NPS52-81-014

NAVAL POSTGRADUATE SCHOOL

Monterey, California

THE AUTOMATIC GENERATION OF SYNTAX DIRECTED EDITORS

Bruce J. MacLennan

October 1981

Prepared for:

Naval Postgraduate School
Monterey, Ca. 93940

NAVAL POSTGRADUATE SCHOOL
Monterey, California

Rear Admiral J. J. Ekelund
Superintendent

D. A. Schrady
Acting Provost

The work reported herein was supported by the Foundation Research Program of the Naval Postgraduate School with funds provided by the Chief of Naval Research.

This report was prepared by:

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: NPS52-81-014	2. GOVT ACCESSION NO.
3. RECIPIENT'S CATALOG NUMBER	
4. TITLE (and Subtitle): The Automatic Generation of Syntax Directed Editors	5. TYPE OF REPORT & PERIOD COVERED: Technical Report
6. PERFORMING ORG. REPORT NUMBER	
7. AUTHOR(s): Bruce J. MacLennan	8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Naval Postgraduate School, Monterey, CA 93940	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: 61152N;RR000-01-10, N0001481WR10034
11. CONTROLLING OFFICE NAME AND ADDRESS: Naval Postgraduate School, Monterey, CA 93940	12. REPORT DATE: October 1981
	13. NUMBER OF PAGES
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)	15. SECURITY CLASS. (of this report): UNCLASSIFIED
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Syntax Directed Editor, Language Oriented Editor, Structure Editor, Translator Writing System, Parser Generator, Syntax Directed Editor Generator, Two Dimensional Language, Programming Environment, Table Driven Parser, Table Driven Editor.

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

A syntax directed editor is an editor oriented towards a particular language. This paper describes a general table-driven syntax directed editor and an algorithm for automatically generating a syntax directed editor for a language from a description of that language. Aside from the convenience of a syntax directed editor, it is also a very efficient parser. No syntactic error recovery is required since the editor does not permit the user to make syntactic errors. Some of the implications of syntax

directed editors for data structure manipulation and two dimensional languages are briefly discussed.

The Automatic Generation of Syntax Directed Editors

B. J. MacLennan. 80/10/29.

1. Introduction

Recently many programmers have discovered the power and convenience of a syntax-directed editor (also called a language-oriented editor or a structure-editor). A syntax-directed editor is specially designed for creating and modifying programs in one language. This is in constrast to a conventional editor, which treats a file as an amorphous sequence of lines or characters. Since a syntax directed editor is tailored to a specific language, it can be much "smarter", that is, it can incorporate knowledge of the structure of the language. For instance using an SDE (syntax-directed editor) it is simple to get from the then part of an if statement to the corresponding else part. This can be difficult with a conventional editor, since the else may be any distance from the then and there may be any number of other if statements (each with their own then) nested between. Other advantages and implications of SDEs will be discussed later.

SDEs are not a new invention. One of the earliest (1970) is the ECL "list structure editor" [2], which made use of the fact that EL1 programs were parsed on entry into a LISP-like list structure. Henceforth all editing was performed directly on this list structure. When a user wished to display a part of his program text, it was "unparsed" from lists back into EL1 source

text. More recent examples of SDEs include those in Smalltalk [13], the Cornell Program Synthesizer [10], Interlisp [11, 12], Gandalf [8], and others.

This paper describes an algorithm for automatically generating an SDE for a language from a description of that language. More precisely, it describes a language-independent algorithm that works with a straight-forward encoding of the grammer to perform syntax-directed editing. Aside from the convenience of an SDE, we find that the SDE is a very efficient parser and that the grammer can be encoded with a minimum of effort. Hence we seem to have most of the advantages and few of the disadvantages of a conventional parser.

2. Example Session

So that the reader may better understand the operation of the system, we present in this section a short example of a session with an SDE. Suppose the following program has been entered into the system:

```
i := 1;
   ^
found := false;
while (i≤n) and not found do
   i := i+1;
if t = n then <stat> [else <stat>];
```

In this example, we wish to replace the statement i:=i+1 with

```
if key = List[i] then
   found := true
else i := i+1;
```

The cursor, ^, indicates that i:=1 is the current node, or focus of attention, of the editor. By pressing the right and left arrow keys on the terminal (→, ←) we can move forward or backward in any sequence. Therefore, by pressing → twice we can advance our position to the while loop:

```
i := 1;
found := false;
while (i≤n) and not found do
   ^
   i := i+1;
if i=n then <stat> [else <stat>];
```

From this position we can treat the while statement as a whole. For instance, we could delete it, or move it elsewhere in the program. In this case however, we don't wish to treat the while as a whole; rather we want to change its do part. That is, we wish to deal with the substructure of the while. This is accomplished by pressing the down-arrow key (↓), which positions us at the left most of the while's components. (Observe that the while has two components, '(i≤n) and not found', and 'i:=i+1'.) The result is:

```
i := 1;
found := false;
while (i≤n) and not found do
             ^
   i := i+1;
if i=n then <stat> [else <stat>];
```

Since we really want the second component of the while (i.e. its body, or do part), we hit the → key once.

```
i := 1;
found := false;
while (i≤n) and not found do
   i := i+1;
     ^
if i=n then <stat> [else <stat>];
```

Since we wish to replace this statement, we will press the delete key (DEL) to remove it. The result is:

```
i := 1;
found := false;
while (i≤n) and not found do
   <stat>
     ^
if i=n then <stat> [else <stat>];
```

Notice that in place of the deleted statement, '<stat>' has appeared. This indicates the syntactic category expected here,

namely a statement.

All of the commands executed so far are universal, i.e., they apply to a program in any language. The commands used to enter a particular statement (if in this case) are peculiar to the language and are indicated in the grammar. In this case we will assume the 'i' key indicates an if-statement. After pressing this key the display will show

```
i := 1;
found := false;
while (i≤n) and not found do
   if <exp> then <stat> [else <stat>];
         ^

if i=n then <stat> [else <stat>];
```

Notice that the cursor has been positioned ready to enter the first component of the if, the condition. Since this is the only line of the display that will change for a while it is the only one that we will show. For the condition, we wish to test for equality, so we push the '=' key:

```
if <exp> = <exp> then <stat> [else <stat>];
    ^
```

The first expression is a variable name. Since these are so common, they are produced by the space bar:

```
if <chars> = <exp> then <stat> [else <stat>] ;
     ^
```

The '<chars>' indicates that we may enter any string of characters, terminated by a carriage return. We enter 'key', carriage-return:

```
if key = <exp> then <stat> [else <stat>];
                ^
```

Notice that the cursor has automatically advanced to the next free slot in the structure.

