A PAIR-GRAMMAR BASED STRING-TO-GRAPH TRANSLATOR WRITING SYSTEM

by

Robert W. Wesson

August 1973 TR-21

This paper was presented as the author's Master's Thesis in August, 1973, and was supported, in part, by a National Science Foundation grant, GJ-36424.

THE UNIVERSITY OF TEXAS AT AUSTIN

DEPARTMENT OF COMPUTER SCIENCES

ABSTRACT

The implementation of a new technique for the formal definition of programming languages is presented. Using as input the source program and a pair grammar defining a mapping of BNF string grammar rules into a set of graph grammar rules, a translator produces a representation of the source program as hierarchies of directed graphs. This representation contains explicit semantic information added through the graph grammar. An associated interpreter implementation for the program graph is discussed as well.

The routines created for the implementation are described in detail, along with a new graph grammar syntax developed to aid in input of the graph grammar rules. Use of the TWS is demonstrated with a pair grammar which defines the translation of full Algol 60 into hierarchical graphs. The construction of a simple interpreter for such a system is outlined. The TWS is evaluated on the basis of efficiency, generality, and practical utility.

TABLE OF CONTENTS

I. Introduction 8

II. Graph/Pair Grammars. 13

A. Graph Grammars and H-Graphs. 13

B. Pair Grammars. 15

III. Design of the TWS. 18

A. Graph Specifier. 20

1. Reserved Words 21

2. Graph Mini-Language. 21

3. The Mini-Language Processor. 28

B. Source Language Parser 30

C. Graph Generator. 33

1. GROPE Description. 33

2. Generator Structure and Operation. 35

3. Graph Output 37

4. Tracing Facilities 38

IV. The Interpreter. 40

V. Evaluation and Direction 42

VI. Bibliography 128

LIST OF FIGURES

1. Sample Right-Side H-Graph 15
2. Gross Structure of the TWS 19
3. The Graph Specifier 22
4. Syntax of the Mini-Language 23
5. Semantics of the Mini-Language 24
6. Pair Grammar Rule 41 27
7. Table Structure Generated by the Specifier 29
8. Graph Generator Structure 36
9. Graph Output Format 38

LIST OF APPENDICES

A. ALGOL 60 Pair Grammar 45

B. Reserved Word List 62

C. Graph Mini-Language Representation and the
FEP's to Recognize It 64

D. Graph Specifier Output Tables 70

E. SNOBOL4 Parser Listing 88

F. Specifier and Generator Listings 92

G. Sample ALGOL Program Graphs 106

H. Function Transformations 117

I. INTRODUCTION

Since the conception of higher-level programming languages, techniques for language translation have been widely discussed and studied. The original and most frequent motivation for translation has been the generation of machine code from a high-level programming language; thus most translators written to date have been compilers or assemblers. The proliferation of programming languages has had two results relating to language translation--first, translators between high-level languages have become necessary, and second, the mechanization of such translators has been shown to be a desirable and attainable goal. The development of translator writing systems (TWS) has begun from many directions.

The first and to date most successful TWS attempts have been the automatically constructed recognizers. These systems typically use grammar characteristics developed from formal syntactic studies to generate parsers--programs which accept as input a source language string and construct its derivation or parse tree as output.

One of the earliest such systems was Floyd's "operator precedence" technique, which works with grammars of the same name (5). A table of precedence relations is used as the basic input; the parsing algorithm stacks source string symbols until a table relation is satisfied and a reduction can be made. The parse thus performed is bottom-up, left-to-right. While this technique has not been used in any true TWS's, it has spawned several extensions which have had moderate success. Wirth and Weber's "simple precedence" technique is less restrictive with respect to the class of grammars allowed, yet most grammars must be manipulated considerably before use (16). Among other matrix-based recognizer techniques is one by Gries which is based on transition matrices; he has written a matrix constructor which helps in the implementation of a grammar considerably (7).

Perhaps the technique most widely known is that used by Floyd (4) and later by Evans (1). A simple interpreter attempts to match stacked incoming symbols against coded "production language" statements, calling semantic subroutines upon request. The technique is efficient, simple, and easy-to-use; coding in the "production language" is still such an art, however, that its use in a TWS is quite limited.

The job of total language translation is in fact much more than mere parsing of the source code. The lack of formal theory in the study of language semantics has so hindered the full automation of translators that even those more or less complete TWS's written have been composed of hand-coded semantic subroutines which are called when certain reductions are made and perform specific actions based upon the structure of the source code stack. Most of the compiler-compilers (TWS's which generate compilers) written are of this variety: McClure's TMG system (8), the BNF-oriented META systems (15), and Reynold's COGENT system (14) all contain subroutines which insert the language semantics during parsing. Systems such as Feldman's FSL (2) and TGS (10) go a step further and provide a new language to express the semantic code generation parts of compilation. Yet even in these systems the separateness of syntax and semantics pervades the entire translation process. Only recently has the formal work in programming language semantics produced a method of representation capable of being used in a TWS.

Such a representation has been developed by Pratt (11,12,13). Using hierarchies of directed graphs (H-graphs), the natural flowchart program representation has been formalized. Translation from the string-based program representation (source code) into the graph-based one is accomplished using a pair grammar. Such a grammar consists of input and output grammars with rules paired one-to-one. In this application, the input (or left-side) grammar is an ordinary context-free grammar defining the programming language's syntax; the output (right-side) tree, using specifier-generated tables and lists to construct the actual program graph. The discussion of this phase terminates Section III; this part also finishes the TWS description.

The final section describes the current implementation of the interpreter necessary to execute the Algol 60 program representation obtained from the translator. Because the interpreter is mainly a set of routines which perform graph transformations, it is closely tied to the language being translated, and thus is the most language-specific part of the entire system.

In all sections, Pratt's Algol 60 definition (12) will be used as an example for translation. To date, no other programming languages have been defined in this manner, so the translator does not have the benefit of extensive testing. Tests using the Algol definition, however, show the translator fully capable of handling a complete programming language with realistic time/space requirements.

