. The cursor will then be filled in by (a copy of) that menu entry. If you wish, you can type the number that appears at the far left, instead of the mnemonic. Of course, if you can remember the mnemonic that you want, then you can use **<ctrl>e** or **<ctrl>q** without looking at the menu.

If there is only one possible production for the nonterminal in question, then the production will be performed without you being asked for a mnemonic. This frequently applies to 'list nonterminals' (such as <statement-list> and <declarative-part>). The editor will fill in such nonterminals with the appropriate list node, containing two (nonterminal) sons. This number of sons (two) is arbitrary, and of course you can later go down into the list and either decrease the number of sons (using **<ctrl>D**), or increase the number of sons (using **<ctrl>{** or **<ctrl>}**).

<ctrl>C will erase any menu that.is displayed, and will give you the normal display window again. (Any other (non-menu) command will also delete the menu and restore the original display, but <ctrl>C is useful if you just want to restore the display, without doing anything else.) If a menu cannot fit entirely on the screen, then you can scroll forward using the <ctrl>~ command, or scroll back using the <ctrl>% command. (A highlighted message will appear at the top (and/or the bottom) of a menu whenever it is necessary to use these menu scrolling commands in order to see more of the menu than is currently showing.)

The above remarks concerning how to select items from a menu, as well as those concerning the <ctrl>C and menu scrolling commands, also apply to the coercion and embedding menus (coercions and embeddings will be described shortly).

When using the editor, you will often have to decide whether to fill in a nonterminal node by using a production, or whether to fill it in by using <ctrl>p and typing in text directly from the keyboard. <ctrl>p is usually quicker for (all but the most complicated) expressions, as well as for simple assignment statements and procedure calls (but don't forget the ";" at the end of Ada statements!). Productions are usually quicker for the more complex statements (such as if, case and block statements), as well as for subprograms and packages.

Note that any command that 'fills in' a nonterminal node will preserve any name that the nonterminal node originally had.

File Output Commands.

It is possible to output the tree that you are editing to a file, either in text form, or in internal tree form (so that it can subsequently be restored using <ctrl>i).

The <ctrl>O command outputs the tree to a file, in text form. You will, of course, be asked for the name of an output file. The tree will be output to this file *in* full. That is, there will be no ellipsis in the output, even if some subnodes happen to be elided on the display when the <ctrl>O command is given. When using this command, you will be asked to choose between SCRIBE output and plaintext (i.e. normal) output. The former choice is useful if you want to (later) make a 'pretty-printed' hardcopy using the SCRIBE program.

The <ctrl>o command outputs the tree to a file, in internal tree form. Both of the above commands can be used even if the tree that you wish to output contains nontcrminals or pattern variables (although if you use <ctrl>O to file away such trees, then you won't be able to read them back later with <ctrl>I or <ctrl>P).

Note that, in contrast to most of the other commands, which apply only to the cursor, the <ctrl>o and <ctrl>O commands output the *entire* tree that is being edited.

Coercions and Embeddings.

It is possible to manipulate an expanded node in any (syntactically legal) way by using

the commands that we have described so far. However, coercions and embeddings allow you to perform some of the more common manipulations far more easily.

A coercion performs an immediate transformation of an expanded node into another kind of node. For example, a (normal) **if** statement can be transformed into an **if** statement with **elsif** parts (or an **else** part, or both). An if statement can also be transformed into a **while** statement. A ">" expression can be transformed into a "<", and so on.

If the cursor is at an expanded node (as opposed to a nonterminal), you can see the coercion menu by typing <ctrl>Q. (Recall that if you are at a *nonterminal*, then <ctrl>Q (and <ctrl>E) will give you the *production* menu for that nonterminal). Each entry in the coercion menu shows a possible way that nodes of the form given at the top of the menu can be transformed ('coerced') into nodes of another form. Items of the form ?*number* are pattern variables, and are used to denote the sons of the original node. Whenever such pattern variables appear in a coercion menu item, it means that after the coercion, the corresponding son of the original node (the actual son-not a copy) will be inserted in its place. Pattern variables are also used (in a similar way) for searching and replacement (to be described later).

To select a coercion from the menu, type <ctrl>q *mnemonic* <cr>. Two common uses for coercions are adding an identifier to (or removing one from) a block or **loop** statement, and adding an initialization part to (or removing one from) an object declaration or a package body.

The node that results from a coercion will retain any name that was held by the previous node.

The following diagram illustrates a coercion of a **while** statement into an **if** statement. This coercion could bc performed by moving the cursor to thc **while** statement, typing <ctrl>q, and then typing "ife" (the appropriate mnemonic). The cursor is then transformed into an if statement (with an **elsif** and/or **else** part), as illustrated.

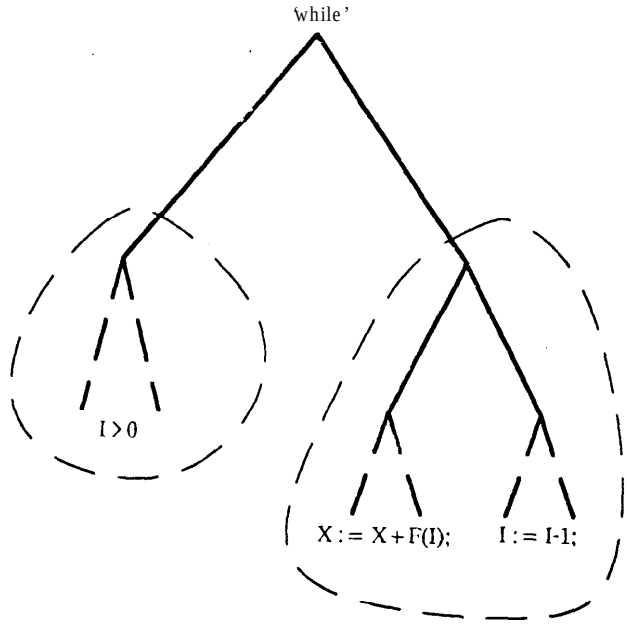
An *embedding* is a similar concept. An embedding embeds an (expanded) node as one of the sons of another node of the same (nonterminal) type. In other words, another node (of the same nonterminal type as the cursor node) is 'grown' around the original cursor node.

As an example, an embedding allows you to easily attach a label to an (expanded) Ada statement-a node representing a labelled statement is created, with its second son being the original statement. (Its first son will be an <identif ier> nonterminal-for the label.) Embeddings are most usefully applied to expressions-if, for example, you have constructed "X+Y" , and then decidc that what you really wanted was " (X+Y) *Z".

If you are at an expanded node, then you can see the embedding menu by typing <ctrl>E. The appearance of the pattern variable ?cursor in an embedding menu entry indicates where the original cursor node would be placed as a result of the embedding. To select an embedding from the menu, type <ctrl>e mnemonic <cr>.

Before:

```
while I > 0 loop
      X : =  X + F(I);
      I : = I-l;
end loop;
```



After:

```
if I > 0 then
    X : = X + F(I);
    I :=  I-l:
<ELSIF-ELSE-PART>
end if;
```

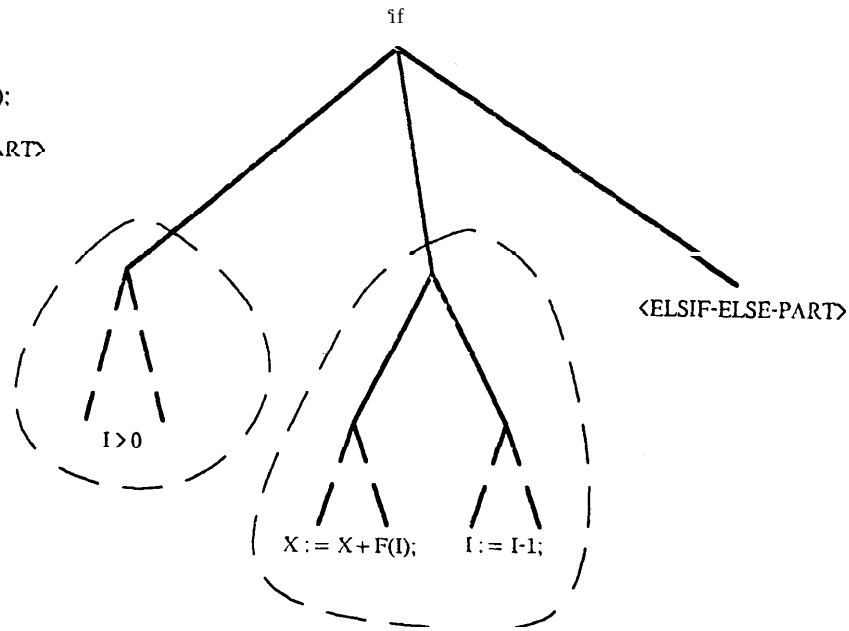


Figure 2: An example of a coercion

Patterns, and Search Commands.

