Bit 7: Trace sequence of calls to EXECSEM and literals passed during graph generation.

Bit 8: After each parse rule has modified the graph, output it anew.

Bit 9: Trace graph when generation is complete.

Bit 10: Output time used in each segment of translation.

Bit 11: Print the string representation of the right-side graph grammar as read in.

Bit 12: Trace the sequence of parse rules used.
IV. THE INTERPRETER

The graphs that the TWS generates are amenable to execution with a very trivial interpreter. This section will attempt to describe such an interpreter, even though one is not currently in existence.

The structure of the completed program graph is detailed in Pratt (12); an informal description will lay the groundwork for the interpreter design here. Basically, the program graph will consist of two main system nodes, P and CEP, with some less important stacks at the system level (E-stack, L-stack). The P node forms a top-level instruction node with the one instruction "fetch-instruction" node operating on the nodes BRANCH, Q, and CIP (current instruction pointer). CIP contains the program graph and its associated sequence of instruction nodes.

All the primitive operations, including the primitive "fetch-instruction", are defined as graph transformations which change the state of the abstract machine. The transformation definitions are included in Appendix H. A careful analysis of the primitive "fetch-instruction" reveals that it places the next instruction node as the value of node Q and increments the instruction pointer. Interpretation is then logically a two-step process: execute the instruction in node P (fetch the next instruction), then execute the instruc...
in node \( Q \) (execute the instruction just fetched). The above process continues until no more instructions can be fetched.

An interpreter to accomplish the above process need be no more than a set of subroutines written in GROPE which perform the primitive graph transformations needed. One main subroutine could be written as the executive which executed the "fetch-instruction", checked to see if node \( Q \) was empty, and then called the subroutine designated by the function in the instruction of node \( Q \). The graph transforming routines themselves are quite uncomplicated; GROPE provides many flexible means of following arcs to retrieve and change the values of the \( \alpha, \beta \), etc. nodes mentioned in the transformation description.
V. EVALUATION AND DIRECTION

The TWS described in this work does not have the benefit of extensive use at this point, nor does it approach the translation problem in a manner similar enough to previous attempts to permit close comparison. The system is moving into a user status at the University of Texas fairly rapidly, however, so concrete results should be forthcoming.

In the meantime, several aspects of the system are notable enough to require comment. The general theme in the design of this TWS has been generality and simplicity at the expense of speed and efficiency. It was determined that an implementation of this sort should be based on the formal translation of string languages to graph representations, without the heuristics that seem to populate so many TWS's whose primary concern is an efficient translation. This TWS was designed to translate a large class of string languages into their graph representations given the appropriate pair grammar with a minimum of modification. To this end, it accomplishes its task quite well.

The usefulness of the TWS will become clear when more analysis work is done on the program graph model. If such work proves fruitful, then the TWS will surely make a sizeable contribution to the field of program modeling. Indeed, even if the specific graph models are never treated formally, an
important step forward will have been made when it is shown that the TWS-generated program model can be executed. When an interpreter is available, the capability will then exist to model an implementation of a language. The virtual machine which supports a language can be graphically depicted simply and quickly through the appropriate pair grammar definition. It seems clear that the various methods of implementing a certain language feature could be changed at will using this technique.

The use of H-graphs to represent programs is showing quite a bit of promise in current correctness-proving work as well. With the capability of now creating program models which include semantic information, perhaps program verification studies can be aided by this system. Certainly the delineation of control paths into arcs connecting instruction nodes makes determination of "what happens next" in a program clearer.

Finally, the TWS may prove useful in the field of automata theory, since the translation produced is nothing more than a finite-state automaton. An interpreter may be used to drive this automaton through its various "states" in a step-by-step fashion, stopping when an "interesting" state has been entered. The state diagram of a recognizer for a particular grammar could be constructed and tested, for example.
In this light, then, the road ahead is clear. The interpreter (for ALGOL 60) comes next. Once this is completed, an extensive testing program must be begun to show whether or not the entire system can effectively model the semantics of an ALGOL program. Several choices remain after that for further work. New pair grammars should be developed for other programming languages; not only will they further test the TWS— they will also test the range of applicability of pair grammars.