II. GRAPH/PAIR GRAMMARS

Essential to the TWS is its use of graph and pair grammars, an area where little formal theory has been produced. The graph grammars used in translation, however, are quite intuitive to the interested reader in two ways: they are formalized representations of the traditional program flow-chart, and they are direct generalizations of ordinary context-free grammars. Additionally, the pair grammar concept as used here is quite simple. For the purposes of this work, an informal discussion of these concepts will suffice; their formal development is considered at length in Pratt (11).

Graph Grammars and H-Graphs

A _graph language_ is a language composed of a set of directed graphs with labeled nodes and arcs. A _graph grammar_ is a generalization of an ordinary context-free grammar which defines a graph language. Such a grammar possesses the familiar terminal and nonterminal sets of symbols, as well as productions used to generate the terminal graphs of the language. Starting with a graph containing a single nonterminal node, the productions are applied, replacing with each a nonterminal node by a graph, until no nonterminal nodes remain.

The problem of knowing just how to "hook up" the graph replacing the nonterminal node is solved by analogy to string replacement: just as a string has a head and tail, every graph is required to have an input node and an output one. Nodes and arcs may be labeled but do not have to be. All nodes have a value--it may be null, an atomic symbol, or a graph. Thus, hierarchies of graphs may be built with nodes whose values are in turn H-graphs.

Such structures effectively model the various levels of detail in a program. In the ALGOL model, for example, flow of control is represented by high level "program graphs" which contain only "instruction nodes". These nodes may have at most two exiting arcs, labeled "true" and "false". Statically-determined flow of control in an ALGOL program is followed by tracing the arcs of the program graph. Should more detail be desired, the values of the instruction nodes are graphs whose entry nodes contain the names of primitive operations. Within the instruction graph, the arcs signify the input-output structure of the operation, pointing from the input nodes to the entry node and from it to the nodes whose value will be changed by the operation. Within this example ALGOL definition, all the semantic information not present in the string syntax (the necessary static/dynamic chain structures, block and procedure entry/exit techniques, etc.) can be added through the graph grammar using only program and instruction nodes and graphs.

The notation used in this work is precisely that of

Pratt in his model description (12), see Figure 1. In this rule of the graph grammar, the nonterminal node with value "primary" is replaced by the two-node graph with an input nonterminal node "var" and an output instruction node whose value is an instruction graph. Within this graph, the entry node (always marked with a * or at the upper left-hand corner of the graph) has as its value the primitive operation "stack". The operation has the node labeled "TEMP" as its input operand, and changes the value of its output operand, the node labeled "E-STACK".

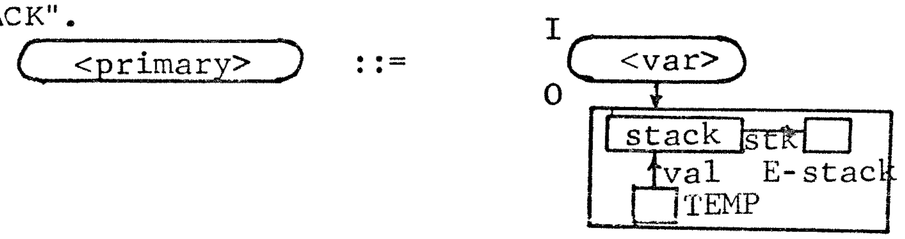


Figure 1. Sample right-side H-graph

In general, an oval represents a nonterminal node; a rectangle represents a terminal one. Node labels, when included, are written to the side of the nodes; their values are inside. The symbol # means a null value. Arcs point one direction or both. They are labeled like nodes but never have a value. The input and output nodes of a graph grammar rule are labeled I and O, respectively.

Pair Grammars

A pair grammar is composed of two grammars whose rules are paired in a one-to-one correspondence. These two grammars are denoted as left-side and right-side. With such a correspondence, it is easy to view it as a definition of the translation of one language to the other. Given a sequence of rules in the pair grammar, sentences may be generated in either language. Alternatively, given a sentence in one language (normally the one defined by the left-side grammar), a translation may be made to the right-side language using only a syntactical parse tree. Of particular interest in this work is the case in which the left-side language is a set of strings (the usual syntactic definition of programming languages) and the right-side language is a set of H-graphs (the semantic model).

The construction of a pair grammar given a left-side programming language is clearly a non-trivial problem. The pair grammar must, in effect, define a mapping from a syntactic representation of a program directly into a semantic model, without the extraneous devices of symbol tables and the like. What semantic information is necessary in the program model must be present in the grammar, and the problems involved in constructing such a grammar are comparable to those of writing a one-pass compiler without tables. The ALGOL 60 definition used as an example in this work is surprisingly simple; yet even it must omit many traditional compile-time tasks (type checking, undeclared variables, etc.). Nevertheless, these problems are mainly optimization problems; the pair grammar-based TWS is not designed for speed but rather for generality.

The specific pair grammar used to test the TWS is included in Appendix A. Each rule is numbered uniquely. The left-side grammar is the ALGOL 60 BNF grammar used in the original language specification with some very minor modifications (17). The right-side grammar is the graph grammar which generates the graph models of ALGOL programs. The entire grammar is presented and discussed at length in Pratt (12).

III. DESIGN OF THE TWS

The translator writing system is diagrammed in Figure 2. The design appears similar to other TWS's with the exception of some sort of semantic information input, which is implicit in the pair grammar input. The TWS functions as follows:

The right-side grammar of the desired pair grammar is first input to the Graph Specifier, which converts each rule to a sequence of graph generation primitive operations and tables the results. While the current version of the system re-inputs this every time, it is a costly process and could be done only once for each new pair grammar. The Source Language Parser is written so that the left-side grammar is an easily replaced module in it. The Parser accepts the source code as input and outputs the sequence of parse rules used to parse the input, together with the specific terminal symbols scanned. The Graph Generator re-traces this list of parse rules, using the tables generated by the Specifier to build the program graph.