The editor has powerful facilities for searching and replacement. It is possible to specify any named node as being a search pattern, and to search for occurrences of that pattern in the original tree. It is also possible to specify another node as being the replacement pattern, and to replace an occurrence of the first pattern by (an appropriate instantiation of) the second pattern.

Any (named) node can be used as a pattern, although patterns will usually be nodes that are 'standing alone'. A pattern will typically be created using <ctrl>W, and built up with productions, <ctrl>@, <ctrl>p and/or <ctrl>T (the <ctrl>@ and <ctrl>T commands will be described shortly). Patterns frequently have pattern *variable* nodes as subnodes. (Actually, pattern variable nodes can appear inside any tree, but, as will be explained below, they are really only useful inside patterns.) Pattern variable nodes are 'atomic' nodes, and are displayed as "?*id*", where id is some symbol. A pattern variable node can be created from a nonterminal node by using the <ctrl>T command. You will be asked for an 'id' for the pattern variable-this can be any sequence of (visible) characters (but not containing characters such as " (", ") ", " ' ", " ` " or "#", which all mean special things to the LISP reader!).

To initiate a search for an occurrence of a pattern (shortly we will explain exactly what it means for a node to match a pattern), use the <ctrl>w, <ctrl>f or <ctrl>s commands. In each case you will be asked for the name of the pattern to search for. These three commands differ only in *where* they perform the search. <ctrl>w searches inside the (original) cursor (including the checking of the cursor itself). <ctrl>f searches after the cursor. <ctrl>s performs the most 'thorough' search of the three-this command searches first inside the cursor, and then after the cursor if an occurrence of the pattern was not found inside the cursor.

If a search is successful, the cursor will move to the *first* occurrence of the pattern, otherwise the cursor will stay where it was originally. All searches are performed in *preorder,* which can be defined recursively as: "check the root", "search the first (i.e. leftmost) son in preorder", "search the second son in preorder", . . . , "search the last (i.e. rightmost) son in preorder". It turns out that this order of searching is 'natural', in that it corresponds to the same order that the user would naturally perceive by looking at the corresponding program text displayed on the screen. (Unfortunately, this isn't *always* true for expressions-see below.) It should be explained precisely what is meant by "searching after the cursor", in terms of the editor's internal tree representation. Consider the (ordered) sequence σ of nodes that would be visited during a preorder traversal of the entire tree (starting from the root). Let σ' be the largest suffix of this sequence that does not contain the cursor, or any of its descendants. (Note that as a consequence of this, σ' does not contain any ancestors of the cursor.) Then σ' is the sequence of nodes that will be traversed in a search after the cursor. Note that this order, too, is 'natural'.

Having performed a search once, you can repeat the search any number of times without having to retype the search pattern name each time. <ctrl>V (that's a logical "or" sign-not a "V") repeats the most recently performed search, searching inside the cursor. <ctrl>> repeats the most recent search, searching after the cursor. <ctrl>Z repeats the most recent search, searching first inside the cursor, and then after it (if the inside search

was unsuccessful). (Note that unlike the <ctrl>w and <ctrl>s commands, the <ctrl>V and <ctrl>Z commands do not check the cursor itself, because they are typically used immediately after a successful search.) To make it easier to remember these commands, note that the characters "V", ">" and "Z" appear above "W", "F" and "S" (respectively) on the SAIL keyboard.

We shall now explain what it means for a node to match a pattern. "match" can be defined recursively as follows: A node n matches a pattern *p* if the following three conditions hold:

(i) Either:

(a) n and *p* are the same sort of node. Formally, this means that n and *p* have the same *node-id*. *(node-ids* are described in section 3.) Informally, it means that n and p look exactly the same when displayed (in full) on the screen (ignoring how their sons are displayed). For example, n and *p* could both be for statements (although if the discrete range of one is preceded by "reverse', , then the discrete range of the other must also be preceded by "reverse"). Also, n and *p* could both be assignment statements, or could both bc + expressions, and so on.

or:

(b) *p* is a pattern variable of the same nonterminal type as n and, if the pattern variable *p* was previously *bound* to another node *m*, n and *m* are identical (analogous to EQUAL in LISP). (If *p* was not previously bound to another node, and *p* and n have the same nonterminal type, then p can be considered as being bound to n from here on.)

(ii) n and p have the same number of sons. (This will automatically be true if n (and hence *p)* is not a list node.)

(iii) Considering the sons (if any) of *n* and *p in order,* each son of n matches the corresponding son of *p.*

Note that, as a consequence of this rule, if p is not a pattern variable, and does not contain any pattern variables, then n matches p if and only if n and *p* are identical nodes.

For example, the assignment statement FOO := FOO + BAR; matches both the pattern ?X : = ?X + ?Y; and the pattern ?X := ?Y + ?Z;, but the statement FOO := BAR + MUMBLE; matches only the second pattern.

It should be noted that the above definition of 'match' is *assymmetric,* in that pattern variables are treated differently depending on whether or not they occur in the *pattern.* In fact, any pattern variable that occurs in the node being tested for a match (as opposed to one that occurs in the *pattern)* does not really behave as a 'variable' at all, since it matches only an identical pattern variable. This assymmetry is intentional, since it was felt that providing fu!l *unification* would be more difficult, and of only limited usefulness.

The <ctrl>Q command can sometimes be useful for creating patterns, when starting from a nonterminal node. This command is the same as the <ctrl>e command (selection of a production), except that the (nonterminal) sons of the 'skeleton' node inserted as a result of the production are each replaced by pattern variables. For example, if the cursor is at a <statement> nonterminal, then whereas <ctrl>e := <cr> will produce <NAME> := <EXPRESSION>;, <ctrl>Q := <cr> will produce ?1 := ?2;.

Problems can arise because of the fact that the editor represents expressions consisting

of associative operators (such as "+") in binary tree form. To illustrate,' consider the expression FOO + BAR + 1. Does this match the pattern ?X + 1? Unfortunately, we cannot be sure. If the expression is represented internally as "+ + FOO BAR 1" (using prefix notation), then it will match the pattern. However, if the expression is represented as "+ FOO + BAR 1", then it will not match the pattern (although the subnode "+ BAR 1" would, of course, match). For this reason, searching should be used with caution if the pattern contains expanded (or partially expanded) expressions.

Replacement Commands.

Having performed a (successful) search, you can then use the <ctrl>r eommand to specify another node as being the replacement *pattern*. This command replaces the found occurrence of the search pattern by an instantiation of the replacement pattern. As a result, any pattern variable occurring in the replacement pattern, that was previously bound as a result of the search, will be replaced by the node that this pattern variable was bound to. Note that you type the replacement command (<ctrl>r) after having first performed the search.

To illustrate this, suppose that you have found the statement FOO := FOO + BAR; using the search pattern ?X := ?X + ?Y; (and that the cursor is still there). Suppose that you wish to replace the cursor by the pattern ?X := ?Y + F (?X, ?Y, ?Z) ;, and that you have previously named this pattern "N", say. Then, after typing <ctrl>r N <cr>, the cursor *will* be replaced by FOO : = BAR + F (FOO , BAR, ?Z) ;. Now suppose that you had found the same statement using the search pattern ?X := ?Y + ?Z instead. Then, the same replacement command would replace the cursor by FOO := FOO + F (FOO, FOO, BAR);.

Once having performed a replacement (using <ctrl>r), you can repeat the search and subsequent replacement (without having to restate either the search or replacement pattern) by using the <ctrl>#, <ctrl>) and <ctrl>] commands. <ctrl># repeats the (most recent) search and replacement, searching only inside (strictly inside) the original cursor. <ctrl>) searches after the cursor. <ctrl>] searches first inside the cursor, and then after the cursor if no replacement was possible inside the cursor. (Note that these commands only perform the first possible replacement—not all possible replacements at once.)

The editor will not perform a replacement if it would lead to a syntactically incorrect construct. Also, the node that results from a replacement will retain any name that was previously held by thc node that it replaced.

Comments.

