FUNCTIONAL SEMANTICS OF PROGRAMMING LANGUAGES

by

William Davis Flannery

Memorandum No. ERL-M395

August 1973

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Functional Semantics of Programming Languages

Abstract

William Davis Flannery

A definition of a programming language *L* consists of two parts, as follows:

1) the syntax of *L*. The syntax of *L* specifies which symbol strings denote legal constructs (including programs) in *L*.
2) the semantics of *L*. The semantics of *L* provides a "meaning" for legal constructs specified by the syntax.

We will assume that the syntax of *L* is given by a context-free grammar. Let P_L denote the set of legal programs in *L*; then the "meaning" of a program $\underline{P}$ in P_L is defined to be the function from input values to output values calculated by $\underline{P}$. Since the semantics of *L* provides a method of determining the meaning of every program in P_L, it can be expressed as a function

$$Sem_L : P_L \rightarrow (D_1 \rightarrow D_2)$$

where D_1 is the set of possible input values of programs in P_L; D_2 is the set of possible output values of programs in P_L; and

$$Sem_L(P) = \alpha \varepsilon (D_1 \rightarrow D_2)$$

where α is the function computed by $\underline{P}$.

In Chapter 1, several methods of semantic definition are discussed using the above perspective as a frame of reference.

In Chapter 2, a number of mathematical results are reviewed, and a new method of defining the semantics of programming languages is presented. In this method, Sem_L is defined by the composition

$$Sem_L = Y_R \circ Trans_{L-R}$$

where $Trans_{L-R}$ maps programs in L to sets of recursive functional equations (this "language" is denoted by R) and Y is the minimal fixed point operator, i.e., if $R \epsilon R$, then $Y_R(R)$ is the unique minimal fixed point of $\underline{R}$.

In Chapter 3, a format for programming language definitions is given in complete detail. There is also an account of a SNOBOL4 program, which, when given a definition of a language L in the proper format, acts as an interpreter for programs written in L.

Chapter 4 contains complete formal definitions of ten languages presented in the paper "Ten Mini-Languages: A Study of Topical Issues in Programming Languages" by Henry F. Ledgard. The ten mini-languages were designed to isolate and illustrate a number of complex features found in current programming languages. These features include the notions of assignment, locations, transfer of control, functions, parameter passing, static type checking, dynamic type checking, data structures, string manipulation, and input/output. These definitions were used in conjunction with the semantics-based interpreter described in Chapter 3 to interpret the sample programs for each of the ten mini-languages given in Ledgard's paper. These results constitute Appendix I.

The advantages of a fixed-point characterization of the function computed by a program, and our particular method of defining

$Trans_{L\text{-}R}$, are discussed in Chapter 5. Our definitional method is compared with other current methods, and directions for future research are indicated.

Appendix II contains a listing of the semantics-based interpreter discussed in Chapter 3.

Possible improvements to the definition format are indicated in Appendix III, and definitions of mini-languages 1 and 4 are given in an easily readable format.

TABLE OF CONTENTS

Chapter 1. Programming Language Semantics, A Historical Perspective

Abstract

A general review is made of the semantic methods used in the definition of FORTRAN, ALGOL-60, and PL/1, along with some of the research literature that accompanied the definition of PL/1. A general framework encompassing these definitional methods is presented. Two alternative methods represented by the work of Floyd and Scott are reviewed.

1.0 Introduction

This chapter will be a review of the methods that have been used for the specification and definition of programming languages. Both practical methods, that have been used in the definition of large, complex, and widely used languages, and theoretical methods that have never escaped from the laboratory, will be discussed. The aim is to provide a reasonably complete overview of the methods and ideas currently creating the most interest in the field of programming language semantics. Except for the cases of LISP and PL/1, which are both defined using an interpreter, the methods that have appeared in journal articles have not been used to define reasonably large and complete programming languages that are actually used. This correctly indicates that the research in this area is just getting started, and most of the theoretical methods

Chapter 1: Programming Language Concepts
Motivation and Perspective

Abstract

A general outline is given of the concepts and issues involved in the definition of programming languages and PL/I. A comparison between PL/I and other programming languages is presented and reasons for the existence of the various differences are proposed. The objectives and principal features of PL/I are discussed briefly.

1.1 Introduction

This chapter will be a motivation for studying what have been used for the specification and verification of programming languages (both procedural and nonprocedural) that have been used in the definition of these languages, compilers, and widely used languages, and theoretical methods of language description that have been proposed. The problems encountered in the design of PL/I will be discussed and ideas concerning the most fruitful areas of research in the field of programming language semantics. Included for the sake of PL/I, which is both advanced and widely used, is a detailed overview of the methods and ideas currently employed in the definition of PL/I. Whether or not these methods are adequate, the details have been presented to permit a reasonably complete overview of the problems encountered in defining a programming language. Finally, the author's personal opinions concerning the relative merits of the various methods are presented. Although these opinions are based on experience with PL/I, they are not necessarily unique to PL/I. Many of the problems encountered in defining PL/I are present in other languages as well.

are not currently well enough developed to simultaneously satisfy the four goals of a language definition:

1) To completely define a complex language.

2) To provide language users with a clear, concise understanding of language features.

3) To provide the compiler writer with enough information to implement the language.

and

4) To facilitate mathematical proofs of properties of programs written in the language.

Programming language definitions consist of two parts: first, the specification of the syntax, i.e., how legal programs can be built up from the elementary constructs in the language; and second, the specification of semantics, i.e., how to determine the "meaning" of legal programs described by the syntax. A great deal of research has been done in the area of programming language syntax, and it is now generally agreed that a programming language should have a syntax that can be specified using a context-free grammar. It will be assumed that the reader is familiar with the basic theory of context-free grammars, and this is the technique that will be used to specify programming language syntax in this thesis.