Next we type '[', indicating that a subscripted variable is requested:

```
if key = <id>[<exp>] then <stat> [else <stat>];
              ^
```

and the space-bar, 'List', carriage-return:

```
if key = List[<exp>] then <stat> [else <stat>];
                  ^
```

and then space, 'i', CR (carriage-return):

```
if key = List[i] then <stat> [else <stat>];
                             ^
```

Now, we wish to enter the assignment 'found := true'; assignment statements are requested by the 'a' key:

```
if key = List[i] then
    <var> := <exp>   [else <stat>];
      ^
```

We enter space, 'found', CR:

```
if key = List[i] then
          found := <exp>   [else <stat>];
                   ^
```

Next is the 't' key, for 'true':

```
if key = List[i] then
          found := true   [else <stat>];
                          ^
```

It should be obvious how this process continues. To completely enter the if statement requires the following keystrokes. ('_' indicates space-bar and '%' indicates carriage-return.)

```
i=_key%[_List%_i%a_found%ta_i%+_i%#1%
```

It can be observed that the order of keystrokes has a prefix flavor about it. This is because this method of program entry is completely top-down - even to the expression level. It is quite natural, however, when adequate feedback is provided through the display. It is interesting to note that the above command sequence requires 37 key-strokes, while the usual method of typing this if statement requires 43 or more:

```
if_key=List[i]_then_found:=true_else_i:=i+1
```

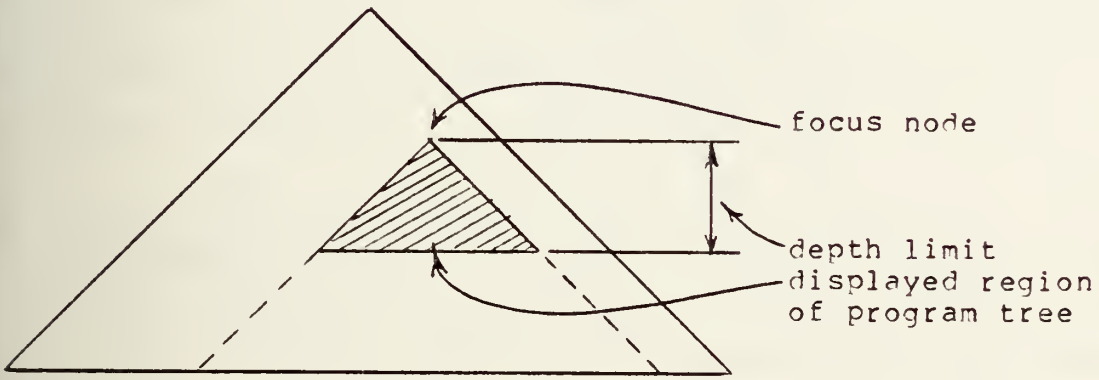
Some other aspects of the interface should be discussed. It will be recalled that we deleted the statement 'i:=i+1' and then inserted the if statement. This required reentering 'i:=i+1'. While this is not important in this case, it could have been if this statement had been much larger. To avoid this, instead of deleting the node 'i:=i+1' we could have "grabbed" it with the "G" key. Grabbing a node deletes it from the program, but saves it where it can later be accessed by a "put" command ('P' key). Thus, instead of entering

```
a_i%+_i%#1%
```

We would just enter 'P', a savings of 10 keystrokes even in this simple case. There is a wide repertoire of similar language-independent editing commands.

One final aspect of the human interface must be discussed: the display. It is intended that the user have all the information relevent to his editing task upon the screen. While Smalltalk-like windows [3, 11] may help him to view several regions at once, the finite spatial resources of the display require that detail be supressed when not needed. To do this, there are two constraints on the region of the program tree which is displayed. The display (or a window, in a multi-window system) represents the current focus of attention of the programmer. The programmer can <u>focus</u> the display on any node in the tree,

then only the subtree rooted at that point is seen on the screen. This focus may be anywhere from the root of the tree, which causes the entire tree to be displayed, to a single statement, or variable. The second parameter which controls the display is the depth limit. All structures of the program more deeply nested (from the focus node) than the depth limit are suppressed and shown as '...'. The intent is that the entire subtree rooted at the focus-node will fit on the screen. The programmer can alter both the focus and depth limit. These parameters can be visualized:



As an example, if we set the depth limit very low, and displayed our previous example, we would see:

```
i := 1;
found := false;
while ... and ... do
        if ... then ... else ...;
```

if ... then ... [else ...];

3. The System

To better understand the SDE it is necessary to understand the system in which it is embedded. This is diagramed in figure 1.

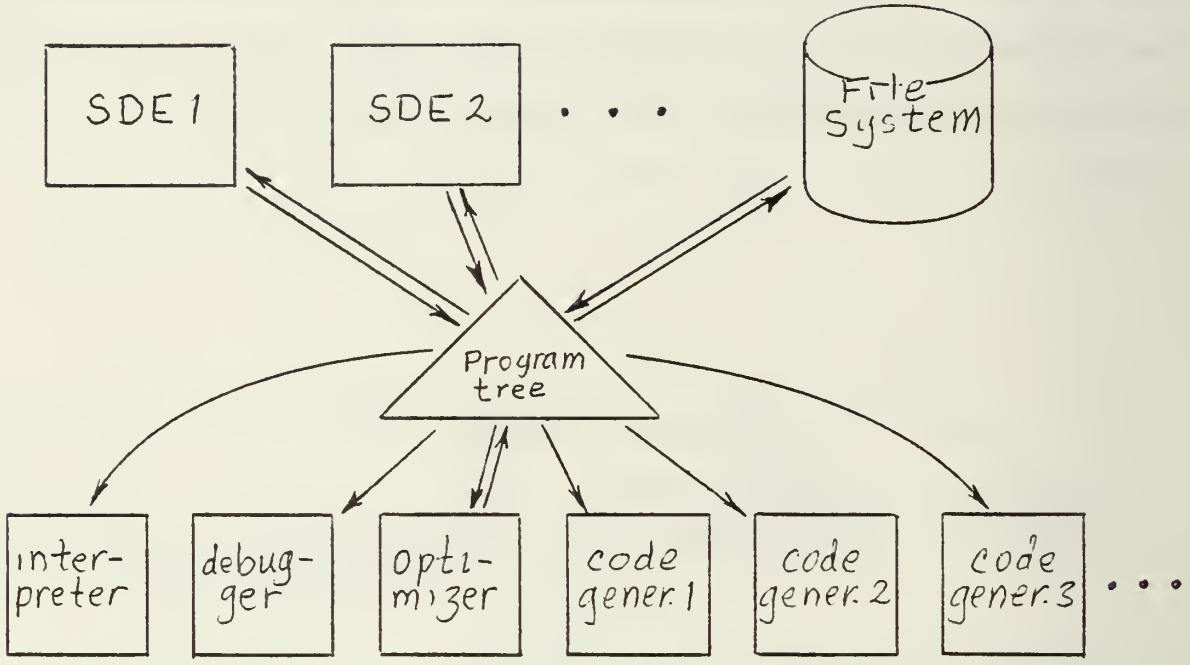


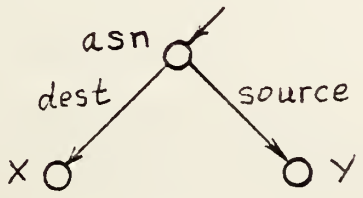
Figure 1. The System

We can see that the program-tree is the central data base through which all the tools interface. (There can, of course, be many programs in the data base.) The SDE's are used for constructing, modifying and displaying the program tree. The code generator (possibly several for various machines), interpreter, and debugger are used to execute the program tree. Using one universal format for the program tree accomplishes several things. By being machine independent, it allows multiple

compatible code generators for one language. This permits programs to be more easily transferred from one target machine to another. For instance, they can be debugged on a large machine and then transferred to a smaller machine. Secondly, by using one program tree format for several SDE's (and hence several languages) a limited facility for language translation is provided. For instance, a program might be entered through the Pascal SDE and displayed through the Ada SDE. Of course there are limits to this simple form of translation: It can only be done for programs in the semantic intersection of the two languages. Nevertheless, many language do have a great deal of their semantics in common, so it is possible to design a program tree format that accomodates many of them. This is discussed in more detail in the next section.