This translator writing system will undoubtedly be followed by many more sophisticated ones as the theory of program semantics grows. They will be more efficient, more general, and will handle a wider variety of output forms than this one. They will eventually take the task of translation from the compiler writer's hands completely. This TWS may be looked upon as merely a step in the right direction—a step away from the heuristically-oriented TWS's of the past and toward the formally-defined ones of the future.
Appendix A: ALGOL 60 Pair Grammar
Formal definition of the translator

Programs and Statements

1* \( \langle \text{program} \rangle := \langle \text{unlab.block} \rangle \)
   or \( \langle \text{unlab.empd} \rangle \)

Note: This rule defines the initial state of the abstract machine and the initial values of all "system" nodes.

2 \( \langle \text{block} \rangle ::= \langle \text{label} \rangle : \langle \text{block} \rangle \)
   \( \langle \text{block} \rangle ::= \# \langle \text{label} \rangle \)

3 \( \langle \text{block} \rangle ::= \langle \text{unlab.block} \rangle \)

4 \( \langle \text{unlab.block} \rangle ::= \langle \text{block.head} \rangle ; \langle \text{empd.tail} \rangle \)

\( \langle \text{unlab.block} \rangle ::= \)
5  <block head> ::= begin <declar>
6  <block head> ::= <block head>; <declar>
7  <declar>  ::= <type decl>
8  <declar>  ::= <array decl>
9  <declar>  ::= <proc decl>
10 <declar> ::= <switch decl>
11 <declar> ::= <label decl>
12 <cmpd stmt> ::= <label> ; <cmpd stmt>
13 <cmpd stmt> ::= <unlab cmpd>
14 <unlab cmpd> ::= begin <cmpd tail>
15 <cmpd tail> ::= <stmt> end
16 <cmpd tail> ::= <stmt> ; <cmpd tail>
17 <stmt> ::= <uncond stmt>
18 <stmt> ::= <cond stmt>
19 <stmt> ::= <for stmt>
20 <uncond stmt> ::= <block>
21 <uncond stmt> ::= <cmpd stmt>
22 <uncond stmt> ::= <basic stmt>
23 <basic stmt> ::= <label> ; <basic stmt>
24 \( \langle \text{basic stmt.} \rangle ::= \langle \text{unlab basic stmt.} \rangle \)

25 \( \langle \text{unlab basic stmt.} \rangle ::= \langle \text{proc stmt.} \rangle \)

26 \( \langle \text{unlab basic stmt.} \rangle ::= \langle \text{goto stmt.} \rangle \)

27 \( \langle \text{unlab basic stmt.} \rangle ::= \langle \text{dummy stmt.} \rangle \)

28 \( \langle \text{unlab basic stmt.} \rangle ::= \langle \text{assign stmt.} \rangle \)

29 \( \langle \text{dummy stmt.} \rangle ::= \langle \text{empty} \rangle \)

30 \( \langle \text{cond stmt.} \rangle ::= \langle \text{label} \rangle ; \langle \text{cond stmt.} \rangle \)

31 \( \langle \text{cond stmt.} \rangle ::= \langle \text{if clause} \rangle \langle \text{for stmt.} \rangle \)

32 \( \langle \text{cond stmt.} \rangle ::= \langle \text{if clause} \rangle \langle \text{uncond stmt.} \rangle \)

33 \( \langle \text{cond stmt.} \rangle ::= \langle \text{if clause} \rangle \langle \text{uncond stmt.} \rangle \langle \text{else} \rangle \langle \text{ stmt. } \rangle \)

34 \( \langle \text{for stmt.} \rangle ::= \langle \text{label} \rangle ; \langle \text{for stmt.} \rangle \)