The overall criteria governing construction of the system were generality and modularity. The TWS was written to accept any pair grammar as input which had a string-based context-free left-side grammar and a right-side graph grammar. The use of a pair grammar to specify the translation greatly

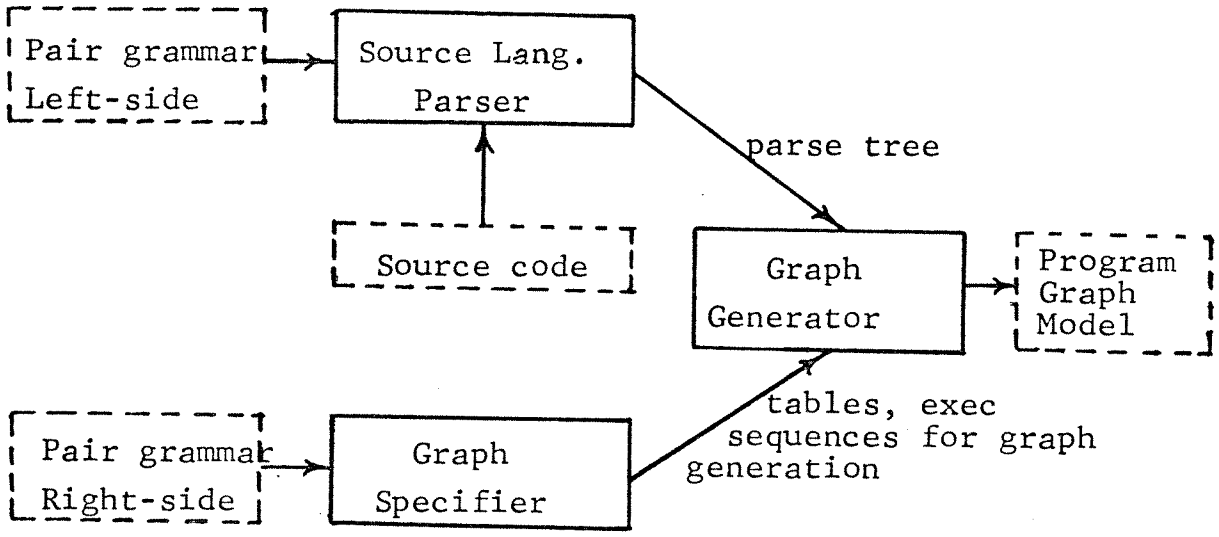


Figure 2. Gross structure of the TWS

increased the potential generality of the TWS, because of formalization of the programming language's semantics incorporated in the right-side grammar. The goal of modularity was high on the list because the system will be put into operational status in the near future, and it was desired to give individual users the capability to easily and quickly tailor the system to their own needs. Given a specific pair grammar, for example, it is trivial for a user to substitute his own Source Language Parser in place of the generalized component currently employed, thereby gaining a significant increase in speed for his application. Additionally, the modularity provides a necessary flexibility at this current formative stage in the semantic modeling field; substantial changes have already been proposed for the right-side graph grammar used in the ALGOL example. Of prime concern at this point, then, is the ability of the system to evolve as required--the optimization comes later.

The heart of the system is contained within the first component--the Graph Specifier. Thus it is here that the detailed description of system operation and construction begins.

Graph Specifier

The language chosen for coding the Specifier was FORTRAN IV. Of foremost importance was the criterion of speed at this stage in the system; the inefficiency was there without the language, and well over 100 complex right-side graph rules had to be input, parsed, and stored. Another important consideration was the space required to store the tables built; FORTRAN contains reasonable facilities for word-packing and array manipulation. Furthermore, the use of FORTRAN meant that the Graph Generator (written in GROPE/FORTRAN) could be called directly after the Specifier in the same computer run, thereby making the program more efficient. Finally, several sub-parts of the Specifier were already written in FORTRAN when the project was begun.

The structure of the specifier is depicted in detail in Figure 3. Input occurs with the entities to the left of the vertical dotted line being read in from top to bottom.

1. Reserved Words: A sequence of reserved words is read in first and hash-coded into a permanent table. This list was specifically required for the ALGOL example to distinguish the primitive operation names (EVAL, REALDIV, etc.) from other atomic names input in the same context (INTEGER, BRANCH, etc.). The list of actual reserved names chosen to correspond to the instructions in the graphs is included in Appendix B. Since the interpreter (described later) treats these names as function calls, they had to be reduced to at most seven characters. Thus, the primitive operation "get-branch-label" is represented by the name (and function in the interpreter) GTBRLBL. This reserved name list is consulted whenever a node with an atomic value is to be constructed. If the node's value is on the list, that value is flagged as a function. The usefulness of this reserved name list in other language translations is not completely clear, although it seems likely that most will contain some sort of primitive operations which require recognition on input.

2. Graph Mini-Language: The other sections of the Specifier form a recognizer (with very primitive semantic operations) for the grammar of Figure 4. This grammar was designed to solve the dilemma of graph representation--how can a hierarchy of directed graphs best be represented in string form? A number of solutions were considered before it

was determined that the construction of an entirely new string language specialized for this project would be necessary.

The mini-language produced by the grammar of Figure 4 is the best representation for the right-side graph grammar found to date. The main goals of simplicity and ease of coding were met quite well with the development of this language, as was a secondary goal of generality.