There is certainly at this time no concensus as to how programming language semantics should be defined, or even, what is the meaning of "semantics" itself. A working definition of semantics may be given as follows: each program P in a language *L* defines a

map from input values to output values, and essentially, this is what the program "means". Thus, the semantics of L is a map $\underline{Sem_L}$ from programs in L to functions, such that $Sem_L(P):D_1 \rightarrow D_2$ is the function computed by $\underline{P}$. The typical way of describing semantics is still an ad hoc informal description of the "action" of various constructs in the language, based on a large body of assumed common knowledge.

Why is this the case, and what can we expect to derive from a formal semantic theory? First, the ad hoc methods have reasonably well satisfied the first three of the goals mentioned earlier. Also, they are easy to understand, and this has made the informal approach the path of least resistance. What can we expect to derive from the correct, formal theory? Consider the corresponding case of programming language syntax. The formalism for generative grammars was developed in the fifties before there was any idea of the different classes of languages, undecidability results, equivalences with special automatons, automatic parser generators, and the like. Thus, the correct formalism led to a number of results that were not even imagined when the formalism was developed. Whether or not the same thing will occur in the area of semantics is not known, but there should be hope; clearly, there is an impetus to arrive at the proper formalism even prior to seeing the resulting consequences.

1.1 The Definition of FORTRAN

To provide historical perspective, we will first consider the definitions of FORTRAN and ALGOL-60, which serve to illustrate the

dominant views on language definition at the time of their creation (1956 and 1962, respectively).

Even though FORTRAN was created in 1956, there was no formal definition of the language until 1964. At that time, the Association for Computing Machinery (ACM) thought it necessary to provide a definition of FORTRAN mainly for purposes of standardization, as there were many versions of FORTRAN extant at that time which were incompatible in some respects. The advantages of standardization are obvious, but FORTRAN had already existed for eight years with no more formal definitions than the language manuals written for various compilers. The ultimate definition was provided by the compiler, which was the judge of syntax and semantics, failing to compile any program not syntactically correct, and translating any correct program $\underline{P}$ to machine language program P_{Mach}. Then $Sem_L(P)$ could be determined by executing P_{Mach}. This method may still be considered by some to be a reasonable way of defining at least the semantics of a programming language. Since no one can understand the intricacies of an actual compiler (other than, hopefully, its author), this method can be used only with the help of language manuals which define the syntax and semantics informally.

1.2 The Definition of ALGOL

By the time of the development of ALGOL-60, there had been considerable research in the area of language syntax and phrase structure grammars, originated by Chomsky in his study of natural languages (Ch 57), and adapted by Backus in a formalism for defin-

ing ALGOL. In ALGOL-60, the syntactic and semantic definitions are explicitly separate. The syntax is specified using meta-linguistic formulas written in what has come to be known as Backus Normal Form (BNF). With a small number of BNF rules, it is possible to completely characterize, identify, and break down into structural units any legal ALGOL-60 program.

A great deal of progress had taken place in the techniques for specifying syntax, but there was no corresponding advance in semantic techniques. The semantics of ALGOL-60, in its formal definition, were specified informally, in the manner of a programming language manual. The ALGOL-60 report is broken into the following sections:

1. Structure of the Language
   - 1.1 Formalism for Syntactic Description
2. Basic Symbols, Identifiers, Numbers, and Strings. Basic Concepts
   - 2.1 Letters
   - 2.2 Digits
   - 2.3 Delimiters
   - 2.4 Identifiers
   - 2.5 Numbers
   - 2.6 Strings
   - 2.7 Quantities, Kinds, and Scopes
   - 2.8 Values and Types
3. Expressions
   - 3.1 Variables
   - 3.2 Function Designators
   - 3.3 Arithmetic Expressions
   - 3.4 Boolean Expressions
   - 3.5 Designational Expressions
4. Statements
   - 4.1 Compound Statements and Blocks
   - 4.2 Assignment Statements
   - 4.3 GOTO Statements
   - 4.4 Dummy Statements
   - 4.5 Conditional Statements
   - 4.6 FOR Statements
   - 4.7 Procedure Statements
5. Declarations
   - 5.1 Type Declarations

5.2 Array Declarations
5.3 Switch Declarations
5.4 Procedural Declarations

Generally, the title of each subsection represents a syntactic class. Each subsection is broken into three (and sometimes more) sections giving the syntax, the semantics, and examples. The syntax is given in BNF, and the semantics is given in one or more paragraphs of English text discussing the result of "executing" the construct, its relation to other parts of the program, etc., as appropriate. The following excerpt from the ALGOL report (Na 60) illustrates how ALGOL was defined:

> 4.2. ASSIGNMENT STATEMENTS
> 4.2.1. Syntax
>
> (left part)::=(variable):=
> (left part list)::=(left part)|(left part list)(left part)
> (assignment statement)::=(left part list)(arithmetic expression)(left part list)(Boolean expression)
>
> 4.2.2. Examples
>
> s:=p[0]:=n:=n+1+s
> n:=n+1
> A:=B/C-v-q × S
> s[v,k+2]:=3-arctan(s × zeta)
> V:=Q>Y^Z
>
> 4.2.3. Semantics
>
> Assignment statements serve for assigning the value of an expression to one or several variables. The process will in the general case be understood to take place in three steps as follows:
>
> 4.2.3.1. Any subscript expressions occurring in the left part variables are evaluated in sequence from left to right
>
> 4.2.3.2. The expression of the statement is evaluated
>
> 4.2.3.3. The value of the expression is assigned to all the left part variables, with any subscript expressions having values as evaluated in step 4.2.3.1.

The ALGOL report was for some time the prime example of a formal definition of a programming language, and ALGOL was the first programming language to have its syntax specified using a BNF grammar; obviously, however, it is stretching things a bit to call the above semantic description "formal".