35 \( \langle \text{for stmt.} \rangle ::= \langle \text{for clause} \rangle \langle \text{stmt.} \rangle \)

\[ \langle \text{for stmt.} \rangle ::= \langle \text{for clause} \rangle \langle \text{stmt.} \rangle \]

\[ \langle \text{create for body} \rangle \rightarrow \langle \text{A-R} \rangle \]

\[ \langle \text{for body} \rangle \rightarrow \langle \text{C-E} \rangle \]

\[ \langle \text{for clause} \rangle \rightarrow \langle \text{A} \rangle \]

\[ \langle \text{for clause} \rangle \rightarrow \langle \text{C-E} \rangle \]

\[ \langle \text{for clause} \rangle \rightarrow \langle \text{C-E} \rangle \]
36 \textbf{for clause}::= \textbf{for} \textbf{for var}::= \textbf{for} \textbf{for list} do \textbf{for clause}::= I<\textbf{var}> \\
\hspace{1cm} \textbf{create} \textbf{temp} \textbf{label} \textbf{temp} TEMP \\
\hspace{1cm} \textbf{install} \textbf{new} \textbf{var}+A-R+CEP \\
\hspace{1cm} \textbf{for list} \\
37 \textbf{for list}::= \textbf{for list}, \textbf{for list elem} \hspace{1cm} \textbf{for list}::= I<\textbf{for list}> \\
\hspace{1cm} \textbf{for list elem} \\
38 \textbf{for list}::= \textbf{for list elem} \\
39 \textbf{for list elem}::= \textbf{arith expr} \hspace{1cm} \textbf{for list elem}::= I<\textbf{arith expr}> \\
\hspace{1cm} \textbf{eval} \textbf{for index} \textbf{locl} \textbf{sp} \textbf{for body} \textbf{locl} \textbf{sp} \textbf{CIP} \textbf{CEP} \\
40 \textbf{for list elem}::= \textbf{arith expr} \textbf{while} \textbf{Bool expr} \\
\hspace{1cm} \textbf{for list elem}::= I<\textbf{arith expr}> \\
\hspace{1cm} \textbf{eval} \textbf{for index} \textbf{locl} \textbf{sp} \textbf{for body} \textbf{locl} \textbf{sp} \textbf{CIP} \textbf{CEP} \\
\hspace{1cm} \textbf{get} \textbf{branch} \textbf{label} \textbf{locl} \textbf{sp} \textbf{E stack} \\
\hspace{1cm} \textbf{br} \textbf{CRUNCH} \textbf{false} \textbf{locl} \\
\hspace{1cm} \textbf{true}
41 \texttt{<for list elem.>}::=\texttt{<arith expr> step<arith expr> until<arith expr>}

\begin{itemize}
\item \texttt{<for list elem.>}::=\texttt{<arith expr> step<arith expr> until<arith expr>}
\item \texttt{<arith expr>}::=\texttt{<for index>}
\item \texttt{<for index>}::=\texttt{<eval> if<arith expr> eg<arith expr>}
\item \texttt{<eval>}::=\texttt{<eval> if<arith expr> eg<arith expr>}
\item \texttt{<assign stmt>}::=\texttt{<left part> assign<stmt>}
\end{itemize}

42 \texttt{<assign stmt>}::=\texttt{<left part> assign<stmt>}

43 \texttt{<assign stmt>}::=\texttt{<left part> assign<stmt> or<arith expr>}

44 \texttt{<left part>}::=\texttt{<var>}

\begin{itemize}
\item \texttt{<assign stmt>}::=\texttt{<left part> assign<stmt>}
\item \texttt{<left part>}::=\texttt{<var>}
\end{itemize}
45 \(<\text{proc.stmt.}>::=\langle\text{proc-id}\rangle\langle\text{actual parameter part}\rangle\)

\(<\text{proc.stmt.}>::=
\)

46 \(<\text{function design.}>::=\langle\text{proc-id}\rangle\langle\text{actual parameter part}\rangle\)

\(<\text{function design.}>::=
\)

47*\(<\text{actual parameter part}>::=()\)

\(<\text{actual parameter part}>::=[\langle\text{actual param. list}\rangle]

48 \(<\text{actual parameter part}>::=[\langle\text{actual param. list}\rangle]

49 \(<\text{actual param. list}>::=[\langle\text{actual param. list}\rangle];\langle\text{param.delim.}\rangle;\langle\text{actual param.}\rangle

50 \(<\text{actual param. list}>::=[\langle\text{actual param.}\rangle

51 \(<\text{actual param.}>::=[\langle\text{string}\rangle

\(<\text{actual param.}>::=
\)
52 \textit{actual param} ::= \textit{expression} \\

53 \textit{expression} ::= \textit{arith. expr.} or \textit{Bool. expr.} \\

54 \textit{go to stmt.} ::= \textit{go to design expr.} \\
Note: The graph is disconnected by this rule.