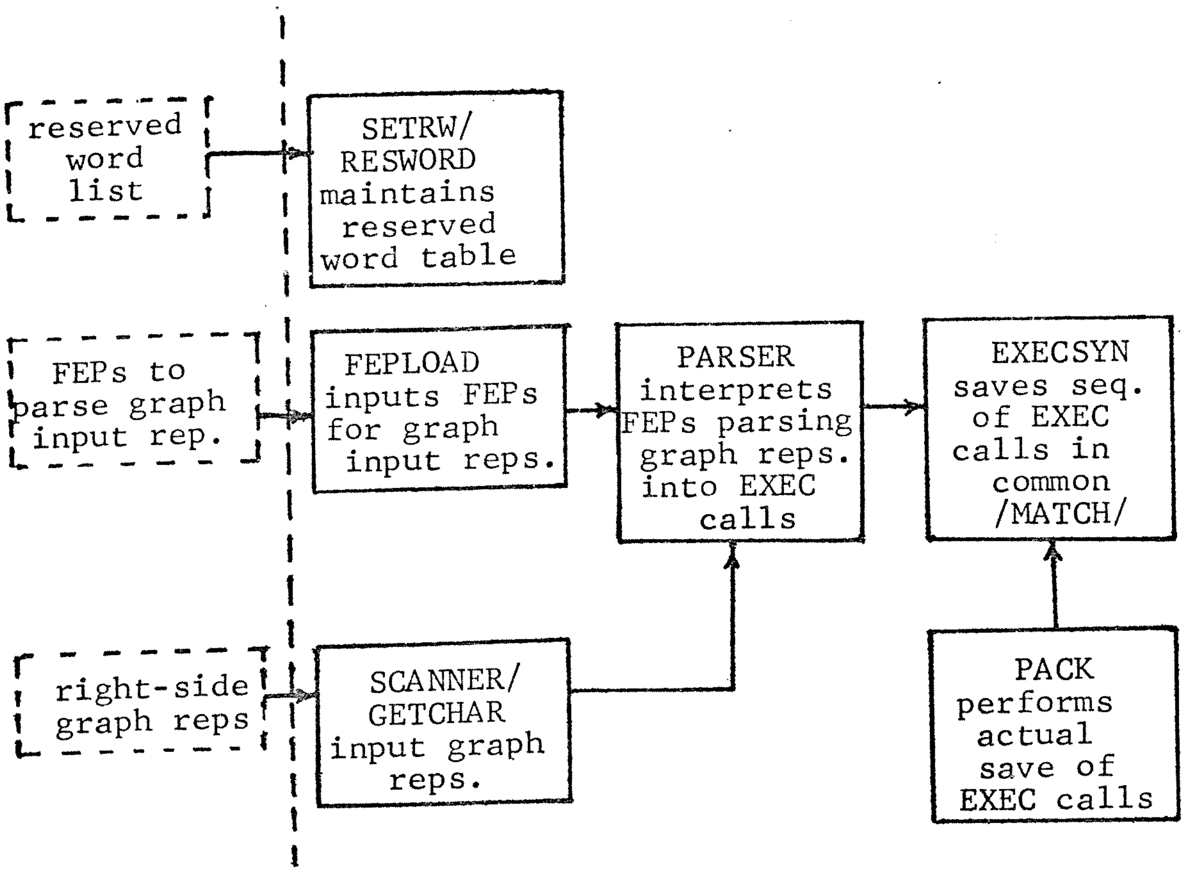

reserved word list
SETRW/ RESWORD maintains reserved word table
FEPs to parse graph input rep.
FEPLOAD inputs FEPs for graph input reps.
PARSER interprets FEPs parsing graph reps. into EXEC calls
EXECSYN saves seq. of EXEC calls in common /MATCH/
right-side graph reps
SCANNER/ GETCHAR input graph reps.
PACK performs actual save of EXEC calls

Figure 3. The graph specifier

```
<replacement> ::= <number> ':' <nonterm> '=' <nodeset> '/'
<nodeset> ::= <node> | <node> <arcset> | <node> <arcset>
              <nodeset>
<node> ::= <nonterm> | <atom-node> | <graph-node>
<nonterm> ::= '<' <nt-label> <value> '>'
<atom-node> ::= '(' <n-label> <value> ')'
<graph-node> ::= '[' <n-label> <value> ']'
<n-label> ::= '*' <ident> | λ
<value> ::= <ident> | λ
<nt-label> ::= '*NT1' | '*NT2' | ... | '*NT10' | λ
<arcset> ::= <arc> <arcset> | λ
<arc> ::= <to-arc> | <from-arc> | <tofrom-arc>
<to-arc> ::= '↓' <n-label> <a-label> <n-label> '↓'
<from-arc> ::= '↑' <n-label> <a-label> <n-label> '↑'
<tofrom-arc> ::= '≡' <n-label> <a-label> <n-label> '≡'
<a-label> ::= <ident> | λ
<ident> ::= letter followed by up to 9 letters or numbers
<number> ::= integer of up to 10 digits
```

Figure 4. Syntax of the mini-language

Construction	Meaning
<number> :	the following replacement is to be pair grammar rule # 'number'
<nonterm> = <nodeset> /	search the graph for the nonterminal 'nonterm'; change all incoming arcs to point to the first node of the nodeset; change all outgoing arcs to originate at the last node of the nodeset; delete 'nonterm' from the graph
< <nt-lab> <value> >	construct a nonterminal node; label it 'nt-lab' if present, 'NONTERM' if not; make its value 'value' if present, 'NULL' if not
(<n-lab> <value>)	construct a terminal node; label it 'n-lab' if present; search the reserved name table for 'value'; if found, make its value the function 'value'; otherwise, make its value the atomic entity 'value'
[<n-lab> <nodeset>]	construct a terminal node; label it 'n-lab' if present; construct a graph and make it the node's value; all nodes in 'nodeset' will be created on this graph; the first node in 'nodeset' will be the graph's entry node.

In the following construction, '↑' or '≡' may be substituted for '↓'. If '↑' is substituted, 'from' and 'to' are reversed; if '≡', then 'from' and 'to' are added:

↓ <n-lab$_1$><a-lab><n-lab$_2$> ↓	construct an arc from node 'n-lab$_1$' to node 'n-lab$_2$' named 'a-lab'; if 'n-lab$_1$' is missing, use the closest actual node to the left as the fromnode; if 'n-lab$_2$' is missing, use the closest node to the right as the tonode; if 'a-lab' is missing, the arc is unnamed; both 'a-lab' and 'n-lab$_2$' may not be deleted at the same time

Figure 5. Semantics of the mini-language

A cursory glance at the productions which form this language reveals the occurrence of many λ's, which represent places in the coding where default options are present. These default options complicate the semantics of the language somewhat; yet they do so much to simplify the coding that their use is clearly justified. The semantics of the language are informally described in Figure 5; most of the default options are explained at this point. Appendix C contains the complete graph representations for the sample right-side graph grammar used in this work. Perhaps the best way to gain familiarity with the language is to compare the graphs in Appendix A with the notations in Appendix C. Using the semantics developed in Figure 5, the interested reader should be able to program his own H-graphs in the mini-language reasonably soon. The fact that the entire semantics of ALGOL 60 were reduced to these few pages testifies to the conciseness of the graph representation.