1.3 Compiler-Oriented Semantic Methods

The first truly formal method to be considered is a formalization of the idea of specifying the semantics of a programming language by using a compiler to translate *L* into a machine language for a real or "abstract" machine. While this was not the first formal method for semantic definition (McCarthy's definition of LISP, using an interpreter, preceded it), it does correspond to one intuitive notion of how a programming language should be defined (there are others). The idea rests on the assumption that the semantics of the object language (originally machine code) are completely known, and are in fact determined by a particular computer. Thus, if P is a program in *L*, *M* a machine language whose semantics Sem_M are known, and $Trans_{L-M}(P)$ is a program in *M*, then $Sem_M(Trans_{L-M}(P))$ is the function from inputs to outputs calculated by P. Hence, all we need to do is to define $Trans_{L-M}$ and define $Sem_L = Sem_M \circ Trans_{L-M}$. In fact, most semantic methods are characterized by this equation; however, the nature of *M* may vary widely.

As stated above, *M* could be an actual machine language, and Sem_M would be "determined" by the machine itself. $Trans_{L-M}$ could be a compiler which translates programs in *L* to machine language

programs. There are two principal objections to defining semantics in this manner. First, it does not seem reasonable to base a formal description on a real machine. Not everyone has the same computer, hence someone must be dissatisfied. Also, any given machine may very well be obsolete in a few years, rendering the formal definition obsolete also. The second objection is the complexity of the translation process. If $Trans_{L-M}$ is defined by a compiler, then the semantic definition will be useless for the simple reason that it will be too complex and obscure.

Both of these objections to a compiler-oriented specification were overcome (to an extent) in a definition of an ALGOL-60 look-alike called EULER by N. Wirth and H. Weber in an article published in the Communications of the ACM in January of 1966 (Wi 66).

The first objection is overcome as follows: rather than choosing as the object language the machine language for some actual computer, Wirth and Weber created a new machine-like language, which was presumably free of the shortcomings and idiosyncracies of a real machine language, and the meaning of which they claim to be obvious. Since the language is very simple, any experienced programmer would be inclined to agree with them on this point. By this device, their semantic definition is no longer tied to a particular machine.

The second objection concerns the complexity of translating a high level language to a low level language. Wirth and Weber dispense with a compiler (described by a program) altogether in favor of the following more formal way of defining the translation. The syntax of EULER is given by a context-free grammar which is

shown to be, in fact, a precedence grammar. An algorithm for parsing sentences in a language generated by a precedence grammar is given in complete detail, with all data structures, stacks, etc. specified explicitly. Associated with each production in the grammar is a set of actions, which produce output (the object language translation) and/or affect internal variables or data structures. A program can be translated by parsing the program, and whenever a reduction is made, the corresponding actions are taken. Note that the order of the reductions is determined by the precedence algorithm, and also that there is no back-up. Thus, $Trans_{L\text{-}M}$ is specified by the precedence algorithm and the sets of actions associated with the productions in the grammar for G. This technique reduced the size of the description of the translator to manageable proportion, making it possible for Wirth and Weber to give a concise definition of a very complex language.

Wirth and Weber have, in fact, described a compiler-compiler (for precedence languages), minus the final step of translating the abstract object language to an actual machine language. While they used their compiler to specify EULER, they could have specified different languages by modifying the grammar and the associated action sets. Others [e.g., Feldman (Fe 66)] have used variations of this approach to define formal semantics of programming languages, and to produce actual compiler-compilers.

1.4 Interpreter-Oriented Semantic Methods

Another approach to semantics was introduced by J. McCarthy in his definition of LISP in 1960 (McC 60). This method was subse-

quently (with a number of embellishments, to be sure) used to define PL/1, and we will for the most part consider this latter definition, as LISP is an atypical programming language, and there are a number of problems that were not encountered in the definition of LISP that were encountered in the definition of PL/1 (and would be encountered in defining any other algebraic language).

This approach can be considered as the reverse extrapolation of the compiler-oriented approach discussed in the previous section. In that approach, there is a trade-off between the power of the object language and the complexity of the translation. That is, the translation can be simplified by increasing the power of the object language. This is often done when designing or specifying an ALGOL translator (e.g., J. Morris, 106 Notes, University of California). However, it then becomes more difficult to specify the semantics of the object language, which, past a certain point, can no longer be considered self-evident. The semantics of the object language are specified by designing an "ideal" or abstract computer which executes the object language as its "machine language". An ALGOL machine, designed to execute the object code of an ALGOL translator, may be considerably more powerful than any real machine, and the translation correspondingly simplified. If the abstract machine is made powerful enough to execute the source language itself, the translation phase can be eliminated altogether. There are two factors making this idea attractive:

1) It is difficult to give a formal specification of a translation about which anything can be proved. The research

in this area of language theory is not extensive at all (Sections 2.2 and 5.0 contain more information on formal methods of specifying a translation).

2) In translating a program in a powerful language L to a program in a less powerful language L', it is necessary to introduce details into the translated program that were not present in the original program, e.g., if L is recursive and L' is not, then the translated program will necessarily have explicit stack manipulation operations. The resulting increase in size of the translated program might make it more difficult to understand than the original program, despite its more elementary operations. If possible, we would like $Sem_L(P)$ to simplify our understanding of P rather than obscure it in a mass of detail.

1.4.1 LISP

If the above approach is adopted, the major problem is defining the "abstract machine" that executes the source language. One of the ways of doing this is to write a program (called an interpreter) which defines the "central processor" of the "abstract machine". Regarding the interpreter as a two-place function, then the semantics of L are expressed by $Sem_L(P)(X)=Int\ (P,X)$. This technique was first used by J. McCarthy in his definition of LISP in 1960 (McC 60). In this definition of LISP, the "abstract machine" which interpreted the LISP programs was defined by a program written in LISP. McCarthy claims the program defining the "abstract machine" is so simple that its meaning is obvious. LISP is a very simple language in terms of the number of different le-

gal constructs, and the program to interpret LISP programs is very small. As for the problem of circularity, McCarthy's answer was, "Nothing can be explained to a stone"!