55 \textit{design expr.} ::= \textit{if clause} \textit{simp design expr.} else \textit{design expr.} \\

56 \textit{design expr.} ::= \textit{simp design expr.} \\

57 \textit{simp design expr.} ::= \textit{design expr.} \\

58 \textit{simp design expr.} ::= \textit{label} \\

59 \textit{label} ::= \textit{identifier} \\

60 \textit{simp design expr.} ::= \textit{switch-id}[\textit{subs expr.}]
Expressions and Variables

61 \( \text{arith.expr} ::= \text{simp.arith.expr} \)
62 \( \text{arith.expr} ::= \text{if clause} \) \( \text{arith.expr} \) \( \text{else} \) \( \text{arith.expr} \)
63 \( \text{if clause} ::= \text{if Bool.expr then} \)
64* \( \text{simp.arith.expr} ::= \text{simp.arith.expr} + \text{term} \)
65* \( \text{simp.arith.expr} ::= \text{simp.arith.expr} - \text{term} \)
66* \( \text{simp.arith.expr} ::= + \text{term} \)
67* \( \text{simp.arith.expr} ::= - \text{term} \)
68 \( \text{simp.arith.expr} ::= \text{term} \)
69 \( \text{term} ::= \text{term} \langle \text{mult-op} \rangle \text{factor} \)
70 \( \text{term} ::= \text{factor} \)
71 \( \text{mult-op} ::= * \)
72 \( \text{mult-op} ::= / \)
73 \( \text{mult-op} ::= \div \)
74 \textless factor\textgreater ::= \textless factor\textgreater \textcircled{\texttimes} \textless primary\textgreater

75 \textless factor\textgreater ::= \textless primary\textgreater

76 \textless primary\textgreater ::= (\textless arith. exp\textgreater )

77 \textless Bool. exp\textgreater ::= \textless if\textgreater \textless clause\textgreater \textless simp. Bool.\textgreater \textcircled{\textbf{else}} \textless Bool. exp\textgreater

78 \textless Bool. exp\textgreater ::= \textless simp. Bool.\textgreater

79 \textless simp. Bool.\textgreater ::= \textcircled{\textbf{I}} \textless simp. Bool.\textgreater

80 \textless simp. Bool.\textgreater ::= \textless implic.\textgreater

81 \textless implic.\textgreater ::= \textless implic.\textgreater \textcircled{\textbf{implic.}} \textless Bool. term\textgreater

82 \textless implic.\textgreater ::= \textless Bool. term\textgreater

83 \textless Bool. term\textgreater ::= \textless Bool. term\textgreater \textcircled{\textbf{V}} \textless Bool. factor\textgreater

84 \textless Bool. term\textgreater ::= \textless Bool. factor\textgreater

85 \textless Bool. factor\textgreater ::= \textless Bool. factor\textgreater \textcircled{\textbf{L}} \textless Bool. second\textgreater

86 \textless Bool. factor\textgreater ::= \textless Bool. second\textgreater
87 <Bool.second.> ::= ¬ <Bool.primary>  
88 <Bool.second.> ::= <Bool.primary>  
89 <Bool.primary> ::= ( <Bool.expr.> )  
90 <Bool.primary> ::= <relation>  
91 <relation> ::= <simple.arith.expr.1><rel-op><simple.arith.expr.2>  
92 <Bool.primary> ::= <var.> or <function desiģ>  
93 <Bool.primary> ::= <logical.value>  
94 <primary> ::= <var.> or <function desiģ>  
95 <primary> ::= <unsigned integer>  
96 <primary> ::= <unsigned real>  
97 <var.> ::= <simple var.>
106 \texttt{<type list>} ::= \texttt{<simple var>}