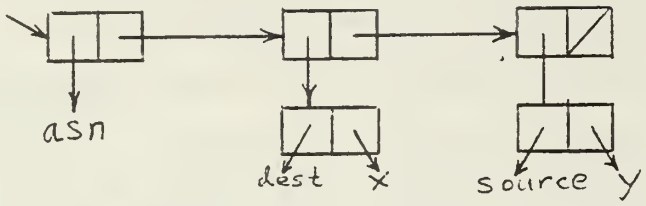
4. The Program Tree

The program tree is just a data structure which encodes as a tree the abstract form of a program. Several of the characteristics of the total system have encouraged a particular representation of this tree. Consider a node such as:



This represents an assignment statement, where the 'dest' branch leads to the destination, and the 'source' branch leads to the source, the nodes x and y in this case. This node is represented

as a tagged a-list, that is, a list of the form:



In LISP notation this is

```
(asn (dest.x) (source.y))
```

In general a tagged a-list has the following form:

```
(tag  (atr1.val1)  (atr2.val2) ... )
```

where atri is the name of the i-th attribute and vali is the value of the i-th attribute. The first element of the list is the tag, which tells us the sort of node we are dealing with. It can be thought of as a data type. If we delete the tag from a tagged a-list we are left with a conventional a-list, or association list [6]:

```
((atr1.val1)  (atr2.val2) ... )
```

To find the value of i-th attribute, atri, in node n, we can use a function assoc which scans down the list looking for atri. Specifically:

```
assoc [atri, n] = (atri.vali)
```

Some LISP systems allow a-lists to be represented by hash tables to decrease lookup time. While this is not necessary for the small a-lists that occur in the program tree, it is desirable for

ome of the other uses of a-lists, discussed later.

There are several advantages to using tagged a-lists to epresent the program tree. First of all, the offspring are tored in a position independent manner, which allows the ffspring of the node to be created in any order. This is con- enient for two reasons: First, use of an SDE permits one to reate the components of a structure in any order (although it is ptimized for left to right entry). Second, it permits a flexi- le, open-ended structure for nodes. This is particularly impor- ant for an experimental system, in which all attributes needed n a node may not be known.

Some attributes may not necessarily correspond to program omponents. For instance, some languages may permit both left- ard and rightward assignment operations:

d←s and s→d

hese will presumably be encoded as the same tagged a-list (TAL):

(asn (dest.d) (source.s))

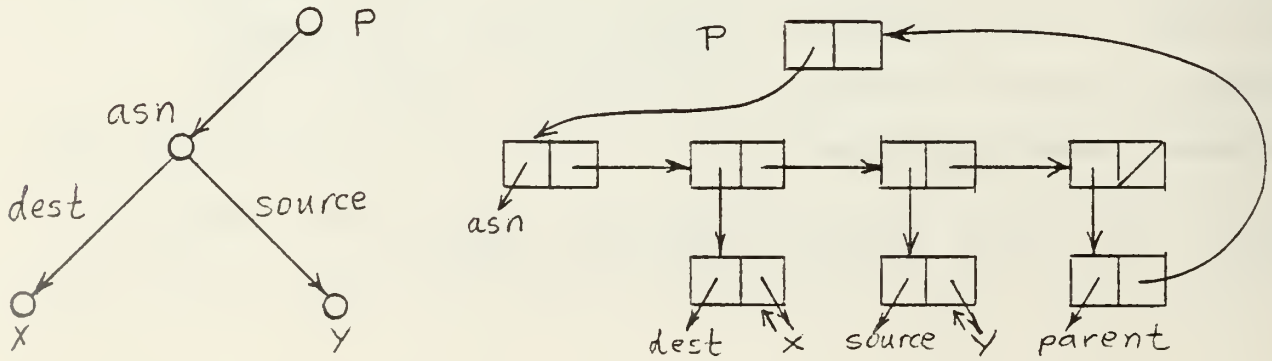
owever, when the user displays this node it will usually be best f it's seen as it was entered: either leftward or rightward. his problem is solved by adding an additional attribute, dir, hat indicates the direction, e.g.

d←s => (asn (dest.d) (source.s) (dir.l))

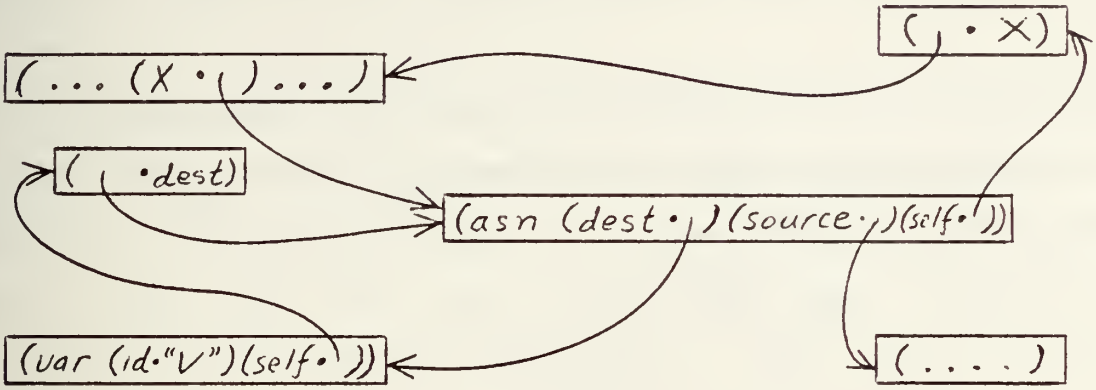
s→d => (asn (source.s) (dest.d) (dir.r))

This can be used by the display routines to show the assignment properly. Any program to which the direction is irrelevant, such as the interpreter, will not query the 'dir' attribute and thus will be unaware that it's present. In this manner any amount of additional information can be added to nodes without interfering with programs that don't need that information. In addition to attributes like 'dir' this extra information might include implementation suggestions (provided by the programmer or an optimizer) or even documentation.