1.4.2 PL/1

The method used in defining PL/1 is an extension of the ideas McCarthy used in defining LISP. Similarly, there is no translation of the source program. However, the "abstract machine" or interpreter is defined in a totally new and unique way. This was necessitated by the overwhelming size and complexity of PL/1, and a great number of new ideas were used in the definition of the PL/1 interpreter.

The input to an interpreter is a source language program, which must be effectively parsed before it can be interpreted.* Provided with a BNF grammar, it is possible to "mechanically" construct a parser; however, the resulting parser is a complicated algorithm and if possible it would be desirable to avoid incorporating this algorithm into the interpreter. To do this, McCarthy provided an alternative formulation of syntax (McC 63b) which, rather than describing how a sentence is built up from more elementary sub-sentences, describes how a sentence is broken down into its more elementary sub-sentences. An "abstract syntax" for a language *L* is a set of functions and predicates which can effectively be used to parse any sentence in *L*. If, in the BNF syntax for

---

*A LISP program is a structured data object which happens to be written in linear form; thus, a LISP program is its own parse tree.

L, there is a rule of the form

<SUM>::=<addend>+<augend>

there will be in the abstract syntax for *L* the predicate IS-SUM and functions addend and augend. IS-SUM applied to any string returns TRUE if the string can be parsed as a sum, and FALSE otherwise. If IS-SUM(α)=TRUE, then addend(α)=the substring that parsed as the addend, and augend(α)=the substring that parsed as the augend.

The functions and predicates that constitute an abstract syntax for PL/1 are incorporated into the interpreter for PL/1. This makes it possible to immediately parse any legal PL/1 program, and can be viewed as equivalent to having the input to the interpreter be not a program P itself, but rather the parse tree for P. That, essentially, is what abstract syntax is all about.

The state of the interpreter at any time is represented by a tree-structured Vienna Object (Vienna Objects, and a set of operators for manipulating Vienna Objects, will be defined and discussed in Chapter 2), called the state descriptor. This data object has one sub-tree called the control tree. Rules are given for transforming the entire state descriptor based on the current terminal vertices in the control tree. The transformation is expressed using a set of operators especially designed for this purpose. When the state descriptor reaches a terminal state, the computation is complete, and the final state descriptor is defined to be the result of executing the original program, i.e., $Sem_L(P)(X)=Int_L(P,X)$.

Other parts of the state descriptor (besides the control tree) include an environment subtree giving the current unique name associated with each identifier, a denotation subtree associating unique names with values, a dump subtree containing a history of the computation, and subtrees representing other information structures, depending on the state of the computation. The state descriptor is a complex object, and the evaluation of a control tree with a terminal vertex that represents a procedure call, say, will cause changes to be made in several parts of the state descriptor. It is possible to express these transformations succinctly in a special data manipulation language, and this is one of the primary reasons accounting for the power of the abstract machine as compared to that of a conventional computer.

The use of translators and/or definitional interpreters represents the mainstream in the field of semantic definition. These are the only methods that have been used to give formal descriptions of reasonably large programming languages. Even so, not much research is being done on these techniques as methods of formal definition. There have been a few papers on definitional interpreters by Wegner (e.g., We 72), and a review and classification of different definitional interpreters by J. Reynolds (Re 73).

1.5 Floyd's Approach to Semantics

There are two other approaches which I will discuss because of their importance in the field of semantics, and their relevance to the method which will be presented in Chapter 2. The first was introduced in a paper by Floyd in 1967 (Fl 67). The main objective

of this paper was to introduce techniques for proving the partial correctness of programs, and Floyd introduced the now familiar method of assertions. This method consists of associating assertions about the current state of the program variables with various points in the program. Suppose we have assertions A_1 and A_2 associated with points p_1 and p_2 in $\underline{P}$, respectively, and there is a path in $\underline{P}$ from p_1 to p_2. To conclude the correctness of $\underline{P}$, we must show that if A_1 is true at p_1, and the path from p_1 to p_2 is executed, then A_2 is also true. This process is called "verifying the path p_1 to p_2". The partial correctness of $\underline{P}$ is shown when all paths between assertions (containing no intermediate points that have assertions) are verified, and can be stated as follows: when the assertion associated with the entry point of the program is true, and the program terminates, then the assertion associated with the terminal point of the program will be true.

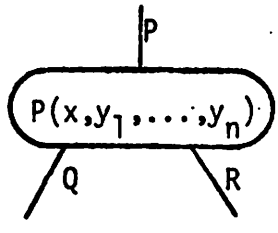
Suppose we have assertions A_1 and A_2 associated with points p_1 and p_2, respectively, and the path from p_1 to p_2 consists of one statement, $\underline{c}$, only. How does one go about verifying that when A_1 is true at p_1 and $\underline{c}$ is executed, then A_2 is true? Clearly, we have to consult the semantic definition of *L* to determine the effect of $\underline{c}$ on the program variables. Floyd, however, presumably found that the methods of semantic definition available to him at that time were of little use, and as a result, he suggested a new method of semantic definition. The problem was, in a semantic definition similar to the ones that had been given for EULER, or PL/1, in order to verify that A_2 follows from A_1 after executing

c, it is necessary to get into the workings of either a complicated translator (as in the case of EULER) or a complicated interpreter (as in the case of PL/1). This would have been especially senseless since the effect of c was already known perfectly well intuitively. Floyd used his intuitive understanding of the statements in a simple language *L* to give a semantic definition that suited his purposes. His formal suggestion was as follows: let c be a statement in *L* (here we will be restricted to non-branching statements); the semantic definition of c will be a verification condition $V_c(P,Q)$, which is a logical formula with free variables P and Q. The interpretation is as follows: P represents an assertion about the variables of a program containing c immediately prior to the execution of c, and Q represents a statement about the variables immediately after execution of c; V(P,Q) is a logical expression such that if $V_c(P,Q)$, for a particular P and Q, is a tautology, the path (consisting of the statement c) is verified, and if $V_c(P,Q)$ is not a taugology, the path is not verifiable (for assertions P and Q).