107* \texttt{<own type list>} ::= \texttt{<simple var>}, \texttt{<own type list>}

108* \texttt{<own type list>} ::= \texttt{<simple var>}

109 \texttt{<switch decl>} ::= \texttt{switch \langle switch-id \rangle ::= \langle switch list \rangle}

110 \texttt{<switch list>} ::= \texttt{<switch list>}, \texttt{<design.expr>}

111 \texttt{<switch list>} ::= \texttt{<design.expr>}

112* \texttt{<label decl>} ::= \texttt{label \langle label list \rangle}

113* \texttt{<label list>} ::= \texttt{<label list>}, \texttt{<label>}

114* \texttt{<label list>} ::= \texttt{<label>}

\texttt{<design.expr>} ::= \texttt{...}

\texttt{<label list>} ::= \texttt{...}

\texttt{<design.expr>} ::= \texttt{...}

\texttt{<label list>} ::= \texttt{...}

\texttt{<design.expr>} ::= \texttt{...}

\texttt{<design.expr>} ::= \texttt{...}

\texttt{<design.expr>} ::= \texttt{...}

\texttt{<design.expr>} ::= \texttt{...}
\[ 115^* \text{ (proc.decl.) := type proc def proc.head proc.body or omit} \]

\[ \begin{align*}
\text{(proc.decl.) :=} & \\
\text{create proc} & \text{proc-id (proc-id)} \\
\text{proc.type} & \text{AR} \\
\text{type or real if omitted} & \text{CEP} \\
\text{if a} & \text{a} \\
\text{(proc.head)} & \\
\text{(proc.body)} & \\
\text{exit} & \text{JEP} \\
\text{CEP} & \\
\end{align*} \]

\[ 116 \text{ (proc.body) := stmt} \]

\[ \begin{align*}
\text{(proc.body) :=} & \\
\text{stmt} & \\
\end{align*} \]

\[ 117 \text{ (proc.body) := code} \]

\[ \begin{align*}
\text{(proc.body) :=} & \\
\text{code} & \\
\end{align*} \]

\[ 118 \text{ (proc.head) := formal.param.part value.part spec.part} \]

\[ \begin{align*}
\text{(proc.head) :=} & \\
\text{inst new wrt AR} & \text{CEP} \\
\text{(value.part)} & \\
\text{(spec.part)} & \\
\end{align*} \]

\[ 119 \text{ (formal.param.part) := empty} \]

\[ \begin{align*}
\text{(formal.param.part) :=} & \\
\text{#} & \\
\end{align*} \]

\[ 120 \text{ (formal.param.part) := (formal.param.list)} \]

\[ \begin{align*}
\text{(formal.param.part) :=} & \\
\text{(formal.param.list)} & \\
\end{align*} \]

\[ 121 \text{ (formal.param.list) := (formal.param.list) param.delim (formal.param)} \]

\[ \begin{align*}
\text{(formal.param.list) :=} & \\
\text{(formal.param.list)} & \text{&} \\
\text{(formal.param.list)} & \\
\end{align*} \]

\[ 122 \text{ (formal.param.list) := (formal.param)} \]

\[ \begin{align*}
\text{(formal.param.list) :=} & \\
\text{(formal.param)} & \\
\end{align*} \]

\[ 123 \text{ (formal.param) := identifier} \]

\[ \begin{align*}
\text{(formal.param) :=} & \\
\text{create-param | (identifier)} & \text{AR} \\
\text{P-stack} & \\
\end{align*} \]
124 \texttt{value part} ::= empty
125* \texttt{value part} ::= value \texttt{value ident list};
126* \texttt{value ident list} ::= \texttt{value ident list}, \texttt{identifier}

127* \texttt{value ident list} ::= \texttt{identifier}

128 \texttt{spec part} ::= empty
129* \texttt{spec part} ::= \texttt{specifier} \texttt{spec ident list};
130* \texttt{spec part} ::= \texttt{spec part} \texttt{specifier} \texttt{spec ident list};
131* \texttt{spec ident list} ::= \texttt{spec ident list}, \texttt{identifier}
132* \text{<spec. ident. list> ::= <identifier>}
\begin{align*}
\text{<spec. ident. list> ::=} & \\
\text{use-specifier} & \text{id (<identifier>)}
\end{align*}