The editor's parser will accept Ada comments (i.e. preceded by "--"). Although it does not yet allow comments anywhere in your program, it does allow comments in most 'reasonable places'. Reasonable places arc bcfo:e a compilation unit, or in any place

where a declaration or statement could legally appear. In particular, nonterminals such as <statement> and <declarative-item> can be replaced by Ada comments (using the <ctrl>p command). Note, though, that any comment will be displayed on a separate line, even if it originally appeared (in source text) on the same line as a declaration or statement.

The Display.

Whenever possible, the editor will display the entire tree being edited (including the cursor) *in full.* However, if the tree being edited becomes too large to display in its entirity, then some subnodcs must be omitted, or 'elided', from the display.

Details of the algorithm that the editor uses in such cases to determine which nodes shall be printed, and which nodes shall be elided, are given in section 3. In general, though, the root of the entire tree will always be displayed, unless this is impossible. Also, should eliding be necessary, the editor will try to keep the displayed size of the cursor within preset limits, so that on the one hand the cursor is shown in enough detail, and on the other hand the context in which the cursor appears can be clearly seen. (Currently, these limits are 1/5 of the size of the screen, and 1/2 of the size of the screen.)

Two commands are provided that will enable you to (temporarily) increase the displayed size of the cursor, should you ever feel that the cursor is not displayed in sufficient detail. The <ctrl> j command will attempt to increase the displayed size of the cursor as much as possible, while retaining the same node as before (usually the root) as being the node that is displayed 'topmost'. <ctrl>J goes even further, in that only the cursor will be displayed (i.e. with none of its ancestors being displayed), should this serve to increase the displayed size of the cursor.

The <ctrl>R command will cause the screen to be rebuilt and redisplayed 'from scratch'. This command can be used as an 'escape hatch', should the screen ever get messed up in some way.

Aborting a Command.

If you mistakenly type a command that asks for more input before taking effect, you can abort it at this stage by typing <ctrl>g. This is, in fact, a LISP command, which exits to the 'top level' of LISP.

Provide that the editor is operating normally, no command should cause an 'infinite loop'. Should an infinite loop appear to be occurring, however, you can interrupt the editor (on SAIL) by typing "<esc> i". This should interrupt the editor. At this stage, you can type <ctrl>g, and then <ctrl>R, to restore the display.

Halting the Editor.

The command <ctrl>z will halt the underlying LISP system, and will therefore halt the editor as well. Should you happen to type <ctrl>z accidentally, you can resume the editor by typing "REENTER<cr>" to the SAIL monitor, and then " (TERMINAL-INITIALIZE> <cr>" and <ctrl>R, to restore the display.

3. System Architecture.

This section of the Reference Manual serves two purposes. Not only does it provide details of the architecture of the system that may be of general interest, but also, it gives further details of the implementation which (when supplemented by the separate Implementation guide) will be of use to maintainers of the system. In particular, it makes reference to many of the source code files on SAIL.

An Overview of the System.

The editor, as a whole, consists of several distinct units. However, it is possible to conceptually group these units together to form five main logical components.

Firstly, there is the *editor kernel*. This can be thought of as the 'editor proper'. It consists of the tree(s) being edited, as well as all of the code responsible for interpreting and implementing the user commands that were described in section 2. It also includes the 'high-level' display routines, which construct the *print* boxes to be output to the screen (print boxes are described later). The code for the editor kernel does not depend upon any particular operating system, or terminal.

· -The second component is the *terminal-dependent* and *I/O* part. This consists of routines responsible for opening and closing files for I/O, and also includes the lexical scanner that is used by the parser. In addition, it includes 'low-level' display routines, which take the *print box* information produced by the 'high-level' display routines in the editor kernel, and display it on the user's screen. Accordingly, these low-level routines are, of necessity, terminal depcndcnt.

Thirdly, there is the *parser*. The parse routines use the parse *table* and the node building routines (both created by the preprocessing phase) to parse input text, and produce a tree structure as output.

The final two components are not present when the editor is running, but are used to 'bring up' the editor for the first time. The *language* description files provide the editor with the necessary information about the programming language that it supports. The files contain information about both the 'external' language grammar that is used by the parser, and the 'internal' tree format that the editor uses directly.

The final component is the *preprocessing part*, which enables the language description files mentioned above to bc preprocessed into a form that can be used by the parser and the editor kernel.

The following two diagrams show how the many different modules that make up the editor can be grouped together to form the five components mentioned above. The first diagram shows the first three components, these three being the components that are present when the editor is running. The arrows indicate major control flow within the editor-an arrow from one module to another indicates that the function pointed to by the arrow is frequently called from within the first module. The second diagram shows thc language description files, and the preprocessing part. Note that the diagrams assume that the programming language supported is Ada. However, if another language were supported, thc only change would be that thc files named "ADA.*" would be different.

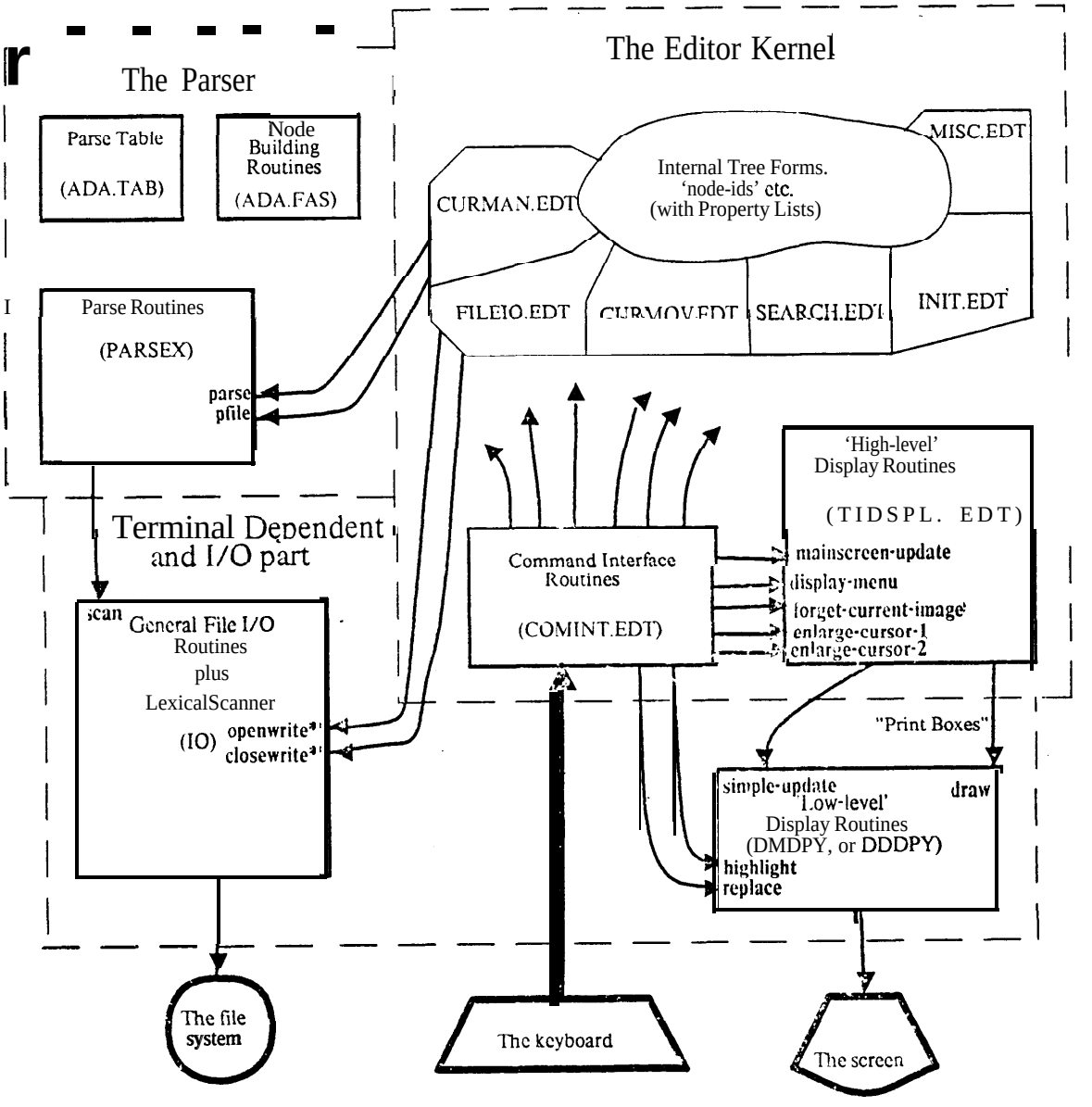


Figure 3 - The logical components of the system.