The mini-language is designed to be general enough to use for most programming language graph representations. This claim has yet to be proven, of course, but it seems likely that most languages will be represented graphically in much the same way as ALGOL. In general, most language implementations can be viewed as the definition of a new "virtual machine" in which certain operations are primitive, and others (as defined by the semantics of the language) are merely combinations of these primitives (13). The mini-language developed for this work gives the designer concise ways of handling these primitives and the H-graphs of which they are a part.

There are several things to remember when using the mini-language--several points that are not immediately clear. Consider, for example, rule 41 of the sample pair grammar. When this rule is used in building the program graph, three nonterminals <arith. expr.> are generated. When the list of parse rules is further processed, which <arith. expr.> will be parsed first? Which nonterminal node <arith. expr.> will be retrieved when the search is made upon processing a rule

<ARITHEXPR> = . . . ?

The answers to these and other similar questions are implicit in the design of the Parser, Graph Generator, and Specifier. In the current version, the Parser always follows the right-most branch of the parse tree. When nodes are created during graph generation, they are stacked onto the lists which are searched when a particular nonterminal is needed. Thus, the solution is to always make the right-most nonterminal the last named in the graph representation in the mini-language. A look at the representation of graph rule 41 reveals that this is not always the most natural thing to do.

41: <for list elem.> ::= <arith. expr$_1$> step <arith. expr$_2$> until <arith. expr$_3$>

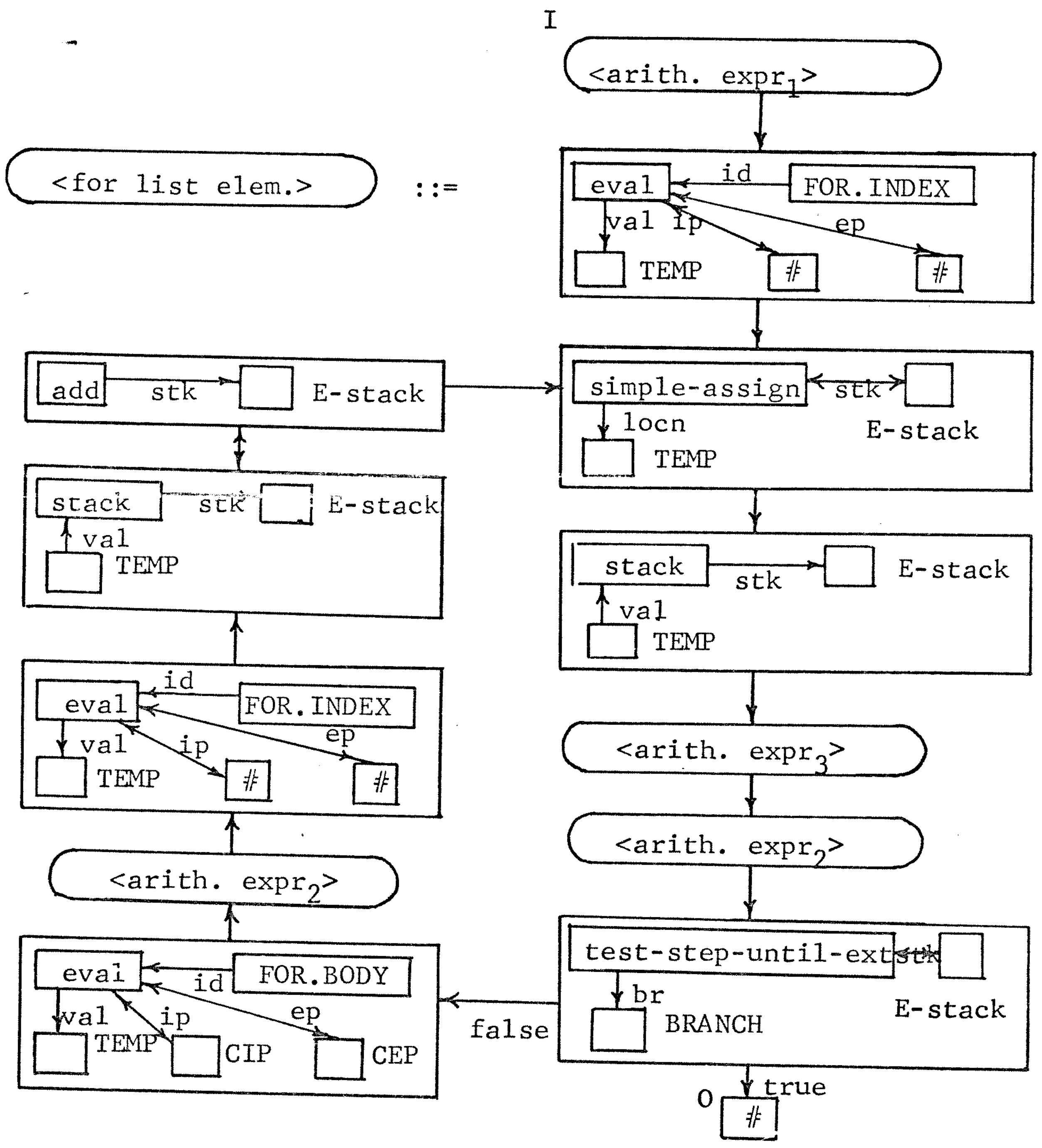


Figure 6. Pair grammar rule 41

3. The Mini-Language Processor: As was noted before, the Specifier is mainly a language processor for this mini-language. It is not particularly innovative, using FEP's to parse the source code (which in this instance is the string representation of the right-side graph grammar) and calling some trivial execs which simply build tables. The routines of Figure 3 interact as follows:

The required FEP's (about fifty productions) are input and tabled by FEPLOAD. The PARSER is then called to interpret the FEP's using the source code. In manipulating this input string, the usual lexical SCANNER is used. It uses a routine called GETCHAR to input the source cards and the "class" of each character (letter, digit, etc.). This scanner is patterned after many of the common ones in use today, most notably the one described in Gries (7).