133* \text{<array decl.> ::= array <real array list>} \\
134* \text{<array decl.> ::= real array <real array list>} \\
135* \text{<array decl.> ::= own real array <own real list>} \\
136* \text{<array decl.> ::= integer array <integer array list>} \\
137* \text{<array decl.> ::= own integer array <own integer list>} \\
138* \text{<array decl.> ::= Boolean array <Boolean array list>} \\
139* \text{<array decl.> ::= own Boolean array <own Boolean list>}
\begin{align*}
\text{<α β list> ::= <α β segment>} & \\
\text{<α β list> ::= <α β list>, <α β segment>} \\
\text{where } α β \text{ one of } & \\
\{ \text{real array, integer array, Boolean array, own real, own integer, own Boolean} \}
\end{align*}

140* \text{<α β list> ::= <α β segment>}
\begin{align*}
\text{<α β list> ::=} & \\
\text{<α β list>} & \xrightarrow{α} \text{<α β segment>}
\end{align*}

141* \text{<α β list> ::= <α β list>, <α β segment>}
\begin{align*}
\text{<α β list> ::=} & \\
\text{<α β list>} & \xrightarrow{α} \text{<α β segment>}
\end{align*}

142* \text{<α array segment> ::= <array-id>, <α array segment>}
\begin{align*}
\text{<α array segment> ::=} & \\
\text{<α array segment>} & \xrightarrow{α} \text{<array-id>}
\end{align*}

143* \text{<α array segment> ::= <array-id>[<bd pair list>]} \\
\begin{align*}
\text{<α array segment> ::=} & \\
\text{where } α \text{ one of } & \\
\{ \text{real, integer, Boolean} \}
\end{align*}
144\* \textit{<own \( \beta \) segment>:} := \textit{<array-id>, <own \( \beta \) segment>}

\textit{<own \( \beta \) segment>:} :=\[
\begin{array}{c}
\text{create-own-array} \\
\text{val} \\
\text{type} \\
\text{d-v} \\
\text{a-r} \\
\end{array}
\]

\textit{<bd_pair list>:} := \textit{<bound pair>, <bd_pair list>}

145 \textit{<own \( \beta \) segment>:} := \textit{<array-id>[<bd_pair list>]}

where \( \beta \) = one of \{real, integ, Bool\}

146 \textit{<bd_pair list>:} := \textit{<bound pair>, <bd_pair list>}

\textit{<bd_pair list>:} := \textit{<bound pair>}

147 \textit{<bd_pair list>:} := \textit{<bound pair>}

\textit{<bound pair>:} := \textit{<lower bound>: <upper bound>}

148 \textit{<bound pair>:} := \textit{<lower bound>: <upper bound>}

\textit{<lower bound>:} := \textit{<arith_expr>}

149 \textit{<lower bound>:} := \textit{<arith_expr>}

\textit{<upper bound>:} := \textit{<arith_expr>}

150 \textit{<upper bound>:} := \textit{<arith_expr>}

\textit{<bd_pair list>:} := \textit{<bound pair>}
Appendix B: Reserved Word List
### Reserved Word List