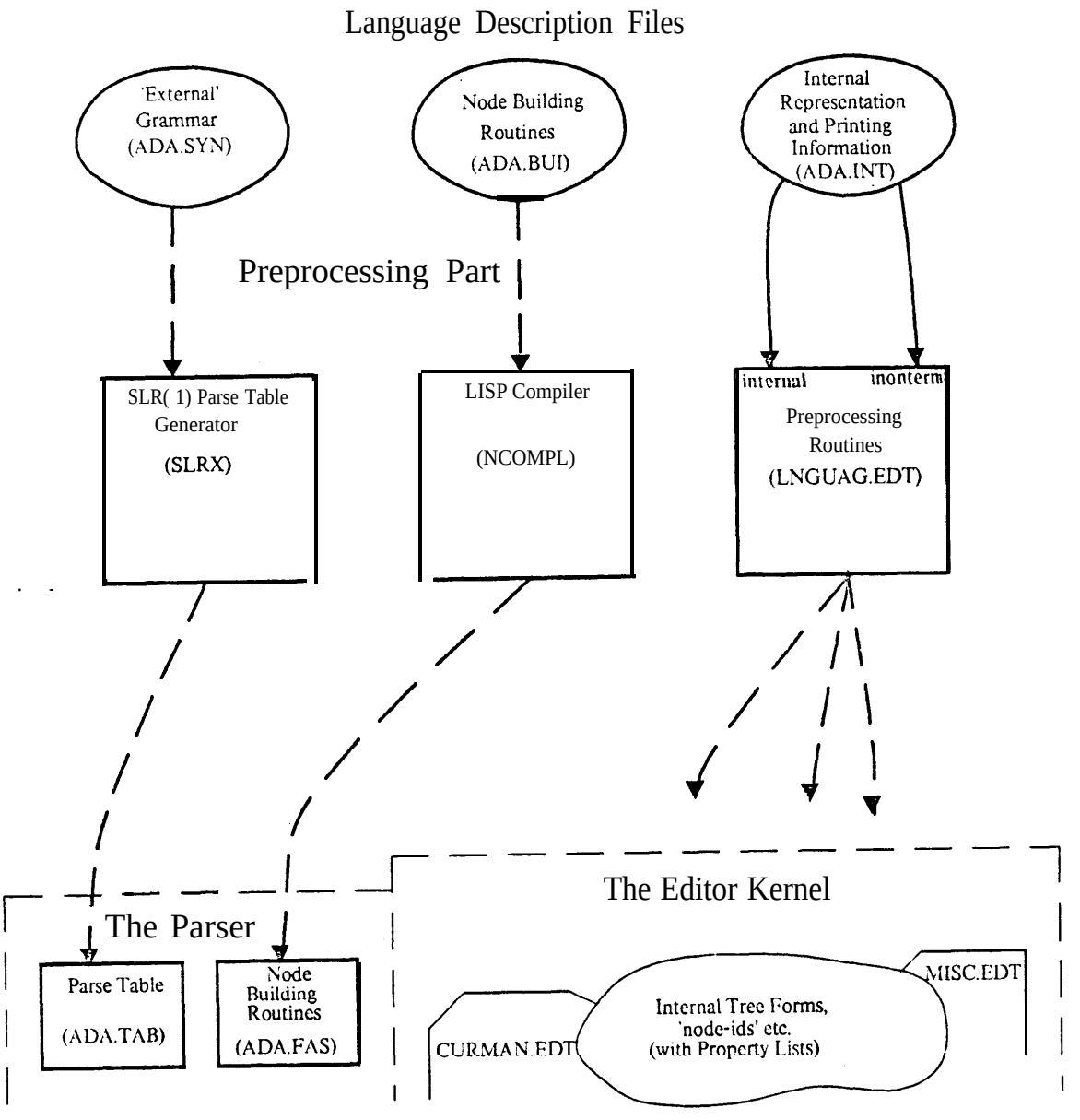


Figure4 -The Preprocessing Phase

It should be noted that the code in most of the files with extension ".EDT", is actually compiled before being run by the editor. Accordingly, the actual files that are input when the editor is set up have the extension ".FAS".

The 'nerve center' of the system is the *command interface,* the code for which can be found in the file COMINT.EDT. Every control character that forms a command to the editor is made to correspond to a LISP function, so that typing the control character causes the execution of that particular function (with no arguments).

COMINT.EDT contains the definitions of each of these *command functions.* When executed, a command function will cause the implementation of the corresponding control character command. However, the command functions do not implement the control character command directly. Instead, they invoke *auxilliary routines* that directly implement the commands. The code for these auxilliary routines can be found in the files CURMOV.EDT, CURMAN.EDT, SEARCH.EDT and FILEIO.EDT. By separating the auxilliary routines from the command interface in this way, the work that would be involved in altering the command interface is greatly simplified.

CURMOV.EDT contains code that implements the commands that move the cursor node around. CURMAN.EDT implements the commands that physically alter the cursor node in some way. SEARCH.EDT implements patterns, searching and replacement. FILEIO.EDT implements the I/O of previously constructed trees from/to a file.

The file TIDSPL.EDT contains code that maintains the main display window and the menu window, as a result of the execution of a command. It should be noted that this code does not assume any particular kind of terminal. Virtually all of the command functions in COMINT.EDT invoke the routine **mainscreen_update**. This is the main routine in TIDSPL.EDT. It checks whether or not a change needs to be made to the display, and if so, causes the invocation of appropriate low-level (terminal dependent) display routines to effect such a change. Further details of the display algorithm used by **mainscreen_update** (and its subsidiaries) are given later in this section. TIDSPL.EDT also contains other routines that are invoked by the command functions in COMINT.EDT. **display_menu** is used to display a production, coercion or embedding menu in the screen's *menu* window (which will cover the main display window). **forget_current_image** helps to implement the **<ctrl>R** command (that refreshes the display). **enlarge_cursor_1** and **enlarge_cursor_2** help to implement the display-oriented commands **<ctrl>j** and **<ctrl> J** (rcspectivcly).

INIT.EDT contains code that initializes the editor's global variables when the editor is set up. MISC.EDT contains a few miscellaneous routines. The documented source code contained in the files mentioned above (those with extension ".EDT") is given in full in the Implementation Guide.

The file containing the low-level display routines will differ, depending upon the type of terminal that is supported. For DataDiscs, the file is DDDPY. For Datamedias, it is DMDPY. The four main low-level routines are **highlight**, **replace**, **simple_update** and **draw**. **highlight** and **replace** are used whenever possible, since they do not entail redrawing the entire screen, but instead only the part of the screen where the cursor is displayed. **simple_update** puts any changes made by highlight and **replace** into effect. **draw** redraws the entire screen.

The Editor's Internal Tree Form.

Every node (including nonterminal nodes and pattern variable nodes) is represented internally in the form: (node-id *plist-atom* contents) (where "*contents*" can be empty).

node-id is an atom which gives the specific type of the node. For example, it is **!** if for an **if** statement, **!package-body** for a package body, ! and for an expression of the form "*x* **and** *y*", and so on. There is a fixed number of these *node_ids*—one for each particular type of node possible. The set of possible *node-ids* depends upon the actual programming language that is supported, of course. This set of node-ids is given (along with other information) in the language description file ADA.INT. In the current version of the editor, supporting Ada, there are 133 different *node-ids*.

For clarity, the name of each node-id begins with the character "**!**". Nonterminal nodes and pattern variable nodes have node-ids of **!!nonterminal** and **!!patternvar**, respectively. These *node-ids* begin with two "**!**"s because of their special status.

The property list of any node-id contains various properties that describe all nodes of that particular type. For example, the property list of the atom "**!** case" contain properties that apply to all nodes that have this as their node-id (i.e. all case statement nodes). These properties will be described shortly.

plist-atom is an atom (formed by **gensym**) which is unique to each particular node. That is, no two nodes will have the same *plist_atom*. The property list of a node's *plist_* atom will contain information that applies to that particular node alone. For example, if the node has a name, then it is recorded as a property on that node's *plist-atom*.

The exact form of "*contents*", which completes the representation of a node, depends upon whether or not the node is an atomic, or 'leaf' node. If the node is not atomic, then "contents" is the sequence of sons of the node, in order. If the node is atomic, then *"contents"* will be empty, or be an atom, or a sequence of atoms-this depends upon the type of node. For example, a node for the identifier "FOO" has the form: **(!** id *plist_atom* f **oo)**. Nonterminal nodes have the form: **(!!** nonterminal *plist_atom "non terminal type")*. For example, the nonterminal node "**<statement>**" has the form: **(!!** nonterminal *plist-atom* statement). Pattern variable nodes have the form: **(!!** **pat-**ternvar *plist-atom "nonterminal type" "pattern variable id")*.