For example, the semantic definition of an assignment of the form $X \leftarrow f(x,y_1,\ldots y_n)$, where P and Q can be written $P(x,y_1,\ldots,y_n)$ and $Q(x,y_1,\ldots,y_n)$ is the formula

$$\exists\, x_0(x=f(x_0,y_1,\ldots,y_n) \wedge P(x_0,y_1,\ldots,y_n) \Rightarrow Q(x,y_1,\ldots,y_n)).$$

The definition of a conditional branch, which can be represented by a flowchart node



is the formula $(P \wedge p(x,y_1,\ldots,y_n) \Rightarrow Q(x,y_1,\ldots,y_n)) \wedge ((P \wedge \sim p(x,y_1,\ldots,y_n) \Rightarrow R(x,y_1,\ldots,y_n))$. Using a semantic definition of this type, it is possible to verify the paths between assertions in a program, and hence, prove the partial correctness of programs.

1.6 Scott's Approach to Semantics

The final method to be discussed is due to D. Scott and C. Strachey (Sc 70), and has not been used to define anything resembling a full-scale programming language. In fact, this method is somewhat difficult to pin down as to how it applies to typical algebraic programming languages. Generally stated, Scott's thesis is that each construct in the language, that is, a member of a syntactic class, should denote something. Just what this "something" should be involves a lengthy discussion of recursive domains necessary to represent the functions and values represented in a program. Basically, the idea is that the constructs will denote functions, and the function denoted by the entire program is the function the program calculates. The semantics of a language consists of a function for each syntactic class, mapping elements of that class to the object it represents. For example, suppose a language *L* has commands Cmd which map states to states, and the set of states is represented by S. Then, the semantics of commands is a function

$$C: Cmd \rightarrow [S \rightarrow S].$$

The semantics for expressions might be a function

$$E: Exp \rightarrow [S \rightarrow V]$$

where V is the set of possible values of the expression, or if the evaluation of the expression might affect the state (through side

effects, for instance), then the semantic function might have the form

$$E:Exp\rightarrow[S\rightarrow S \times V].$$

The semantic equations of the different constructs in a production in the grammar can be related by functional equations; for example, if

Cmd→Cmd; Cmd is a production; and $Y\rightarrow Y_0; Y_1$

is an instance where Y, Y_0, and Y_1 are commands, then clearly we should have

$$C(Y)=C(Y_1) \circ C(Y_0).$$

The set of all these recursive functionals can be used to define the semantic functions by an application of the fixed point operator. That is, the minimal fixed point of this set of recursive functionals determines the semantic functions.

Neither Floyd's method nor Scott's method involves real or abstract machines; however, only Scott's method gives an expression for the function calculated by a program, and this is one of the primary aims of a semantic definition. Scott was also the first to use fixed point theorems in defining programming language semantics, even though the fixed point theorems he used have been known for some time [they originally appeared in Kleene (K1 52)].

1.9 Conclusion

There are a number of different ways of defining the semantics of programming languages, but the surprising thing is that none of them are accepted as satisfactory by a great percentage of the peo-

ple working in this field. For this reason, there has been a flurry of activity in this area, and hopefully there will be results forthcoming to match the effort.

Chapter 2. A New Method for Specifying Programming Language Semantics

Abstract

A number of techniques and results are reviewed; these include: Knuth's formalization for specifying the "semantics" of context-free languages; the subset of the Vienna Definition Language consisting of the data manipulation operators; an easy technique for translating flowcharts to recursion equations; and a fixed-point theorem characterizing the function defined by a set of recursive functional equations. A new technique for specifying programming language semantics, which makes use of each of the results above, is then presented.

2.0 Introduction

The method presented in this thesis is an extension of a method presented by W. D. Maurer in "A Semantic Extension of BNF". Professor Maurer's work, in turn, made use of a number of techniques earlier developed by J. McCarthy and D. Knuth. The extensions incorporated into the new method make explicit use of a data language which is part of the Vienna Definition Language, and a number of results on the fixed points of sets of recursive functionals; these results can be found in papers by Lucas (Lu 67) and Manna, Ness, and Vuillemin (Ma 72).

It would be unreasonable to assume that the reader is familiar with all of these techniques and results, so the first part of this

chapter will be devoted to a detailed review of each of the above subjects.

Most of the concepts that appear in this thesis are contained in germinal form in two papers by J. McCarthy published in 1963 [(McC 63a) and (McC 63b)]. In particular, the concept of defining the semantics of a program by its effect on a state vector, and the initial formalization of state vector functions, appeared in these articles. The state vector formalism became the basis for the Vienna data manipulation sublanguage discussed in Section 2.1. Also, the idea of defining the state vector function associated with a program by a set of recursion equations appeared in these articles. We will be considering extensions of this work as reported by Manna (Ma 72) in Section 2.3. Hence, although there will be no explicit references to the work of McCarthy in the remainder of this chapter, his work was the starting point for the techniques presented in Sections 2.1 and 2.3.

2.1 The Data Manipulation Sublanguage of the Vienna Definition Language

The Vienna Definition Language (VDL) is a very complex language which can be used to give a formal description of a tree-structured PL/1 interpreter. We, however, will use only the most elementary subset of VDL, which will be referred to as the data manipulation sublanguage. This sublanguage consists of only three operators which can be used to build tree-structured objects from elementary objects, to modify existing tree-structured objects, and

to reference sub-trees of tree-structured objects. In the Vienna report (PL 66), these tree-structured objects are defined and given the name "Vienna Objects" (VO's).