For editing and program entry purposes it is necessary to move about within the program tree. The four motions allowed are those provided by the four arrow keys (→ ← ↑ ↓). The down arrow moves to the first offspring (in source order) of the current node; the right and left arrows move to the left and right siblings, respectively; and the up arrow moves to the parent of the current node. This last operation requires a method of obtaining the parent of a node. There are many ways of doing this; one of the most obvious is to have a 'parent' attribute in each node, that points to the parent. For example:



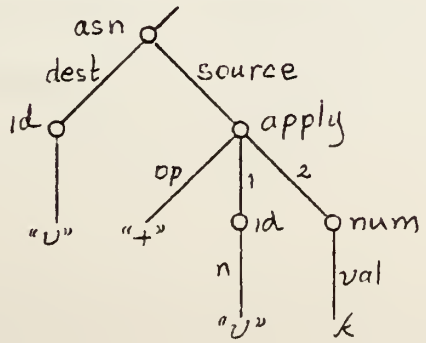
Due to the method used to address nodes (described in sec-
on 8), the method used in this system makes use of a 'self'
tribute which gives the parent node and attribute of each node,
e.



ere is another attribute that is found in almost all nodes: the
yntax' attribute. This attribute refers to the rule in the
ammar that generated that node; it is described further in sec-
on 7.

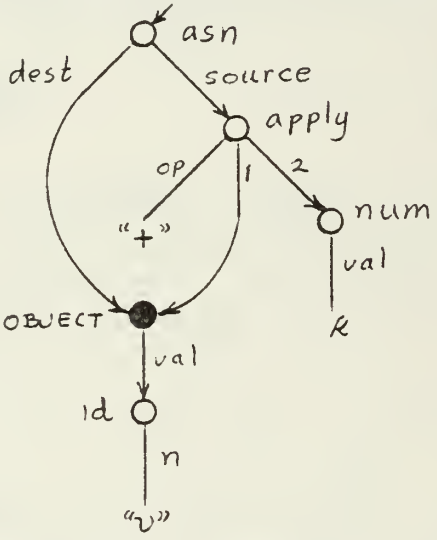
Since the program tree is a tree, it can be represented in a
npact form as prefix bytes for storage in the file sytem. It
only converted to tagged a-lists for editing.

The one exception to this use of trees is the sharing of
les. This is required for language constructs such as ':+'
so written '+:=' and '+='). The meaning of 'v:+k' is 'v:=v+k'.
corresponding program tree must "logically" have two
tances of 'v'. One way to do this is to actually have two
tances of v, e.g.



This creates problems in editing since it is possible to cause the tree to become inconsistent by altering one instance of v but not the other. This could result in a structure that couldn't be unparsed.

The alternative is to allow sharing in a specially restricted way: only object nodes are allowed to be shared. For instance, v:+k would be represented:



Since there is only one copy of v, there is no consistency problem in editing. This does mean the program "tree" is no longer a tree and that it can no longer be represented in prefix form on backing store. However, the fact that object nodes are the only nodes that can be shared allows them to be handled specially.

5. Grammatical Notation

The part of an SDE that varies from language to language is the specification of the grammar and the translation rules for

that language. The form of this specification is a simple translation grammar, with extended BNF used for analysis and with node building templates used for synthesis. The details of the notation are based on the Argot translator writing language [4, 5].

The grammar in the Appendix (which is the grammar for an Algol Subset) will be used for the following examples. Notice that the grammar begins with a name for the language (MiniGol in this case), and the name of the goal symbol ('block' in this case) and is followed by a sequence of rule (or non terminal) definitions.

Consider the rule for assignment statements:

<assignment>: % <var>_:=_ <exp> ⇒ asn (D:<var>, S:<exp>);

This corresponds to the BNF rule:

<assignment> ::= <var> := <exp>

There are several obvious differences: The BNF expresses analysis (i.e. our '% <id>_:=_<exp>') but not synthesis (our 'asn(D:<var>, S:<exp>)'). The reader will notice that our notation encludes an extra terminal, viz., "%", that does not appear in the BNF. The "%" is a carriage-return or newline character. This is formatting information for the display processor and indicates that each assignment statement is to be displayed on a new line. The set of required formatting requests has not been

delimited and probably depends on the type of terminal in use. We can envision commands for newlines, tabbing and vertical alignment so that clearly formatted program displays can be produced. More ambitious two dimensional output is discussed later.

The inclusion of formatting information in the "syntax" of a language may seem unusual, but in our notation, the lefthand (or analysis) side of a rule is not so much a specification of what the programmer will type as what will be seen. Thus, in a traditional batch-oriented compiler the user types

```
i_:=_i+1
```

which agrees pretty well with the syntax:

```
<var> := <exp>
```

In an SDE, however, the user types

```
a_i%+_i%#1%
```

and sees on the screen

```
i := i + 1
```

Thus the expression

```
% <var>_:=_<exp>
```

is primarily used to determine how an 'asn' node is displayed.

Where does the 'a' command, which calls for the creation of an assignment statement, get defined? We can see this in the rule for statements:

```
statement:  'b' <block>
          | 'a' <assignment>
          | 'i' <if stat>
          | 'w' <while>;
```

This is an example of an alternation. Whenever the user is in a state where a 'statement' is expected the four alternatives shown above will be available. One of these can be selected by typing one of the command characters 'b', 'a', 'i' or 'w'. If 'a' is typed then an unititialized 'asn' node will be created and

```
<var> := <exp>
   ^
```

will be displayed on a new line. If the user types a command that is not allowed by the current alternation, then an error indication is produced.

The construction of nodes is indicated by the right hand (or synthesis) side of rules. Consider again the rule for assignments:

```
<assignment>:  % <var>_:=_<exp> => asn(D:<var>, S:<exp>);
```

The synthesis side, 'asn(D:<var>, S:<exp>)', says that this rule generates an 'asn' tagged node with two attributes, called 'D'

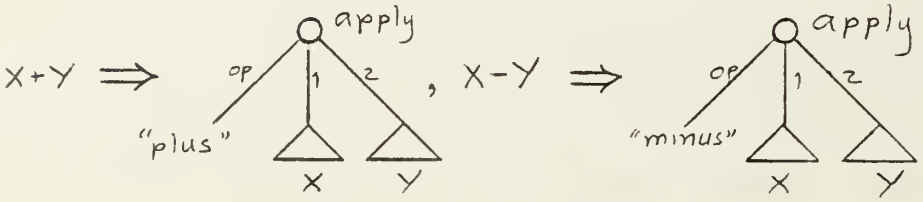
and 'S' (for 'destination' and 'source'). The D attribute will be a node constructed according to the '<var>' rule and the S attribute will be a node constructed according to the '<exp>' rule.

The definition of an expression illustrates several ideas:

```
<exp>: '+' <exp>+<term> => apply(op:"plus",1:<exp>, 2:<term>)
     | '-' <exp>-<term> => apply(op:"minus", 1:<exp>, 2:<term>)
     | '' <term>;
```

Here we see an alternation in which the command (or key) of the last alternative is empty, i.e. ''. This is interpreted in the following way: if a command character (for example '*') is entered when the editor is expecting an <exp>, then it will be compared against '+' and '-'. If it is neither of these (as is the case in our example), the editor will check to determine if it can be processed by a '<term>' (which it will be in our example, see the definition of '<term>'). This "chaining" of rules can occur to any depth.