As an example, consider the Ada expression "FOO **in 1..12 or** FOO **>** 100". This expression would be represented internally as:

```
(!or g0106 (!in g0109 (!id g0113 f o o )
                      (!num g0114 num1)
                      (!num g0115 num12))
           (!> g0121 (!id g0119 f o o )
                     (!num g0120 num100)))
```

Note that the **gensym** numbers that I have chosen for the *plist_atoms* are arbitrary. They will always be distinct, however.

Properties of *node-ids*.

Each *node-id* has a number of properties that describe various features common

to all nodes having this *node-id*. These properties are given (along with 'the *node-ids* themselves) in the language description file ADA.INT, and are attached to the *node-ids* during the preprocessing phase. The properties are: possible-nonterminals, *is_atomic_node*, *is_list*, template, *cmenulist*, print-format, *always_0_indents*, always-nil-context, nodes-not-below, and printable. They are described in detail below.

The property *possible-nonterminals* is'a list of all possible nonterminal types, to which nodes with this *node-id* could belong. Many *node-ids* have only one such nonterminal type-the possible-nonterminals property of the *node-id* !case is (statement), for example. On the other hand, the *possible-nonterminals* property of !and is (expression parameter choice), since an "and" expression can occur as a parameter of a procedure and function call, and also as a choice in a case statement, as well as in place of an <expression> nonterminal. Unless a node occurs as the root of a tree, however, it is always possible to uniquely determine a node's nonterminal type, by looking at the context in which it appears. The function nonterminal_types (the code for which is in the file CURMAN.EDT) performs this task.

The property *is-atomic-node* specifies whether or not such nodes are atomic, or 'leaf' nodes. For example, this property is *t* for the *node-ids* ! id (identifier), !num (number), and of course ! !nonterminal and !! patternvar. For most other *node-ids* it is *nil*. The editor makes use of this property to ensure that the cursor is always pointing to a legitimate node, and never inside an atomic node.

The property *is-list* specifies whether or not such nodes are *list* nodes. List nodes can have a variable number of sons, and the list insertion commands <ctrl>{ and <ctrl>} can be applied if the cursor is a son of a list node. If the node type in question is not a list node, then the *is_list* property of the *node-id* is *nil*. Otherwise, it is an integer, giving the minimum number of sons that list nodes with this *node-id* could have. For example, the *is_list* property of statement-list is 0, since an Ada statement list can have as few as 0 elements. On the other hand, the *node-id* ! enum-type (which is the *node-id* of a node that lists the elements in an enumeration type definition) has an *is-list* property of 1, since an enumeration type definition must contain at least one element. In this way the editor ensures that the hard deletion command <ctrl>D always preserves syntactic correctness.

The *template* property is used to specify the nonterminal types that the sons of any node with this *node-id* must have. The form that this property takes depends upon the *is-fist* property mentioned above (i.e. upon whether or not such nodes are list nodes).

If the *is-list* property is non-nit, then the *template* property is a one-element list, whose (sole) element is the nonterminal type of any son of such a (list) node. For example, the *template* property of the *node-id* ! statement-list is (statement). Recall that sons of list nodes are homogeneous, in that they all have the same nonterminal type.

If, on the other hand, the *node-id's is-list* property is *nil*, then the *template* is a list giving the nonterminal type that each of the node's sons must have. Each element of this list is the nonterminal type of the corresponding son. For example, the ***template*** property of ! := (the *node-id* for an Ada assignment statement) is (name expression). This indicates that "!:=" nodes have two sons, the first being a name, with the second being an expression.

The cmenulist property specifies the possible *coercions* that nodes with this node-id can undergo (i.e. by the use of the <ctrl>q command). If no coercions are possible, then this property is *nil.* Otherwise, the property is a list of *coercion descriptors.* There is one descriptor for each possible coercion. Each descriptor has the form: (mnemonic . "coercion map"). mnemonic is the mnemonic that will appear when the coercion menu is displayed (i.e. it is the mnemonic that the user can enter after applying the <ctrl>q command). The coercion map is a list, the **car** of which is the *node-id* of the new node that will result from the coercion. The cdr is a list of non-negative integers, which indicates how the sons of the new node will be made up from the sons of the old. A "0" indicates that the corresponding son of the new node will be a nonterminal node. A positive integer *n* indicates that the corresponding son of the new node will consist of the nth son of the original node.

As an example, the cmenulist property of !while-loop-1 (which is the *node-id* for an Ada **while** statement without an attached loop identifier) is:

```
((+id !while-loop-2 0 1 2 0)
 (loop ! basic-loop-l 2)
 (for !for-loop-l 0 0 2)
 (forr !forr-loop-l 0 0 2)
 (if !if 1 2)
 (ife !ife 1 2 0)
 (block !block 2)
 (dblock !dblock 0 2))
```

Consider, for example, the fifth element of this 1ist: (if !if 1 2) . This means that if the cursor is at a **while** statement, then the coercion command <ctrl>q if <cr> will transform the cursor node into an **if** statement. The first son of this **if** statement (i.e. the conditional expression) will be the first son (i.e. the conditional expression) of the **while** statement. The second son of the **if** statement (i.e. the list of statements following "**then**") will be the second son of the **while** statement (i.e. the list of statements following ('loop").

The *print-format* property is the most complicated property that *node-ids* have. It describes all the possible ways that nodes with this node-id can be printed.

It should be noted that when a node is printed, it is printed in a particular *context.* This context is given as a LISP atom. The default context is *nil,* and most of the time nodes are indeed printed in a context of nil. (In particular, the root is always printed in a context of nil.) Sometimes, however, nodes will be printed in different contexts, depending upon their surroundings. Consider, for example, the expression "**x+y**". If this expression does not appear inside another expression, then it will be printed in the default context (i.e. nil). If this expression appears as one of the terms of an additive operation, then it will be printed in the context "term". If, however, this expression appears as a factor in a multiplicative operation or an exponentiation operation, then it is printed in the contexts "factor" and "primary" (respectively). In such a case the expression should be printed surrounded by parentheses—the parenthesis of expressions is the main example of a situation where context is important.

The *print-format* property of any node-id consists of a list of one or more *context* options. A context option consists of a *context specifier,* followed by a sequence of one

or more print *option atoms*. Admittedly, this sounds complicated, but in actual practice the *print-format* property of most node-ids has the simple form: (((all) "print option atom")). (That is, there is usually just one context option, with "(all)" as the context *specifier,* and with only one *print-option-atom.)* More generally, though, a context specifier can take one of three possible forms:

(i) (all) . This indicates that the following *print option atoms* can be used regardless of what context the node is printed in.

(ii) (a sequence *of* one *or more contexts).* This indicates that the following *print option atoms* can be used if the node is printed in a context that appears in the aforementioned sequence.

(i) (allbut a *sequence* of one or *more* contexts). This indicates that the following print option atoms can be used if the node is printed in a context that *does not* appear in the aforementioned sequence.

A print option atom is an atom that is constructed by gensym during the preprocessing phase. Such atoms have various properties that specify, in detail, a possible way that a node with the node-id in question can be printed. These properties are described later.

For examples of print-format properties, consider the node-ids ! if and ! +. The print-format property of ! if is (((all) g0070)) . That is, an Ada if statement is always printed the same way, regardless of the context in which it appears. (Of course, one or more of the descendants of an ! if node may be elided, depending upon the space available, but the ! if node itself will always appear the same.) The print-format property of ! + is:

```
(((factor primary) g0135)
 ((allbut factor primary) g0136) )
```

This specifies that if a ! + node is printed with a context of "factor" or "primary", then it will be printed surrounded by parentheses. Otherwise, it will be printed with no surrounding parentheses.

In both of the examples above, there was only one print option atom in each context option. It is possible for there to be more than one print option *atom* in a context option. In such a case the editor will choose from amongst these different options, based upon the space available on the screen. This provides a way to specify one or more 'concise' printing options to use if space on the screen is scarce. In fact, this capability is not used in the Ada language implementation, where any necessary display 'condensation' is provided by eliding, in a standard way, appropriate elements of statement lists and/or declaration lists.

The properties *always_0_indents* and always-nil-context are used so that the display can be updated more efficiently in certain circumstances. A great many editor commands alter the structure of the cursor node in some way, but do not move the cursor. Ideally, we would like to use the low-level display routine replace to update the display after such a command, rather than have to redisplay the entire tree. In order to do this, though, we need to be sure that space taken up by the cursor node will not change, and that we can be sure of the context in which the new cursor should be printed.