Our goal is to represent a program by a state vector function. To do this, we need a mathematical representation of state vectors (VO's) and a method of functionalizing variable referencing and assignment. Why? The semantics of these actions are generally taken to be self-evident, or explained mechanistically, and, if the languages we wished to define were simple enough, we might be able to incorporate our intuitive understanding into our mathematical formalization by mimicking these actions by the very careful use of mathematical variables. However, in the presence of features like block structure, procedures and parameters, recursion, dynamic storage allocation, etc., naming and variable referencing and assignment become complex actions, and a uniform, explicit approach becomes a necessity. What does it mean to functionalize variable referencing and assignment? Let $\underline{S}$ represent a state vector, and let α be a variable. To functionalize variable referencing means to specify a function f_α such that $f_\alpha(S)$ is the value of α in $\underline{S}$. Note that f_α may not be simple; if, for example, α is a call-by-name parameter, f_α may be complex indeed. To functionalize variable assignment means to specify a function g_α for each variable α such that $g_\alpha(v,S)$ is a state vector $\underline{S}'$ equal to $\underline{S}$ except for the value of α, which is $\underline{v}$ in $\underline{S}'$. The functionalization of variable referencing and assignment can only be completely spec-

ified in the context of a complete semantic definition; the VDL data manipulation sublanguage is the first step towards that goal.

The notation of (PL 66) has been modified to make it reasonably easy to implement. The two main difficulties with the notation are the use of the string <a:b> to represent a two-place function (which returns the VO {(a,b)}), and the notation $s(t)$, where $\underline{s}$ is essentially any string, to represent the selection operation (if $t=\{(s_1,o_1),\ldots,(s_n,o_n)\}$, $s_k(t)=o_k$). Neither of these notations corresponds to normal functional notation, and they are difficult to recognize in a context where general functional expressions are allowed. For this reason, a functional notation for the Vienna Operators was created and will be used in the definitions in Chapter 4. In this chapter, both notations will be presented, with the new notation underlined.

There are two types of Vienna objects:

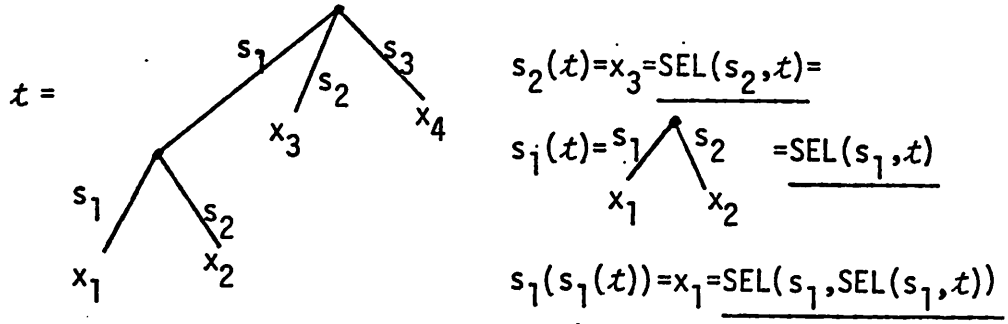
1) Elementary objects: an elementary object is an unstructured data value.

2) Composite objects: a composite object is a set of ordered pairs $\{(s_i,o_i)\}$, $i=1,\ldots,N$ where each s_i is a string, $s_i \neq s_j$ for $i \neq j$, and each o_i is a Vienna object either elementary or composite. The s_i are referred to as selectors, and the o_i are referred to as the components of the composite object.

If $\underline{s}$ is a selector, and t is a Vienna object of the form (s,o), the object $\underline{o}$ is the component of t corresponding to $\underline{s}$. Clearly, any component of a VO is also a VO. Given a VO, t, and a selector, $\underline{s}$, the selection operation, written $s(t)$ in VDL and in our notation

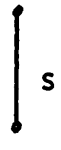
SEL(s,t), denotes the component of t corresponding to s. If there is no component corresponding to s, SEL(s,t) is the null object denoted by η.

Vienna objects can be represented graphically by labelled trees. An elementary object corresponds to a tree with no branches (hence represented by a single node). A composite object t is represented by a node having a branch for each ordered pair $(s,o)\varepsilon t$, labelled with the selector s, and the tree corresponding to the VO, o, is at the end of this branch. E.g.,

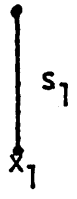


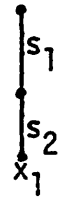
There are two other VDL operators which we will need, the constructor operator and the assignment operator. In VDL, <s:o> represents the compolite object having the single component o, i.e.,

In our notation, this object is constructed by the operator UO. UO takes two arguments, the first, a string, s; the second, a VO, o, and returns the VO



e.g.,





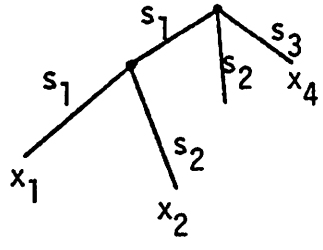
$\langle s_1:x_1\rangle$=UO(s_1,x_1)

$\langle s_1:\langle s_2:x_1\rangle\rangle$=UO($s_1$,UO($s_2$,$x_1$))

VDL also has an operator μ_0 but this is a special case of the assignment operator μ, and is therefore redundant. Note that the VDL operator μ_0 does not correspond to our UO.