Another facility is illustrated by the rule for '<exp>': notice that both the '+' and '-' commands generate an 'apply' node. The difference is the operation applied in each case:



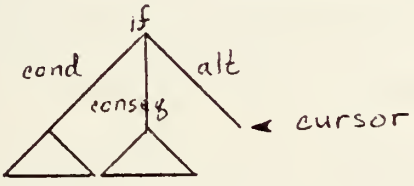
An attribute value that is a constant, that is, that isn't the value generated by another nonterminal, can be defined by "attribute: value".

A number of modifiers can be appended to non-terminal names. The simplest is the question-mark, indicating optionality, which we see in the 'if-stat' rule:

```
<if-stat>:  %if_ <relation> _then <statement><else-part?>
                => if(cond:<relation>, conseq: <statement>,
                      alt:  <else part?>);

<else-part>:  %else_ <statement>;
```

The 'if' node generated by the '<if-stat>' rule has the following form:



The 'alt' attribute can be in one of three states of definition. First, it can be open, which is the state any attribute is in before it's created or after its deleted. Second, it can be defined, which is the case if some statement has been entered as the <else-part>. Finally, it can be closed, which is the state when the user has elected to have no <else-part> to this statement. If the current node is open, as indicated in the above diagram, it can be closed through a carriage return command to

the editor. For instance, to enter

```
if i=0 then i:=1;
```

we would type "i_i%=#0%a_i%#1%" and see

```
if i=0 then
     i=1  <else-part?>;
               ^
```

on the screen. An else part could now be entered directly, but to suppress it we type '%' and see:

```
if i=0 then
     i=1;
          ^
```

If we later wish to add an <else-part>, we can press the '%' key, which opens the closed node. The display will show:

```
if i=0 then
     i=1 <else-part?>;
               ^
```

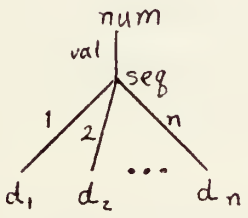
as before. Thus the carriage return toggles a node between open and closed status. One new feature can be seen in the definition of <else-part>: there is no synthesis part. A rule of this form is called an identity rule because the value returned by the rule is the same value as that returned by the one non-terminal in its analysis part.

The open/closed dichotomy is also used to control the construction of sequences. These are indicated by the '*', '+' and '...' modifiers on non-terminals.

The '+' modifier, which means one or more repetitions, appears several places in the example grammar. The simplest is the 'number' rule.

```
<number>:  <digit+> => num(val:<digit+>);
```

The node that will ultimately be generated by this rule has the form:



or, in Lisp notation:

```
(num (val.(seq (1.d1)(2.d2)...(n.dn)) ))
```

The di are the (one or more) digits allowed by the <digit+> in the rule. When this rule is entered, we see

```
<digit+>
    ^
```

displayed on the screen. If we then enter the digits "512" we will see

```
512 <digit+>
    ^
```

Notice that the display is prompting us to enter more digits. That is, the sequence is still open. We can close it by typing a carriage return, in which case we see

```
512
  ^
```

If we wish to reopen this sequence, say to add the digit '0', we can use the carriage return again.

entered	display
	512
%	512 <digit+>
0	5120 <digit+>
%	5120

The remaining non-terminal modifiers are simple variations of the '+'. The '*' modifier (Kleene star) indicates zero or more repetitions of the non-terminal. This can be seen in the rule for blocks:

```
<block>:  %begin_ <decl*><statement;...>_end
     => block(head:<decl*>, body:<statement;...>);
```

Use is exactly like '+' except that a closing an empty sequence is permitted for '*' but not for '+'.

The above rule also contains an example of delimited repetition:

<statement;...>

This means: a sequence of one or more <statement>s separated by semicolons. The protocol for entering these sequences is the same as for '+'; the display format is different. Suppose a function invocation is defined:

<func-call>: <id>(<exp,...>) => .....;

and we wish to enter "f(X, Y-1)". Here is the command sequence.

commands	display
_f%	f(<exp>,...) ^
_X%	f(X, <exp>,...) ^
-_Y%#1%	f(X, Y-1, <exp>,..) ^
%	f(X, Y-1) ^

Of course, another carriage return would reopen the sequence.

Some additional language-independent facilities are provided for editing sequences. These include the ability to extract or delete a subsequence of an existing sequence, to insert such subsequences within another sequence, and to add new elements within sequences. These work on any sequences, whether they represent sequences of statements, actual parameters or characters in an identifier.

One final modifier must be discussed, the prime. This does not alter the meaning of a non-terminal, it is used just to distinguish multiple occurences of the same non-terminal. An example is in the rule '<relation>':

```
<relation>:  <exp><relop><exp'>
     => apply (op:<relop>, 1:<exp>, 2: <exp'>);
```

This completes the discussion of the grammatical notation. We should add that the fact that parsing is driven by commands from the user means that there are very few restrictions on the class of grammars usable. For instance, they can be either left or right recursive. There is no need for look ahead. This greatly simplifies the parsing process, since the user is driving the parse through the commands entered.

6. Processing the Grammar

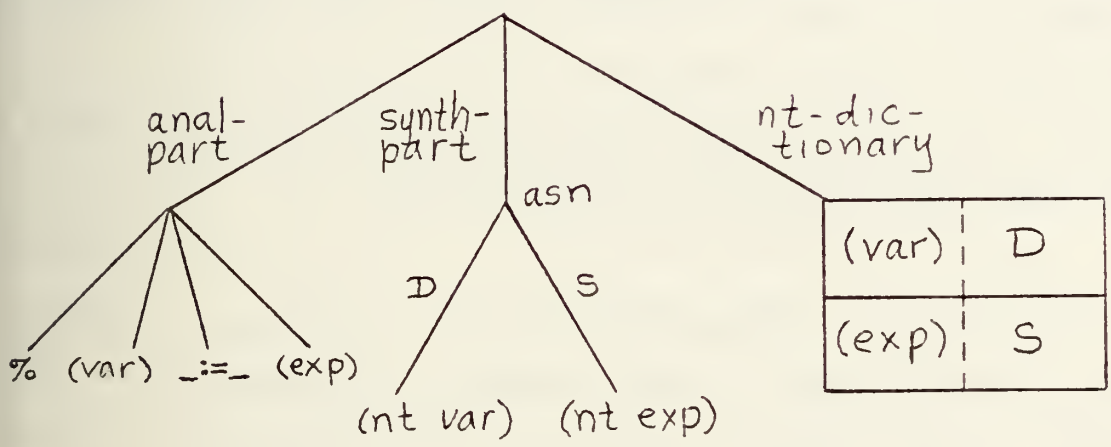
In this section we will discuss the translation of the grammar into the internal form used by the universal SDE. The grammar as a whole is translated into an a-list that associates non-terminal names with the data structures representing their definitions. These latter structures are easily understood through examples. Consider the rule for assignments:

```
% <var> _:=_<exp> => asn(S:<exp>, D:<var>);
```

The data structure representing this, in Lisp notation, is:

```
( ("%" (var) "_:=_" (exp))
     ( asn (D. (nt var))  (S. (nt exp)))
     ( ((var). (D))
       ((exp) . (S))  )   )
```

i.e.