A node-id's *always_0_indents* property is *t* if a node with this node-id will always occupy no more than one line on the screen. The property always-nil-context will be *t* if nodes with this node-id can always be printed in the default context of *nil.* If the *always_0_indents* property is *t* for both the old and the new cursor nodes, and if the

always-nil-context property is *t* for the new cursor node, then the editor knows that it can safely use **replace**.

The nodes-not-below property is used to speed up searching. It is a list of all node-ids that cannot possibly appear as a descendant of any node having the original node-id. Thus, if we are searching for a node with node-id *n*, then the editor will not need to continue the search inside a node with node-id m, should n appear as an element of the nodes-not-below property of m. In fact, for the Ada language implementation, each node-id currently has its nodes-not-below property set to nil-this will be changed once more of the Ada language is supported. This has the effect only of making searching a little less efficient than it could be.

The final property of a node-id is the property printable. This property is *t* if it would be okay for nodes with this this node-id to be the topmost node that is displayed on the screen, should it not be possible for the *root* of the tree to be displayed. (Of course, if it *is* possible to display the root on the screen (as is almost always the case), then the root will be displayed, even if the root's node-id has a printable property of *nil.)*

Properties of Nonterminals.

A nonterminal type (such as "identifier" and "statement") has various properties that apply to all nodes of this particular nonterminal type. Like the properties of node-ids, these properties are given (along with the nonterminal types themselves) in the language description file ADA.INT, and are assigned to the nonterminal types during the preprocessing phase. The properties given to nonterminal types are *is_nonterminal,* pmenulist, *emenulist* and *prin tname.*

The *is_nonterminal* property is set to *t* for every nonterminal type. This property is used only in the implementation of the **<ctrl>W** command (which creates a new, isolated, nonterminal node), to check that the **atom** that the user types at the keyboard is really a nonterminal type.

The *pmenulist* property specifies all possible *productions* on the nonterminal in question. The property takes the form of a list of dotted pairs. The **car** of each pair is the mnemonic for a production, and the **cdr** of the pair is the *node-id* of the node that will be formed as a result of the production. For example, the *pmenulist* property of the nonterminal type "**type-def** inition" is:

```
((int .  !integer-type)
 (enum .  ! enum-type)
 (arr .  ! array)
 (rec .  ! record)
 (priv .  ! private-type)
 (lpriv .  !limited-private-type)
 (acc .  !access))
```

The emenulist property specifies all possible embeddings on nodes with this nonterminal type. As with the *pmenulist* property, it is a list of dotted pairs, with the **car** of each pair being the mnemonic for an embedding. The cdr of the dotted pair is a list that begins with the node-id of the node into which the original node will be embedded, and is followed by a sequence of O's and l's that correspond to the sons of the new node. There will be one, and only one "1" in this sequence, and this "1" indicates where the original node is to be embedded in the new node.

The emenulist property of all but a few nonterminal types is *nil*. For an example of the emenulist property, consider the nonterminal type "name". The emenulist property of this nonterminal type is:

```
((index .  (!indexed 1 0))
 (select .  (!selected 1 0))
 (attr .  (!attribute 1 0))
 (all .(!select-all 1))
 (opsym .  (!select-opsym 1 0)))
```

Suppose, for instance, that the cursor is at some name n. Then, by applying the embedding command **<ctrl>e**, the cursor would be replaced by: "*n*<parameter-list>", "*n*. <identifier>", "*n*'<identifier>", "*n*. all" or "n. <character-string>", depending upon which mnemonic was given.

The *printname* property of a nonterminal type is used by the display routines. It is formed by 'exploden'ing the nonterminal name. For example, the *printname* property of "name" is (78 65 77 69). *printname* properties are also used in 'leaf' nodes such as identifiers and numbers—for such nodes the property is set up by the node building routines.

Print Boxes.

Information that is to be displayed on the screen is represented by a data structure called a *print* box. This representation is 'high-level', in that it is the same no matter what particular terminal is used to display the information.

Some 'static' print box information is computed once, by the preprocessing routines. An example would be the information that the reserved word "is" should follow the case-expression whenever an Ada **case** statement is displayed. Such 'permanent' display information is put on the property list of *print option atoms* during the preprocessing phase. (Details of the properties of *print option atoms* will be given shortly.)

Other, 'dynamic' print box information (i.e. information that shows what is currently displayed on the screen) is computed at runtime by the 'high-level' display routines (the code for which is in the file TIDSPL.EDT).

A print box can take one of three forms. The most fundamental type of print box has the form:

"(! string *highlighting code' sequence of FIXNUMs*')". This type of print box represents a simple string, with the sequence of FIXNUMs being the ASCII (actually the

SAIL) codes of the characters of the string. The second type of print box has the form:

"(! seq 'highlighting code' 'sequence of boxes')". This type of print box is used to represent a sequence of boxes which are to be displayed one after the other. The third type of print box has the form:

"(! indent 'highlighting code' 'indentation' 'sequence of boxes')". An ! indent box is like a ! seq box, except that the sequence is to be displayed starting on a new line, and indented by 'indentation' indentation units. (An indentation unit is currently defined to be 4 spaces.) 'indentation' can be any non-negative (or even conceivably a negative) integer. The indentation is relative to the start of the immediately enclosing ! seq or ! indent box. (It should be noted that the terms "indent" and "indentation" are used rather loosely in this manual. What I call an "indentation" is really just a skip to a new line-true indentation will occur only if the indentation code is non-zero.)

In actual fact, a 'box' in the sequence following ! seq or an ! indent can be either a true print box, or an atom (such as a plist-atom) whose print-box property is a print box. The latter scheme has the advantage that the low-level display routines can use the atom to store (temporary) properties of its own.

At present, there arc three possible highlighting codes: 0, 1 and 4. A highlighting code of "0" is the default, and indicates no special highlighting at all. A highlighting code of "1" is used for the print box that represents the cursor node. A highlighting code of "4" indicates that the print box represents some special item such as a reserved word. It should be noted that it is the job of the low-level (terminal dependent) display routines to decide how to use these highlighting codes. For example, neither the DataDisc nor the Datamedia low-level display routines will do anything special when displaying print boxes with a highlighting code of "0" or "4". If, however, a print box has a highlighting code of "1", then the Data Disc routines will display it in reverse video, and the Datamedia routines will display it 'intensified'. Neither the DataDisc nor the Datamedia routines treat the highlighting code "4" especially, because of the limited range of highlighting options available on these terminals. However, one can imagine some more sophisticated terminal being able to display code 4 print boxes in boldface, for instance, so that reserved words would be appear on the screen in boldface.

It should be noted that despite their similar appearances, print boxes are not nodes, and the atoms ! string, ! seq and ! indent are not node-ids.

Properties of *plist_atoms*.

The plist-atom of any node contains various properties that describe that particular node alone. (Recall that the *plist_atoms* of every node are distinct.) These properties are described below.

If the node has a name, then this name wi11 be given by the my-name property of the plist-atom.

Apart from the my-name property mentioned above, all of the properties on a node's *plist_atom are* used by the high-level display routines. The print-box property gives the print box, if any, of the node. If the node is currently displayed on the screen, then this property will exist (i.e. be non-nil). The converse of this is not necessarily true-it is possible for a node not to be visible on the screen, but to still have a print-box property. This can happen if the node was visible at some earlier stage.

The property currently-showing is *t* if the node is currently displayed on the screen, and nil otherwise.

The properties *maxoptions* and minoptions are each a list of print option *atoms* that describe how to display the node so as to use up the greatest and least number of lines on the screen, respectively. In the current Ada language implementation, each of these properties will always consist of a one-element list. Furthermore, the single element of the list (i.e. the print option atom) will be the same for both the maxop tions and minoptions property. The reason for this is that, as mentioned earlier, there happens to be only one print option atom in any context option of a *node-id's* print-format property.

The *maxminsize* property contains information about the maximum and minimum possible sizes that the node can be displayed in. This property is computed by the maxmin routine (after first computing the maxoptions and minoptions properties mentioned above). The property takes one of three possible forms, depending upon whether or not the node is the cursor, or has the cursor as one of its descendants.