As the Parser uses the FEP's to process the source string, exec calls are made, as in the usual compiler. However, rather than generating the graph directly at this point with the semantic execs (which could easily be done), the calls are re-directed to a routine called EXECSYN, for "syntactic exec". This routine merely packs into linear arrays the exec numbers called, and the stack elements necessary for the semantic execs to use in building the graph. It uses the routine PACK to accomplish the task of packing ten integers into one 60-bit CDC word. The structure of the tables generated is defined in Figure 7. The specific tables generated for the ALGOL graph grammar are included in Appendix D.

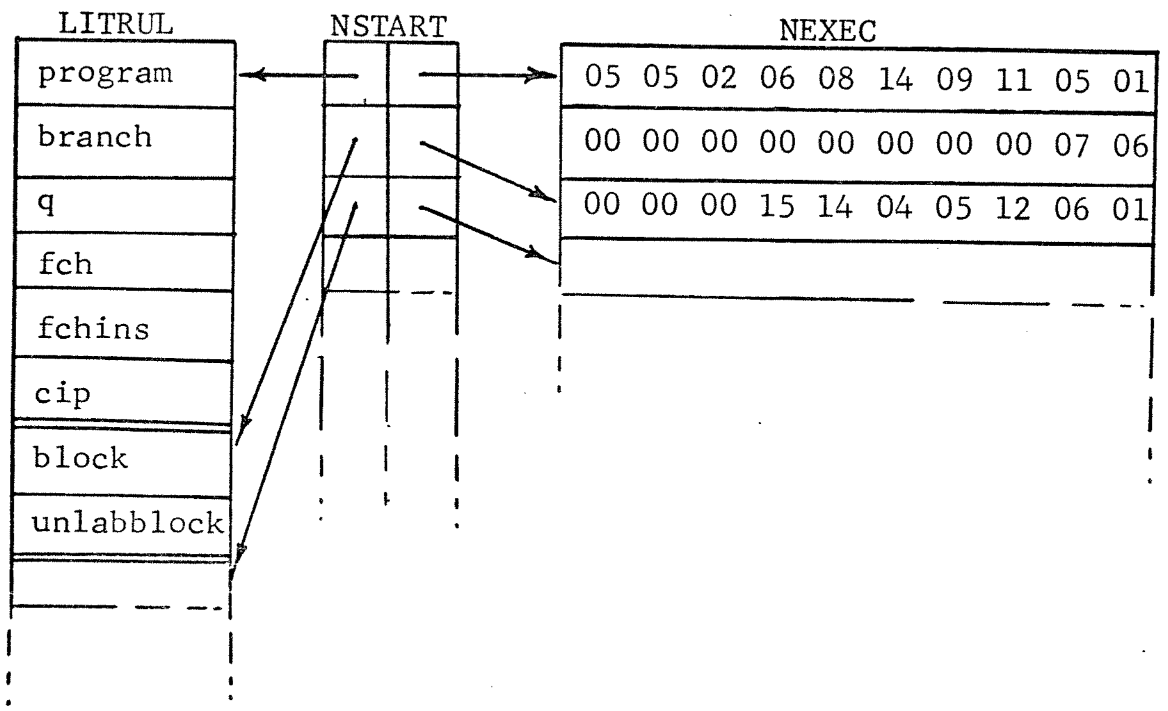


Figure 7. Table structure generated by the specifier

The tables are structured as above. If N is the graph rule member, then NSTART (N) contains pointers to the positions in LITRUL and NEXEC where the list of literals and sequence of exec calls for rule N are stored. Exec calls are packed 10/word right-to-left with 0-fill in the last word.

These tables then contain representations of the right-side graphs in terms of sequences of graph-building semantic exec calls. Each rule of the pair grammar now is a set of instructions for doing the actual graph-building. Before proceeding on with the description of other parts of the system, it is both interesting and enlightening to note the structure taking form. If one views the entire graph-building system as a "virtual machine", then these exec calls become the primitive operations of such a machine. The mini-language processor becomes a traditional compiler in that each right-side graph rule read in represents a statement in a higher-level language which requires translation into a sequence of primitive operations. The complete set of graph rules form a program, whose "compiled code" is conveniently stored awaiting interpretation. The Graph Generator sub-system is the interpreter of this virtual machine, executing the "statements" of the source program not sequentially but as directed by output from the string-representation Parser.

Source Language Parser

The segment of the TWS which parses the source program turned out to be the most trivial part, but only because some very powerful heuristics could be employed by it. Written in SNOBOL4, the parser makes use of the pattern-matching and tracing facilities built into the SNOBOL system.

The grammar of the source language, rather than being read in as data, is defined as a very large pattern in the SNOBOL parsing program. The left-side grammar itself is expressed, using SNOBOL statements, as a sequence of pattern-valued variables, each used in subsequent statements and all building the giant pattern 'PROGRAM' (or whatever the language's starting symbol is). Within each of these sub-patterns, conditional value assignments are made to elements of a V-array. The indices of these array elements correspond to the pair grammar rule numbers that the associated pattern will match.

Once this pattern is built, it merely has to be inserted into the SNOBOL program. With the appropriate programmer-defined trace turned on, an attempt can then be made to match the input string (the source program) with the pattern 'PROGRAM'. The programmer-defined trace saves a list of the index values of the V-array which were referenced. Hence, if a successful match is performed, the list saved is the sequence of matches made in the successful path, and this forms the parse tree.