A sequence of simple selectors, separated by the symbol "→", denotes a composite selector. For example, if s_1 and s_2 are simple selectors, $s_1 \rightarrow s_2$ is a composite selector, and $s_1 \rightarrow s_2(t)=s_1(s_2(t))=$ SEL(s_1,SEL(s_2,t))=SEL($s_1 \rightarrow s_2$,t) where t is any composite object. In general, $s_1 \rightarrow s_2 \rightarrow \ldots \rightarrow s_n(t)=s_1(s_2(\ldots s_n(t)\ldots)$. E.g.,

t=



$s_1 \rightarrow s_1(t)=x_1$=SEL($s_1 \rightarrow s_1$,$t$)

$s_2 \rightarrow s_1(t)=x_2$=SEL($s_2 \rightarrow s_1$,$t$)

$s_1 \rightarrow s_2(t)=n$=SEL($s_1 \rightarrow s_2$,t)

The assignment operator μ is a very powerful and useful operator which makes it possible to modify the components of a VO, to add components to a VO, and to delete components from a VO. In our notation, there are two operators corresponding to the assignment operator and the generalized assignment operator in VDL (these are both denoted by the same letter μ in VDL). The simple assignment operator, denoted by U1 in our notation, takes three arguments: a VO, a selector, and a value (also a VO). It returns a VO identi-

cal to the first argument, except the component selected by the selector, which is the second argument, equals the third argument. We will first define U1 for the case when the second argument is a simple selector, and then extend the definition to include the case when the second argument is a composite selector.

The definition for $U1(t,s,o)$ follows:

If s is simple, and

$o=\eta$ and there is a pair $(s,x)\in t$ for some x, then

$U1(t,s,o)=t-\{(s,x)\}$

$o=\eta$ and there is no pair $(s,x)\in t$ for any x, then

$U1(t,s,o)=t$

$o\neq\eta$ and there is a pair $(s,x)\in t$ for some x,

$U1(t,s,o)=(t-\{(s,x)\})\cup\{(s,o)\}$

$o\neq\eta$ and there is no pair $(s,x)\in t$ for any x,

$U1(t,s,o)=t\cup\{(s,o)\}$

If s is not simple, then $s=s_1\rightarrow\ldots\rightarrow s_n$ for a sequence of simple selectors $s_1,\ldots,s_n$. There are two cases:

If there is a pair $(s_1,x)\in t$ for some x,

$$U1(t,s,o)=(t-\{(s_1,x)\})\cup\{(s_1,\{(s_2,\ldots\{(s_{n-1},\{(s_n,o)\})\}\ldots)\})\}$$

If there is no pair $(s_1,x)\in t$ for any x,

$$U1(t,s,o)=t\cup\{(s_1,\{(s_2,\ldots\{(s_{n-1},\{(s_n,o)\})\}\ldots)\})\}$$

The generalized assignment operator U takes an arbitrary number of arguments (as implemented, up to ten), where each argument is an arbitrary Vienna object. The result is the same as if a sequence of simple assignments had been made to the first argument, assign-

ing each component of each of the arguments 2 through 10 to the first argument, i.e.,

Let $t_i=\{(s_{i_1},o_{i_1}),\ldots,(s_{i_{n_i}},o_{i_{n_i}})\}$ for i=1,...,m

then $U(t_1,t_2,\ldots,t_m)=$

$U1(\ldots U1(\ldots U1\ U1(U1(t_1,s_{2_1},o_{2_1}),s_{2_2},o_{2_2})\ldots s_{2_{n_2}},o_{2_{n_2}})\ldots s_{m_{n_m}},o_{m_{n_m}})$.

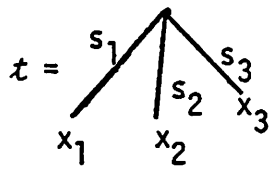
The formal definition of the assignment operator can be given in terms of characteristic sets (the characteristic set of a VO, t, is the set of all pairs {(s,o) such that s is a selector, possibly composite, and $s(t)=o$} as is done in Lucas (Lu 67). This requires development of a number of other topics which have been avoided in the above definition.

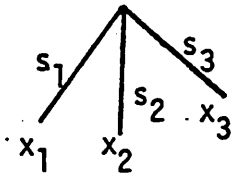
This definition looks alarmingly complex; however, it corresponds to the following simple definition in terms of trees:

U1(t,s,o) is obtained by

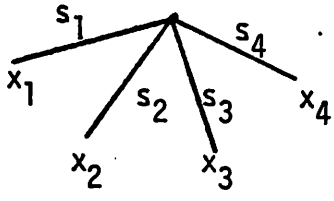
1) Assigning the value o to the s-component of t if t has an s-component;
2) Adding a new s-component o to t if t does not have an s-component;
3) If o=n delete the s-component from t if it has one.

e.g.,

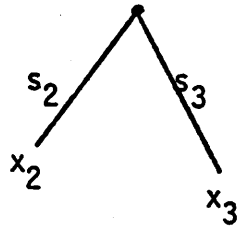




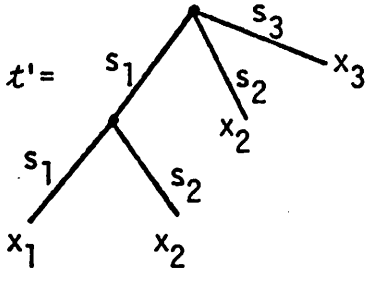
$\mu(t,\langle s_2:y\rangle)=U1(t,s_2,y)$



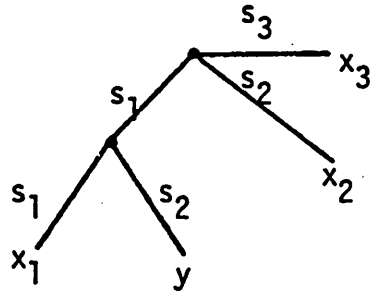
$\mu(t,\langle s_4:y\rangle)=U1(t,s_4,y)$