If the node is the cursor node, then the *maxminsize* property has the form:

(cursor "the minimum *size* that the node *can* be printed *in"*
"the maximum size that *the* node *can* be printed *in")*

BY "size", we mean the number of lines that will be taken up when the node is printed, minus 1. More precisely, it is the total number of 'skips to a new line' that occur when the node is printed. expressions, for example, will always have a maximum (and minimum) size of 0. Note that all possible eliding is considered when 'minimum' sizes are calculated.

If the node is not the cursor node, and the cursor is not one of its descendants, then the maxminsize property has the form:

(notmixed *"the* minimum *size* that *the node* can *be* printed *in"*
"the maximum *size* that *the* node can *be* printed *in")*

The most complex situation occurs if the node is not the cursor, but does have the cursor as one of its descendants. In such a case we need to know the maximum and minimum size constraints not only for the node under consideration, but also for the cursor node, as well as for the collection of descendants to the 'left' of the cursor, and the collection of descendants to the 'right' of the cursor. The reason for this is that the display routines consider the size constraints for the cursor node to be more important than the size constraints of the other nodes. Also, if some ellipsis needs to be done on the nodes surrounding the cursor, then the display routines needs this information in order to bc able to assign a 'fair' number of screen lines to the nodes both to the left and to the right

of the cursor. In this case, the *maxminsize* property of the node's *plist_atom* has the form:

(mixed "the minimum size that *the node can* be printed *in*"
"the minimum size *of the whole node*"
"the maximum size of the whole node"
"the minimum *total size of the* descendants *to the* left *of the* cursor"
"the minimum *size of the cursor node*"
"the minimum total *size of the* descendants *to* the right of the cursor"
"the *maximum total size of the* descendants *to the left of the cursor*"
"the maximum size of the cursor *node*"
"the maximum total *size* of the descendants to the right *of the cursor*")

The property actualsize gives the 'size' that the node's print box (given by the print-box property) takes up when displayed on the screen. As before, 'size' means "the number of lines taken up by the node, minus 1".

Properties of Print Option Atoms.

Each print option atom (which appears inside the print-format property of a node-id) has various properties that describe, in detail, one particular way to display nodes with that node-id. The set of properties that each print option atom has depends upon whether or not such nodes are list nodes (i.e. on whether or not the node-id's is-list property is *nil*).

For list nodes, the properties are: prefix, prefix-indents, separator, separator_indents, suffix, suffix_indents, ellipsis, ellipsis_indents, context, when_empty and when-empty-indents. These properties are now described, using the print option atom g0066 as an example. This is the sole print option atom occurring inside the *print*-format property of the node-id ! statement-list.

The prefix property is a list, the car of which is a print box for the prefix of the list. That is, this print box is printed before any element of the list is printed. The cdr of the list is either nil if the first, son of the list; is not to be indented relative to the prefix, or is an integer *n* if the first son of the list is to be indented by n units relative to the prefix. The property prefix-indents is a non-negative integer giving the total number of indentations (not the size of any indentations) that occur in the prefix, plus 1 if an 'indent' was specified by the cdr of the *prefix* property. This property, which could also have been derived from the *prefix* property, is used to quickly compute the *maxminsize* constraints of such a list node. The *prefix* property of g0066 is ((! string 0)), indicating an empty string, and no indentation of the first statement in the list. The *prefix_indents* property is thus 0.

Similarly, the *suffix* property is a list, the car of which is a print box to be used for the suffix of the list. Since there is nothing (in the list node) that will follow the suffix, the cdr of the list will always be nil. *suffix_indents* plays a role analogous to that of prefix-indents. The *suffix* and *suffix_indents* properties of the print option *atom* g0066 are the same as the *prefix* and *prefix-indents* properties (respectively).

The separator and separator-indents properties are similar, except that they deal with the print box that separates adjacent elements of the list. The cdr of the separator property specifies the indentation, if any, of the element that follows the separator. The separator property of g0066 is ((! string0) . 0). That is, the separator is an empty string, but the statement following the separator is to be indented by "0" units. (i.e. it is printed starting on the next line, but with no relative indentation.) Thus, the separator-indents property is 1. Note that the separator does *not* contain a semicolon, since semicolons are considered to be a part of Ada statements, and not statement separators.

The context property is the context in which each son of the list is to be printed. The context property of g0066 is *nil* (i.e. the default context).

The ellipsis property specifies how list elements shall be elided, should that be necessary. As usual, this property is a list, the car of which is the print box for the ellipsis. The ellipsis-indents property does the usual. The *ellipsis* property of g0066 is ((! string 0 46 46 46 46 46 46)) (i.e. the string ". . . . . "). The ellipsis-indents property is 0.

Finally, the when-empty property specifies a print box to be used (instead of those specified by the *prefix* and *suffix* properties) should the list be empty. Of course, if the node-id's *is_list* property is > 0, then this property can just be nil, because in such a case it will be guaranteed that the list will never bc empty. when-empty-indents does the usual. The when-empty property of g0066 is ((! string 4 110 117 108 108 59)), which is the string "null ;". (Note that the highlighting is 'special'.) The when-empty-indents property is 0.

For non-list nodes, print format atoms have only two properties: print-fist and total-indents. The print-list property is a list consisting of alternating fillers and *son specifiers*. The first and last elements of the list are *fillers*. When the node is displayed on the screen, the items in this list are processed in order.

A *filler* has the form *(n. b .m)* . *b* is a print box to be displayed. m is either *nil* if the son given by the following *son specifier* element is not to be indented (or if the *filler* is the last element of the list), or is a non-negative integer, indicating that the following son is to be 'indented' by m units. n is an integer giving the total number of 'indents' that occur in the print box *b*, plus 1 if m is non-nil. *n* could, of course, be computed directly from *b* and m.

A son specifier has the form *(s . c)* . s is a non-negative integer. c is a print context (in particular, it can be *nil).* If *s* is non-zero, then it indicates that the '*s*th' son of the node is to be printed, in a context of c. If s is zero, then the *son specifier* just serves as a placeholder between two adjacent fillers (and thus c is irrelevant).

The property *total-indents* is the total number if (indents' that are specified in the *print-format* property.

As an example, consider the *node-id* !if. This *node-id* has a single print option atom, g0070, occurring in its *print-format* property. The *print-list* property of g0070 is:

```
((0 (!string 4 105 102 32))
 (1)
 (1 (!string 4 32 116 104 101 110) . 1)
 (2)
 (1 (!indent 0 0 (!string 4 101 110 100 32 105 102 59))))
```

That is, an ! if node is printed in the Following way: Firstly, the string "if " is printed. (Note the space after the letter "f".) Then the node's first son (i.e. the conditional expression) is printed (in a context of nil). Thirdly, the string " then" is printed. The second son (the statement list) is then printed, on a new line and indented one unit. Finally, on a new line and indented 0 units (relative to the start of the if statement), the string "end if;" is printed. The *total-indents* property of g0070 is 2.

An Outline of the Editor's (High-Level) Display Algorithm.

The. high-level display routines must determine the following:

(i) Which node is to be printed 'outermost' on the screen?
(ii) How large should the cursor node be displayed?
(iii) How large should the collection of nodes (if any) to the 'left' of the cursor be displayed?
(iv) How large should the collection of nodes (if any) to the 'right' of the cursor be displayed?
(v) Which nodes shall be displayed in full, and which nodes shall be condensed or elided?

There can be no definitive answers to these questions, of course. However, the high-level display algorithm that the editor uses is based upon the following hypotheses:

(1) Unless it is absolutely impossible to do so, the root of the tree should bc the outermost node that is displayed.
(2) If the entire* tree can be displayed in full, with no subnodes elided, then it should be displayed in full. More generally, if it is possible to completely display a node within the space available for it, then this should be done.
(3) Whenever there is a possiblity of flexibility in the displayed size of the cursor, the editor should try to keep the size of the cursor within a certain range.
(4) To indicate the result of a simple cursor movement command, nothing more than a simple change of highlighting should be required, unless this would place the displayed size of the cursor outside the desirable range mentioned above. In particular, the objects on the screen should not appear to 'move around' unduly.
(5) Suppose that a vertical list needs to be compressed by eliding some of the elements of that list. If the cursor is not an element of this list, then ellipsis should occur at the bottom of the list (so that the elements at the top of the list are still visible). If, however, the cursor is an element of the list, then ellipsis should occur at each end of the list (so that the elements adjacent to the cursor remain visible).
(6) If the space available for the nodes to the left and to the right of the cursor node needs to be restricted, then it should be allocated amongst the two sets of nodes in proportion to their sizes.