At the outermost layer, we can see that a rule A => S is represented by a three element list (A S D), where D is the "non-terminal dictionary". The analysis side is just a list of the representations of the terminals and non-terminals. Non-terminals with affixes are translated in accord with these exam-

ples:

```
<else part?>      =>    (nt else_part ?)
<exp'>            =>    (nt exp ')
<exp' *>          =>    (nt exp ' *)
<statement;...>   =>    (nt statement : ";")
```

The synthesis side is represented by a fragment of program tree in which non-terminals are represented by lists tagged 'nt'. The third part of a rule is the non-terminal dictionary, which must be generated by the grammar processor. This is an a-list which associates each non-terminal appearing in the rule with the path for reaching the corresponding subtree. The use of the non-terminal dictionary is described in sections 7 and 8.

It remains to discuss the representation of alternations. These are represented as tagged a-lists so that the appropriate rule can be selected, given a command character. For instance, the rule for statements:

```
        'b'  <block>
    |   'a'  <assignment>
    |   'i'  <if stat>
    |   'w'  <while loop>
```

is represented:

```
(ALT    ("b".(((block)) nil ))
        ("a".(((assignment)) nil ))
        ("i".(((if_stat)) nil ))
        ("w".(((while_loop)) nil )) )
```

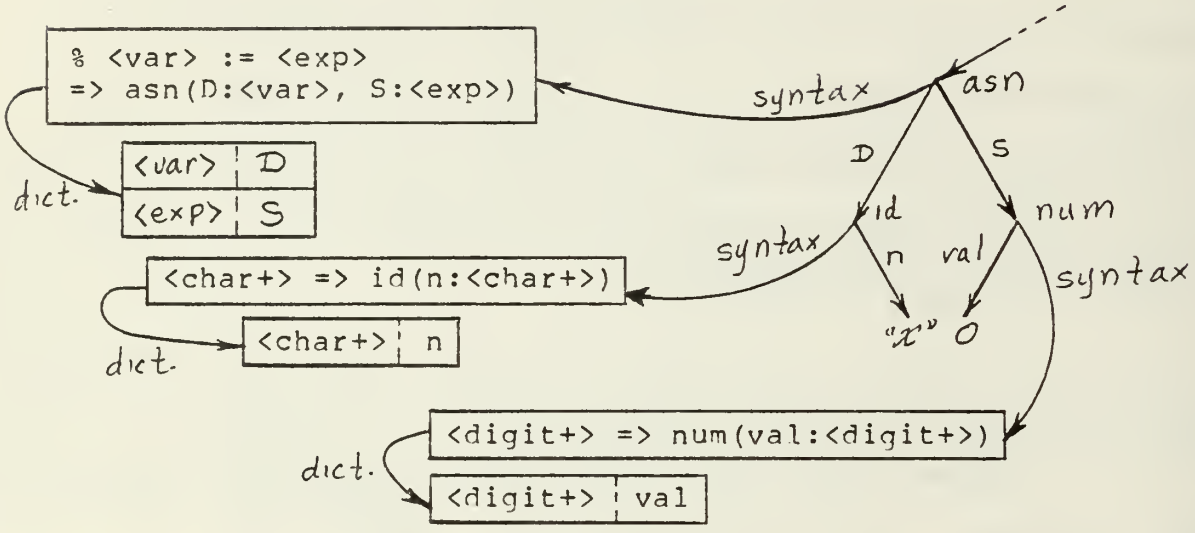
e 'nil' represents an identity-rule; otherwise the translation
exactly as described before. (I.e., (block) represents
lock>, so ((block)) is an analysis part containing only <block>
d (((block)) nil) is an identity rule with this analysis part.
sp notation does have its disadvantages!)

There are, of course, more compact ways of storing the gram-
r, although most grammars are sufficiently small that these
chniques aren't required. It is also possible to improve the
rformance a little by constructing indexes or hash tables to
orten a-list searching. However, most of the a-lists are suf-
ciently short that this seems unlikely to show a great benefit.

The Unparsing Algorithms

In Section 2, Example Session, we presented several examples
the use of the display during program editing. We also dis-
ssed how the focus/depth parameters determine what is
splayed. In this section we will describe the unparsing algo-
thms, the algorithms that regenerate the source form of the
ogram from its abstract, or tree, form.

Recall that each node has a 'syntax' attribute which points to the grammar rule which generated this node. This pointer is established whenever a node is created. An example, in diagrammatic form is:



This provides all the required information. The LHS (e.g., "%<var>_:=_<exp>") provides the display format and the NT Dictionary correlates attributes and non-terminal names e.g., (((var).(D)) ((exp).(S))). Hence, unparsing a node is a simple recursive procedure that proceeds as follows:

(1) Use the 'syntax' attribute to get the rule that generated the current node. Separate its analysis and dictionary parts.

(2) Consider each item in the analysis part in order.

(3) If it is a terminal, then display it. This may include ing formatting commands (e.g., newline or tab).

(4) If it is a non-terminal, then one of two actions is n, as described below.

(5) If the depth limit has been reached, then "..." is layed for the substructure.

(6) Otherwise, look up the non-terminal in the non-terminal ionary to get a path for accessing the node. Follow this in the program tree.

(7) If the node found by the above process is a non- inal (i.e., that attribute is undefined) then display the terminal's name.

(8) If the node found is not null, then it will be pro- ed in accordance with the modifiers on the non-terminal:

(9) If there are no modifiers or an "option" modifier ., '?') then the node is unparsed recursively (i.e., from (1)).

(10) If there is a sequence modifier (i.e., '*' or '+') unparse each descendent of the sequence node, recursively.

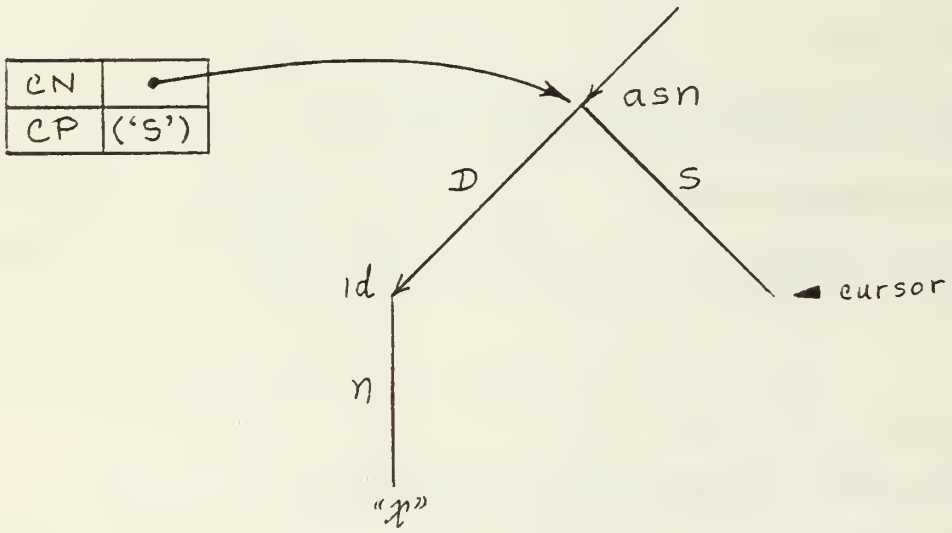
(11) If there is a list modifier (i.e., ':' in list form, in source form) then unparse each descendent of the nce node, recursively. The delimiter is displayed between sequence element.