<table>
<thead>
<tr>
<th>Word</th>
<th>Description</th>
<th>Word</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FCHINS</td>
<td>fetch-instruction</td>
<td>MULT</td>
<td>mult</td>
</tr>
<tr>
<td>ENTBLK</td>
<td>enter-block</td>
<td>REALDIV</td>
<td>realdiv</td>
</tr>
<tr>
<td>INSNAR</td>
<td>install-new-a-r</td>
<td>INTDIV</td>
<td>intdiv</td>
</tr>
<tr>
<td>EXITX</td>
<td>exit</td>
<td>EXPON</td>
<td>expon</td>
</tr>
<tr>
<td>CRFRBDY</td>
<td>create-for-body</td>
<td>EQUIV</td>
<td>equiv</td>
</tr>
<tr>
<td>CRFRNDX</td>
<td>create-for-index</td>
<td>IMPLIC</td>
<td>implic</td>
</tr>
<tr>
<td>EVAL</td>
<td>eval</td>
<td>OR</td>
<td>or</td>
</tr>
<tr>
<td>SIMPASS</td>
<td>simple-assign</td>
<td>AND</td>
<td>and</td>
</tr>
<tr>
<td>GTBRLBL</td>
<td>get-branch-label</td>
<td>NOT</td>
<td>not</td>
</tr>
<tr>
<td>ADD</td>
<td>add</td>
<td>RELOP</td>
<td>relop</td>
</tr>
<tr>
<td>STACKX</td>
<td>stack</td>
<td>EVSBVAR</td>
<td>eval-sub-var</td>
</tr>
<tr>
<td>TSUEX</td>
<td>test-step-until-exit</td>
<td>SIMPMOV</td>
<td>simple-move</td>
</tr>
<tr>
<td>STKASSGN</td>
<td>stack-assign</td>
<td>CRVAR</td>
<td>create-variable</td>
</tr>
<tr>
<td>UNSTACK</td>
<td>unstack</td>
<td>CROVAR</td>
<td>create-own-var.</td>
</tr>
<tr>
<td>CALL</td>
<td>call</td>
<td>CRSW</td>
<td>create-switch</td>
</tr>
<tr>
<td>GTPRVLN</td>
<td>get-proc-val-node</td>
<td>CRLAB</td>
<td>create-label</td>
</tr>
<tr>
<td>CRNMPAR</td>
<td>create-name-parameter</td>
<td>CRPROC</td>
<td>create-PROC</td>
</tr>
<tr>
<td>GOTOLAB</td>
<td>go-to-label</td>
<td>CRPAR</td>
<td>create-parameter</td>
</tr>
<tr>
<td>GOTOSW</td>
<td>go-to-switch</td>
<td>STVLPAR</td>
<td>set-value-param</td>
</tr>
<tr>
<td>SUBT</td>
<td>subt</td>
<td>USESPEC</td>
<td>use-specifier</td>
</tr>
<tr>
<td>NEGATE</td>
<td>negate</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Appendix C: Graph Mini-Language Representation

and the FEP's to Recognize It
126: <VALIDLIST> = <VALIDLIST> ++ [([VAL] ++ TEMP ++ EIP ++ CIEE ++ CEPF ++ IIC ++ [AN] ++ <IDENT>)] ++ [([STVLPAR] ++ EEP ++ CEPF ++ VAL ++ TEMP ++ IIC ++ [AN] ++)]
127: <VALIDLIST> = [([VAL] ++ TEMP ++ EIP ++ CIEE ++ CEPF ++ IIC ++ [AN] ++ <IDENT>)] ++ [([STVLPAR] ++ EEP ++ CEPF ++ VAL ++ TEMP ++ IIC ++ [AN])] / 
128: <SPECPART> = () / 
129: <SPECPART> = [((SIMPNOV) ++ CUTF ++ TEMP ++ IIC ++ [<SPECIFIEH>])]
130: <SPECPART> = [([STYPMOV) ++ OUT ++ TEMP ++ IIC ++ [<SPECIFIEH>])]
131: <SPECIFIEH> = <SPECIFIEH> ++ [([STYPMOV) ++ OUT ++ TEMP ++ IIC ++ [<SPECIFIEH>])]
132: <SPECIFIEH> = [([USESPEEC) ++ SPEC ++ TEMP ++ EEP ++ CEPF ++ IIC ++ [IDENT>])]

201: <IDENT> = <LITERAL> / 
202: <CPNID> = <LITERAL> / 
203: <SWING> = <LITERAL> / 
204: <UNINTER> = <LITERAL> / 
205: <USEHCAL> = <LITERAL> / 
206: <RELUP> = <LITERAL> / 
207: <TYPE> = <LITERAL> / 
208: <UCVAL> = <LITERAL> / 
209: <USPECFEC> = <LITERAL> / 

Appendix D: Graph Specifier Output Tables