The routine rebuild_mainscreen is invoked (by mainscreen_update) whenever the main screen needs to bc redrawn. This routinc does the following:

(a) If the entire tree can fit onto the screen without any nodes being elided, then the routine maxprint is invoked, to display the root. This routine is fairly efficient, since it knows that every subnode can also be printed in full. In particular, maxprint is able to call itself recursively.

(b) The situation is more complicated if the tree cannot be displayed without some nodes being elided (i.e. if the maximum print size of the root exceeds the size of the screen). In this case the following steps are performed:

(i) *outermost-node* is found. This is the node that is to be displayed 'outermost' on the screen. This will be the root, unless this is impossible, in which case we select the 'highest' ancestor of the cursor that can be displayed, and whose node-id has a *printable* property of t.

(ii) *cursorguide* is calculated. *cursorguide* is a 'guideline' size for the cursor. See the routine **find_cursorguide** in the file TIDSPL.EDT to find out exactly how *cursorguide* is calculated.

(iv) *leftguide* and *rightguide* are calculated. These are the 'guideline' sizes for the portion of the tree to the left and to the right of the cursor node, respectively. *leftguide* is calculated by the routine **find_leftguide**. *rightguide* is found simply by subtracting *leftguide* and *cursorguide* from the size of the screen.

(v) If *outermost-node* is in fact the cursor node, then it is printed using the routine **print_cursor**. Otherwise, the routine **print_mixed** is used to print the node *outermost_* node, using the guideline sizes mentioned above.

Appendix A: Summary of Commands.

File Input Commands:

<ctrl>i	Input a previously saved tree.
<ctrl>I	Input a text file (must be a compilation unit).

Basic Cursor Movement Commands:

<ctrl>/	Leftmost son.
<ctrl>\	Rightmost son.
<ctrl>↑	Parent.
<ctrl>∩	Root.
<ctrl>←	Left brother.
<ctrl>C	Leftmost (immediate) brother.
<ctrl>→	Right brother.
<ctrl>⊃	Rightmost (immediate) brother.

Naming Commands:

<ctrl>n	Name the cursor.
<ctrl>N	Rename the cursor.
<ctrl>?	Tell me my name.

Deletion and Undeletion Commands:

<ctrl>d	'Soft' deletion.
<ctrl>D	'Hard deletion (in a list).
<ctrl>u	Undelete.
<ctrl>U	Simultaneous delete and undelete.

Insertion into a List:

<ctrl>{	Insert to the left of the cursor.
<ctrl>}	Insert to the right of the cursor.

Invoking the Parser (to Fill in a Nonterminal Node):

<ctrl>p	Parse from the keyboard.
<ctrl>P	Parse from a text file.

File Output Commands:

<ctrl>o	Output the tree being edited, in its internal form.
<ctrl>O	Output the tree being edited, in text form (plaintext or SCRIBE format).

Productions, Coercions and Embeddings:

<ctrl>E	Display the production menu if at a nonterminal, otherwise the embedding menu.
<ctrl>e	Select a production if at a nonterminal, otherwise an embedding.
<ctrl>Q	Display the production menu if at a nonterminal, otherwise the coercion menu.
<ctrl>q	Select a production if at a nonterminal, otherwise a coercion.

General Menu-Related Commands:

<ctrl>~	Scroll forwards in the menu.
<ctrl>%	Scroll backwards in the menu.
<ctrl>C	Remove a menu from the screen.

Pattern, Search and Replacement Commands:

<ctrl>T	Create a pattern variable (from a nonterminal).
<ctrl>@	Perform a production, but with pattern variables replacing nonterminals.
<ctrl>w	Search inside the cursor.
<ctrl>f	Search after the cursor.
<ctrl>s	Search inside the cursor, then after the cursor.
<ctrl>V	Repeat the most recent search, inside the cursor.
<ctrl>>	Repeat the most recent search, after the cursor.
<ctrl>2	Repeat the most recent search, first inside the cursor, then after.
<ctrl>r	Perform a replacement.
<ctrl>#	Repeat the most recent search and replacement, inside the cursor.
<ctrl>)	Repeat the most recent search and replacement, after the cursor.
<ctrl>]	Repeat the most recent search and replacement, first inside the cursor, then after.

Display-Related Commands:

<ctrl>j	Try to show more of the cursor.
<ctrl>J	Display only the cursor.
<ctrl>R	Rebuild the display.

Miscellaneous Commands:

<ctrl>c	Insert a copy of a previously named node.
<ctrl>W	Create a new (nonterminal) node.
<ctrl>t	Go to a previously named node.
<ctrl>v	Revert to the previous context.
<ctrl>g	Abort a command.
<ctrl>z	Halt the editor.

Appendix B: The Ada Implementation.

Parts of Ada that are Not Supported.

The following features of the Ada language described in [1] are not yet supported by the editor:

(i) The Ada real number facilities.
(ii) Exception handlers (although exception declarations and the two forms of the raise statement are supported).
(iii) Discriminant declarations, and record variant parts.
(iv) Derived types.
(v) Renaming.
(vi) Separate compilation facilities (using the reserved word "separate").
(vii) Representation specification and implementation dependent features.

Extensions to the True **Ada Syntax.**

Usually, the editor will only allow structures that conform to the Ada syntax given in [1]. (That is, in general the parser will not accept syntactically incorrect constructs, and it is not possible to create such constructs using productions, coercions etc.) However, there are a few circumstances where a more 'liberal' syntax is allowed. This was done in order to keep the (external) Ada grammar (in the file ADA.SYN) SLR(1), and also to allow a more useful set of coercions in some situations. The changes are:

(i) In declarative-parts, declarative items are not required to precede program components.
(ii) Indexed names (which can serve as array elements, function calls, procedure calls (when ";" is appended) and slices) can be more liberal than allowed in true Ada. In particular, the indices of such names can have the form: <choice-list> **=>** <expression>.
(iii) A procedure name, and the 'closing id' of a procedure, can be a string (e.g. **"+"**). In true Ada this is allowed only for the names and 'closing ids' of functions.
(iv) A pragma has the form: **pragma** <name>;. This is slightly more liberal (but no less restrictive) than [1] requires.

Possible Nonterminal **Types.**

Here is a list of all possible nonterminal types. Note that the only time that the user needs to actually enter **a** nonterminal type is when he or she is using the **<ctrl>W** command.

compilation-unit	(A subprogram/package specification or body.)
context-part	(A list of 'context-specifications'.)

context-specification	(A **with** or **use** clause at the start of a 'compilation-unit'.)
declarative-part	(A list of 'declarative-item-w'.)
declarative-item-+	(A declaration.)
declarative-item-list	(A list of (declarative-items'.)
declarative-item	(A declaration, but not of a package body, subprogram body or **task** body.)
entry-declaration-list	
entry-declaration	
object	(The item that follows the ":" in an object declaration. This will be a type name or an array specification.)
index-list	
index	(An array index specification.)
type-definition	
enumeration-literal	
subtype-indication	
subprogram-specification	(A procedure specification, or a function specification.)
formal-part	(The list of formal 'parameter-declarations' that appears inside a 'subprogram-specification'.)
parameter-declaration	
unit-name	(A procedure/function name-this can be a character string.)
component-part	(The part that follows the word record in a record declaration.)
component-declaration	(The declaration of a record component.)
statement-list	
statement	
elsif-else-part	(The optional **elsif** and/or else part of an **if** statement.)
elsif-list	(A list of 'elsif -items'.)
elsif-item	(A clause of the form: "**elsif** <expression> **then** ...".)
delay-statement	
select-option-list	
select-option	(A choice in a selective wait statement.)
select-alternative	(Like a 'select-option', except that it does not include '**when** clauses'.)
accept-statement	
entry-call	
generic-formal-part	(The list of 'generic-parameter-declarations' for a generic subprogram or package declaration.)
generic-parameter-declaration	
generic-type-definition	(The definition of a type that is given as a generic parameter .)
case-list	(A list of 'case-items'.)
case-item	(A branch of a case statement.)

choice-list	
choice	
identifier-list	
identifier	
name-list	
name	
parameter-list	
parameter	(A procedure/function actual parameter, or an array index.)
discrete-range	
character-string	
expression	

Appendix C: References.

[1] *Reference Manual for the Ada Programming Language.* Proposed Standard Document, United States Department of Defense, July 1980.

[2] Teitelbaum, T., Reps, T., *The Cornell Program Synthesizer: A Syntax-Directed Programming Environment.* CACM, Volume 24, Number 9, pp 563-573, September 1981.

[3] Wegner, P., *Self-Assessment Procedure VIII.* CACM, Volume 24, Number 10, pp 647-677, October 1981.