(12) This process is continued until the entire analysis part is processed.

Thus, unparsing is driven by the analysis part of a rule.

8. The Parsing Algorithms

One of the advantages of using a syntax-directed editor is that parsing is so much simpler than in a traditional batch-oriented compiler. We will describe the processing that occurs when a user strikes a key on his terminal. Before this can be done, the reader must know that the current position in the tree is recorded as a node/path pair, called CN and CP. For instance, if the current position is the undefined 'S' attribute of a particular 'asn' node, (indicated by ◀ below),



then CN points to this 'asn' node and CP is (S). This two-coordinate method of designating positions in the tree is used because it is only allowable to position the cursor on nodes that

correspond to non-terminals (i.e., well-defined subparts of the program).

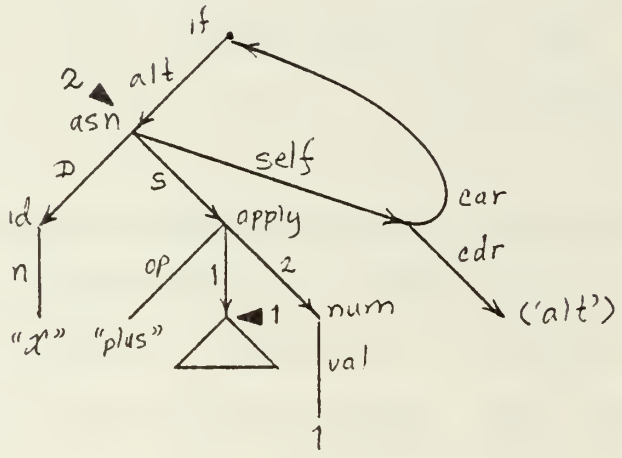
Any key the user strikes is either a language independent executive command or a language specific editing command. The executive commands will be described first. The user is only allowed to position the cursor to nodes which correspond to source language constructs (i.e., to non-terminals in the grammar). When using the positioning keys (→ ← ↑ ↓) any other nodes are skipped.

This can be done by using the non-terminal dictionary. This is an a-list which maps non-terminals into the corresponding paths. It is stored in the order in which the non-terminals occur in the LHS. Suppose we have the following non-terminal dictionary for an if statement:

```
(  ( (relation).(cond)    )
   ( (statement).(conseq) )
   ( (elsepart?).(alt)    ) )
```

If CP = (conseq), then it is easy to process either ⇒ or ⇐. For ⇒ we set CP to (alt) and for ⇐ we set CP to (cond). If there is no attribute to the right or left, respectively, these commands are equivalent to ↑⇒ or ↑⇐ respectively. That is, they search for the next component at the next higher level. The ↓ key is processed in a simple way: follow the CP path from

CN to give the new CN. The path from the first entry of the non-terminal dictionary associated with this new node becomes the new CP. To obey the ↑ key, the 'self' attribute of CN must be accessed. This gives the new (CN, CP) pair. e.g.



1▶ = CN = the 'asn'
CP = (S 1)

2▶ = CN = the 'if'
CP = (alt)

Typing ↑ moves from 1▶ to 2▶. The new (CN,CP) is (the if, alt), the 'self' attribute of the old CN.

The other executive commands (delete, grab, put, copy, open/close, etc.) are straight forward manipulations of the tree.

We will now consider those keys that do not correspond to executive commands. These are keys that are defined in the grammar for the language. Since these keys cause the creation of nodes in the program tree, they are only legal if the current position is undefined or an open sequence. If it is undefined, then the key is processed as follows:

(1) Access the 'syntax' attribute of CN. This gives the rule that generated the parent node of the current position.

(2) Search the non-terminal dictionary of this rule for CP, that will give the non-terminal associated with the current position.

(3) The key entered must be compatible with this non-terminal. There are six possible classes into which this non-terminal may fit:

1. Primitive, e.g., <char>, <digit>, <letter>.
2. Simple, e.g., <block>, <exp>.
3. Optional, e.g., <else part?>
4. Star, e.g., <decl*>
5. Plus, e.g., <digit+>
6. Listing, e.g., <statement;...>

These are described individually, below.

(4) Primitive: if the character is in the indicated class (e.g., <digit> or <letter>) then it is placed at the current position in the tree.

(5) Simple: The rule corresponding to the non-terminal's name is looked up in the grammar table. This either an analysis-synthesis rule or an alternation a-list. (a) Suppose we have a simple analysis-synthesis rule. If it is an identity rule (null synthesis part) then we extract the non-terminal name from the analysis part and repeat step (5). Otherwise we create the node specified by the synthesis part, set the current node (CN/CP) to be this node, and continue from step (1). (b) If, on the other hand, we have an alternation a-list then we look up the command character in this a-list. If it is defined then this will give us an analysis synthesis rule to be processed as in (a) above. If it is not defined, then we check for a default alternative, and if its found process according to (a). In all other cases the command is illegal.

(6) Optional: The character is processed according to the underlying non-terminal (<else part> in the example above). See step (5).

(7) Star: If the current position is undefined, then create an empty 'seq' (sequence) node. If it was undefined or not, ensure that the element of the sequence is undefined by adding a new list element if necessary. Process the key according to the

underlying non-terminal ('<decl>' in the example above).

(8) Plus and Listing: these are processed exactly like the star, except that they are unparsed differently.

9. Implementation Details

A prototype implementation of the algorithms described in this paper has been implemented in the MACLISP [7, 14] dialect of LISP on MIT's Multics machine. The use of LISP allowed the author to implement and debug the system in about 15 hours over a week period. The use of LISP in an interactive environment is to be credited with the speed with which these algorithms were debugged.

A similar system has been designed by Shockley and Haddow, which is described in [9]. This system is more general but requires the grammar to be described at a lower level. This thesis also provides a theoretical basis for grammar-driven editing and discusses the wider implications of SDEs.

10. Implications

We review some of the results of the previous sections and discuss their implications. We have seen that a syntax-directed editor is much more effective for program entry than a conventional text editor. The user is continually prompted with the allowable syntactic category and is not allowed to enter a

syntactically incorrect program. Further, the user is provided with convenient editing mechanisms because the editor "understands" the structure of the programming language. There are no complicated parsing algorithms and no concerns about look ahead, because the parsing process is entirely driven by keyboard commands from the user. There is no need for complicated error-recovery algorithms because the user is never permitted to make a syntactic error. Thus, we have a small, efficient, very easy to use parser.