As an illustration, consider the following language:

```
1: <program> ::= <addend> + <addend>
2: <program> ::= <addend> - <addend>
3: <addend> ::= <term>
4: <addend> ::= <term> * <addend>
5: <term> ::= X | Y | Z
```

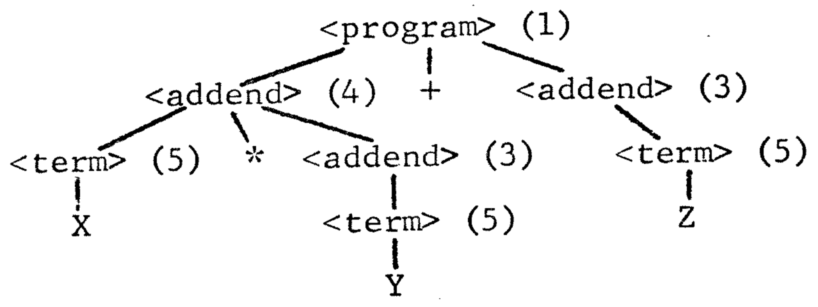
The SNOBOL pattern structure, as defined above, to match it would be as follows:

```
+       PROGRAM = (ADDEND '+' ADDEND) . V[1] |
                  (ADDEND '-' ADDEND) . V[2]
        ADDEND = *TERM . V[3] | (*TERM '*' *ADDEND) . V[4]
        TERM = ('X' | 'Y' | 'Z') . V[5]
```

The use of unevaluated expressions accomplishes three things: it allows the use of forward references and recursive definitions, and it saves space.

The trace routine merely stacks the V-index onto a list whenever that array element is changed in value; if the rule is one which matches only literals (such as 5), then the index number is preceded by a minus sign and the actual literal matched follows it.

The string X * Y + Z would thus be parsed as follows (the rule number is in parentheses following the nonterminal parsed):



The SNOBOL program's output list for this string would then be:

1 3 -5 Z 4 3 -5 Y -5 X

The complete listing of the SNOBOL program including the ALGOL grammar is located in Appendix E. Several attempts have been made to optimize the program specifically for the ALGOL example. For instance, in the listing shown, code to break the incoming source string into separate strings of "begin-end" pairs was used to break the pattern-matching job into small pieces. The ordering of alternatives, particularly in recursive rules, was found to be quite important in optimization.

Nonetheless, the parser is quite straight-forward, and, for the most part, a switch to another left-side language would merely entail a re-coding of the pattern for that language. The generality of this parser is such that any language whose grammar can be expressed as SNOBOL code can be accepted by it.

Graph Generator

The final stage of the TWS comes with the actual generation of the program graph. The language chosen for this stage was GROPE, developed at the University of Texas by Friedman and Slocum (6). Its graph construction and processing facilities are usable as FORTRAN functions, so that the language may be easily interfaced with other FORTRAN sub-programs. Additionally, the use of FORTRAN as a host language for GROPE provides the added advantages of speed, ease of use, and flexibility.

1. GROPE description: The GROPE graph-processing functions are quite intuitive, and an informal discussion of the nomenclature as it applies to this work should suffice:

The basic GROPE items fall into three classes: atomic values, elements such as nodes, lists, graphs, sets, arcs, etc., and readers. Every element within a GROPE graph structure (graphs, nodes, arcs, etc.) possesses an _object_ and a _value_. The _object_ is the element's name or label--that by which it can be referenced. A _value_ is said to be _hanging_ from an element; if nothing is hanging, the element's value is null. Many system sets exist which contain, for example, all the graphs generated, all the nodes on a certain graph, all the nodes with a given object, and the like. The existence of an element may generally be determined by its presence in one of these sets. _Readers_ are useful elements which allow tracing through graphs, lists, system sets, and the like, examining each of the elements one at a time. Within any particular graph, an _entry node_ exists which is the value of that graph. Every _arc_ on a graph has a _tonode_ and a _fromnode_, the definitions of which are obvious. Arc system sets which are helpful include the sets of all incoming/outgoing arcs from a particular node (_RSETI_, _RSETO_).

Within the specific use of GROPE for this program graph model, restrictions are placed on some aspects of the graphs generated. Every graph _must_ have an entry point node. A node may have a programmer-defined object (used as its label) and/or value; if he does not designate an object, the system generates an atomic integer for it. A node's value may be null, an atomic value, or a graph. Graphs never have designated objects but are always referred to as the "value of a certain node". Arc labels are the objects of arcs; if unlabeled, an arc has an object of 0. A double-ended arc is represented by two arcs on the graph generated between the same two nodes, each pointing different directions. All nonterminal nodes have the atomic value 'NONTERM' as their object; their values are their atomic names.

2. Generator Structure and Operation: With this background of GROPE, then, the third and final stage of program graph construction can be detailed. Figure 8 shows the block diagram from parse tree input to graph output. The routine functions are outlined there.

When the parse tree has been generated and the tables of exec calls compiled, control is given to CRGRAPH. Using a scanner for the parse tree, it calls RSIDE for each of the rules scanned. Within this routine, which acts as the interpreter for the virtual machine, table pointers are followed to the starting locations of the lists of literals needed for that rule and the sequence of exec calls required to generate the graph. Using UNPACK, RSIDE calls EXECSEM successively, passing the exec number and, if required, literal needed. This continues until all the packed exec numbers for the parse rule are processed (0 is unpacked from the table), then CRGRAPH continues with the next parse rule. The semantic exec routine EXECSEM is merely a set of separate graph building code sequences which manipulate a common semantic stack, retrieving literals to be used as node labels, etc. and placing the new nodes, graphs, and arcs back onto the stack. The several worker routines mentioned in Figure 8, GETNT, CRTNODE, and RDECOD/NDECOD, are some routines many of the code sections have in common. A complete lising of both the Graph Specifier and Generator is included in Appendix F.