Syntax-directed editors of the kind discussed in this paper are very easy to generate automatically. The grammatical specification is about the minimal possible: a BNF specification of the syntax, translation rules describing the trees to be generated for each construct, and an association of command keys with grammar rules. A small amount of additional processing generates the internal tables required by the simple parsing and unparsing algorithms. It is also simple to process this information to generate menus and other editing aids. In summary, it is very easy to generate syntax-directed editors. Hence, we seem to have the best of both worlds: a system which is easy to use, efficient in operation and simple to generate automatically.

Finally, we note some of the generalizations and future directions suggested by this work. As was indicated briefly, earlier, the use of a language independent program tree format provides for a limited ability to translate between languages.

For instance, a tree could be constructed using a Pascal syntax and then unparsed using an Ada syntax. For this purpose the grammar is processed to yield an inverted description, that takes node types into the syntax rules that generate them. This is used to build new 'syntax' attributes in the program tree.

A more general view of a syntax-directed editor is that it is a convenient human interface for constructing data structures. In our previous discussion the interface was a programming language and the data structure was a program tree, but it should be clear that the mechanisms are more generally applicable. For instance, the data structure to be manipulated might be a user's file directory. Ideas similar to this are discussed in [1].

Although a major application of syntax-directed editors will be the manipulation of conventional linearly structured languages, such as Pascal and Ada, we can see that they are not limited to languages of this sort. With somewhat more elaborate formatting constructs in the grammar, these same algorithms can be used for parsing and unparsing tables, mathematical equations and other two dimensional notations. The old card or line oriented notion of a language's syntax is no longer necessary.

11. Acknowledgements

Many of the ideas described in this report were developed in discussions with Bill Shockley and Dan Haddow in conjunction with

their thesis research [9]. Their project and the development of their system helped stimulate the work described here.

12. References

[1] Fraser, C.W., A generalized text editor, CACM 23, 3 (March 1980), 154-158.

[2] Holloway, G., Townley, J., Spitzen, J., and Wegbreit, B., ECL Programmer's Manual, Center for Research in Computing Technology, Harvard University, December 1974.

[3] Learning Research Group, Personal dynamic media, IEEE Computer Magazine, March 1977, 31-41.

[4] MacLennan, B.J. Prototype Linear Argot System Users' Manual, June 1978, available from author.

[5] MacLennan, B.J. Semantic and Syntactic Specification and Extension of Programming Languages, Purdue University PhD Dissertation, December 1975.

[6] McCarthy, J. et al., LISP 1.5 Programmer's Manual, The M.I.T. Press, 1969.

[7] Moon, D. MACLISP Reference Manual, Version 0, MIT Laboratory for Computer Science, Cambridge, Mass., April 1974. Parts 1, 2, and 3 revised 1978.

[8] Notkin, D.S. and Habermann, A.N., Software Development Environment Issues as Related to Ada, Carnegie-Mellon

University Computer Science Department, 1979.

[9] Shockley, W.R. and Haddow, D.P., A Conceptual Framework for Grammar-Driven Synthesis, Masters thesis, Computer Science Department, Naval Postgraduate School, December 1980.

[10] Teitelbaum, T. and Reps, T., The Cornell program synthesizer: a syntax-directed programming environment, CACM 24, 9 (September 1981), 563-573.

[11] Teitelman, W., A Display Oriented Programmer's Assistant, Xerox Palo Alto Research Center, CSL-77-3, March 1977.

[12] Teitelman, W., et al., Interlisp Reference Manual, Xerox Palo Alto Research Center, December 1975.

[13] Tesler, L. The Smalltalk environment, Byte 6, 8 (August 1981), 90-147.

[14] Winston, P.H. and Horn, B.K.P. Lisp, Addison Wesley, 1981.

APPENDIX: Minigol Translator

```
Minigol: <block>;

<block>:   %begin_<decl*>  <statement;...>_end
        => block (head: <decl>*,  body: <statement;...>);

<decl>:  <type>_<id>  =>  decl(n: <id>, t:<type>);

<type>:  'n' integer  =>  int()
       | 'r' real  =>  real();

<statement>:   'b' <block>
             | 'a' <assignment>
             | 'i' <if stat>
             | 'w' <while loop>;

<assignment>:  %<var> _:=_<exp> => asn(D:<var>, S:<exp>);

<if stat>:  %if_ <relation> _then <statement> <else part?>
        =>  if(Cond:<relation>, Conseq:<statement>, Alt: <else part?>);

<else part>:  %else_ <statement>;

<while loop>:  %while_ <relation> _do <statement>
        =>  while(Cond:<relation>, body:<statement>);

<relation>:  <exp>_<relop>_<exp'>
        =>  apply(op:<relop>, 1:<exp>, 2:<exp'>);

<relop>:   '='   =   =>   "eq"
         | 'n'   ≠   =>   "ne"
```

```
        | '<'   <   =>   "lt"
        | '>'   >   =>   "gt"
        | 'l'   ≤   =>   "le"
        | 'g'   ≥   =>   "ge";

<exp>:  '+' <exp> + <term>
        => apply(op: "plus", 1:<exp>, 2:<term>)
      | '-' <exp> - <term>
        =>  apply(op:"minus", 1:<exp>, 2:<term>)
      | ' '  <term>;

<term>:  '*' <term>*<factor>
         => apply(op:"times", 1:<term>, 2:<factor>)
       | '/' <term>/<factor>
         => apply(op:"over", 1:<term>, 2:<factor>)
       |  "  <factor>;

<factor>:  '(' (<exp>)
         | ' '  <primary>;

<primary>: '#' <number>
          |' '  <var>;

<var>:  '_' <id>
      | '[' <id> [<exp>]  =>  subs(a:<id>, i:<exp>);

<id>:  <char+>  =>  id(n: <char+>);

<number>:  <digit+>  =>  num(val: <digit+>).
```

INITIAL DISTRIBUTION LIST

Defense Technical Information Center Cameron Station Alexandria, VA 22314	2
Dudley Knox Library Code 0142 Naval Postgraduate School Monterey, CA 93940	2
Office of Research Administration Code 012A Naval Postgraduate School Monterey, CA 93940	1
Chairman, Code 52Bz Department of Computer Science Naval Postgraduate School Monterey, CA 93940	40
Professor Bruce J. MacLennan, Code 52Ml Department of Computer Science Naval Postgraduate School Monterey, CA 93940	12
Bill Brown Intel Corporation Aloha, OR 97005	1
John Banning Zylog Corporation 10460 Bubb Road Cupertino, CA 95014	1
Jeffrey D. Liotta Amdahl Corporation 1250 E. Arques Avenue P.O. Box 470 Sunnyvale, CA 94086	1
Susan L. Graham Editor in Chief, TOPLAS Computer Science Division - EECS University of California at Berkeley Berkeley, CA 94720	1
Christopher W. Fraser Department of Computer Science The University of Arizona Tuscon, AZ 85721	1
Tim Teitelbaum Department of Computer Science Cornell University 405 Upson Hall Ithaca, NY 14853	1