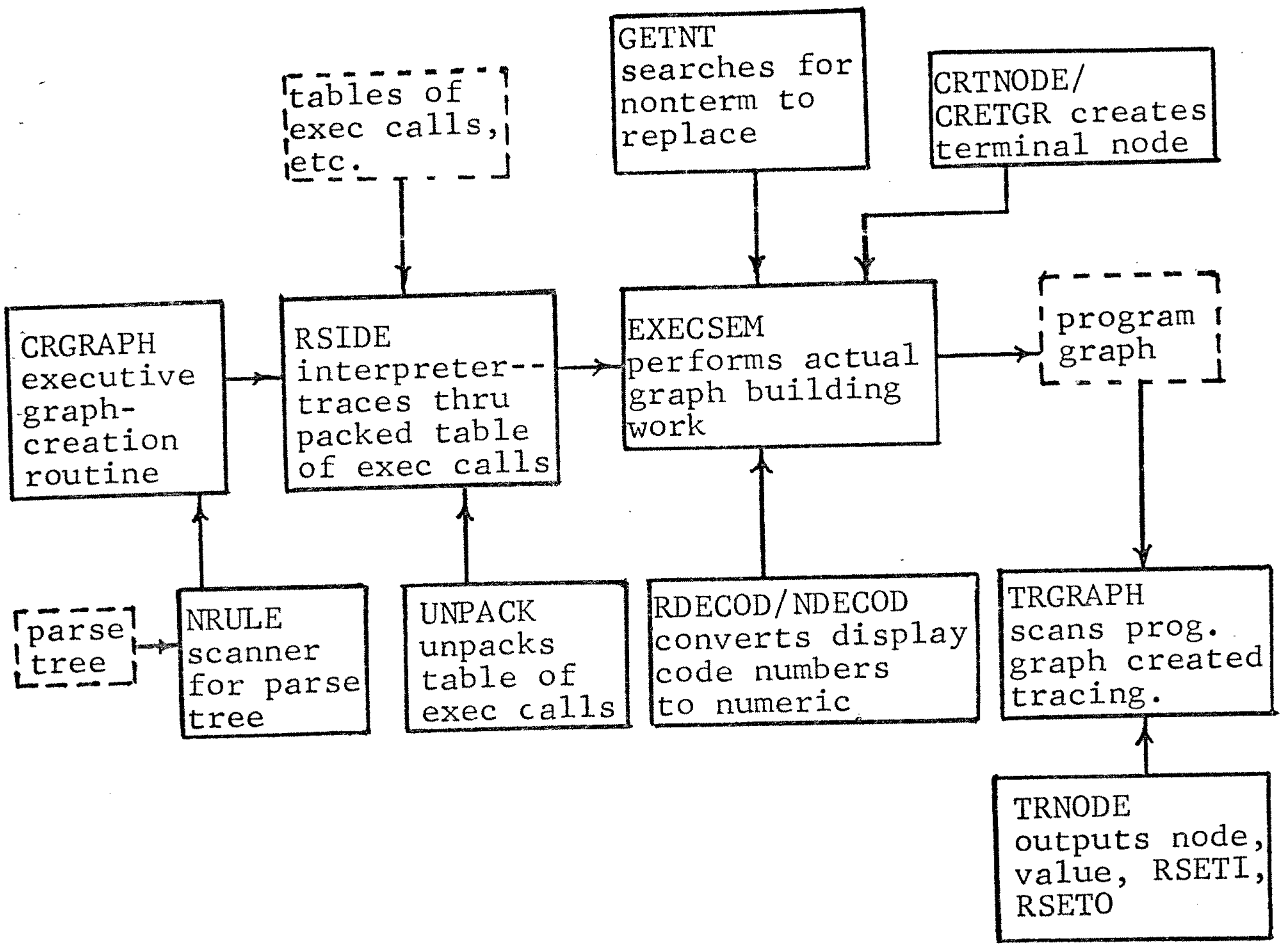

tables of exec calls, etc.
GETNT searches for nonterm to replace
CRTNODE/ CRETGR creates terminal node
CRGRAPH executive graph-creation routine
RSIDE interpreter-- traces thru packed table of exec calls
EXECSEM performs actual graph building work
program graph
parse tree
NRULE scanner for parse tree
UNPACK unpacks table of exec calls
RDECOD/NDECOD converts display code numbers to numeric
TRGRAPH scans prog. graph created tracing.
TRNODE outputs node, value, RSETI, RSETO

Figure 8. Graph generator structure.

The final program graph is generated within the GROPE system; no capabilities currently exist for saving it on external storage. The programs which eventually use the graph (such as the interpreter) will have to be executed from the same main program as the Graph Generator.

3. <u>Graph Output</u>: Once the graph has been generated, it is desirable to see if the translator has done its job properly. For small programs, at least, a hand-translation can be made fairly quickly and compared to the actual program translation if a means of graph output can be found.

The routines TRGRAPH and TRNODE are the first faltering steps in that direction. Since the program graph is really an intermediate step in the larger task of correctness studies, it was decided to not spend too much time on graph tracing/output facilities. Certainly a conversion back to the input form for output would involve too much pattern-recognition to be practical.

The method of output detailed in Figure 9 was finall chosen as a compromise. This type of trace was easily programmable, and the output, while not easily readable, is unambiguous and may be used to re-draw the entire program graph quickly and accurately. A sample of the graphs generated using test ALGOL programs is included in Appendix G. Both the graphical and trace output forms are included for comparison.

```
(NODE: <n-label> VALUE: <value>)
(RSETI: set of inpointing arcs, in (<a-label> <from-
     node> format))
(RSETO: set of outpointing arcs, in (<a-label> <to-
     node> format))
     If node is a graph, indent two spaces and begin
     tracing the nodes on its value graph.  Otherwise,
     output "next" node on same graph.  Order of node
     trace is determined by position in the NDSET(graph).
```

Figure 9. Graph output format.

4. <u>Tracing facilities</u>: As a final note, the interested user should be aware of the many tracing facilities built into the design of the TWS. The first card read in during system operation is the tracing flag; in the comment section of the main program are the octal digits corresponding to each of the results below. The folowing trace options are available by setting their associated bit number:

Bit 1: Output the FEP table. Additionally, print the jump pointers and left and right side element counts.

Bit 2: Print the hashed reserved word table. It is currently a 54-place table using a hash bucket/chaining techniqu

Bit 3: Trace the syntactic stack during the initial parse of the graph grammar rules. This is useful when syntacti errors in the mini-language recur.

Bit 4: When Bit 3 is set, also trace the semantic stack durin initial parse.

Bit 5: During graph generation, trace semantic stack. This is quite useful if an error persists in graph generati

Bit 6: Print literal table, packed exec sequence table, and starting pointer table generated by the specifier. An example of this output is located in Appendix D.