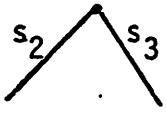
$\mu(t,\langle s_1:n\rangle)=U1(t,s_1,n)$



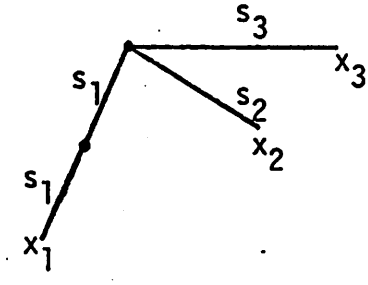
$t'=\mu(t,\langle s_1:\mu_0(\langle s_1:x_1\rangle,\langle s_2:x_2\rangle)\rangle)$
$=U1(t,s_1,U(U0(s_1,x_1),U0(s_2,x_2)))$



$\mu(t',\langle s_2\to s_1,y\rangle)$
$=U1(t',s_2\to s_1,y)$

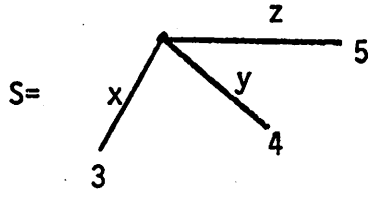


$\mu(t',\langle s_1:n\rangle)$
$=U1(t',s_1,n)$



$(t',\langle s_2\to s_1,n\rangle)$
$=U1(t',s_2\to s_1,n)$

We will represent the variables and values manipulated by a program by a state vector, and the state vector will be represented by a Vienna object. Programming language statements will be translated into functions that transform the state vector using the Vienna operators. As an indication of the direction we will pursue, suppose a program has variables x, y, and z. Then their current values will be represented by a state vector



Then

i. the selection operation corresponds to variable referencing, e.g., SEL(x,S)=3

ii. assignment in the programming sense corresponds to the first action of the assignment operator, e.g.,

"x=4" corresponds to UI(S,x,4)

"x=y" corresponds to UI(S,x,SEL(y,S))

iii. data allocation and deallocation correspond to the second and third actions of the assignment operator; thus, UI(S,x,0) allocates a variable x initialized to 0, and UI(S,x,n) deallocates x.

2.2 Knuth's Method of Specifying a Syntax-Directed Translator

In the preceding Section, it was shown how elementary programming language actions could be modelled by state vector functions.

However, it is a large step from these simple actions to any reasonable programming language, and the state vector model for a construct in such a language may be fairly complex. We will use two techniques to associate state vector functions with programming language constructs, one from the theory of syntax-directed translation, presented in this Section, and the other from recursive function theory, presented in Section 2.3.

What is needed is a general technique for specifying the translation of context-free programming languages to sets of recursive functional equations. The method we will use was originated by Knuth in his study of the semantics of context-free languages (Kn 68). A remark is in order here. The title of Knuth's paper is misleading; what is developed in this paper is a syntax-directed translation method, which Knuth incorrectly assumes answers most of the questions of programming language semantics. The work of two of Knuth's students (W. Wilner and I. Fang) will be discussed in this regard in Section 3.3.5. Chapter 5 includes a discussion of why this method of specifying the translation was chosen.

The general strategy of this translation method is to work from a parse tree, building partial translations at each node. The value of the translation at a particular node is dependent on the information (including partial translations) associated with adjacent nodes in the parse tree. The partial translations are built first at the leaves of the tree, and then at successively higher nodes. The translation associated with the root node is the

output of the translator.

The information associated with a node in the parse tree is represented by a set of attribute values (e.g., the partial translation is an attribute value). For each production in the grammar generating the language, there are equations which define the attribute values of the nonterminal (node) on the left side of the production in terms of the attribute values of nonterminals (nodes) on the right of the production. This permits semantic information to pass from the leaves of the tree upward. Attributes defined in this manner are called "synthesized attributes" and the equations defining them are called "synthesized attribute equations".

Knuth's method also makes it possible to pass semantic information down the tree; that is, some of the attribute values of a node may depend on attribute values of nodes above it in the parse tree. For attributes of this type, called "inherited attributes", there is an equation defining their value whenever the nonterminal representing the node appears on the right side of a production. These equations are called "inherited attribute equations". We will see a great number and diversity of attributes and attribute equations in the following two chapters.

Formally, let G={N,T,S,P} be an unambiguous context-free grammar for *L* [extensions to the case when G is ambiguous are considered in Fang (Fa 72)], where

N=finite set of nonterminal symbols

T=finite set of terminal symbols

$S \in N$

P=finite set of productions of the form $\alpha \rightarrow \beta$,

$\alpha \in N$, $\beta \in (N \cup T)^*$

A translation of L may be specified in the following manner: to each symbol $X \in N \cup T$ we associate a finite set of functions $A(X)$, called attributes of $\underline{X}$. Let $\alpha \in A(X)$, then the range of α is the set of possible values the attribute might have, denoted by V_α. The domain of α is the set

{(P,X)|where $\underline{P}$ is a parse tree of a program in L,

and $\underline{X}$ is an instance of a node labelled by $\underline{X}$}.

Thus, attribute values for α are defined for each occurrence of $\underline{X}$ in a parse tree. Of course, we do not have a "closed" form for the functions α, and the attribute values are actually determined by a set of attribute equations, defined next.

$A(X)$ is partitioned into two disjoint sets $A_0(X)$, called the synthesized attributes, and $A_1(X)$, called the inherited attributes. Let $\underline{P}$ consist of $\underline{m}$ productions, and let the p-th production be

$$X_{p0} \rightarrow X_{p1} \cdots X_{pN_p}$$

where $N_p \geq 0$.

Associated with this production are attribute equations of the form

$$f_{pja}: V_{\alpha_1} \times \cdots \times V_{\alpha_t} \rightarrow V_\alpha$$

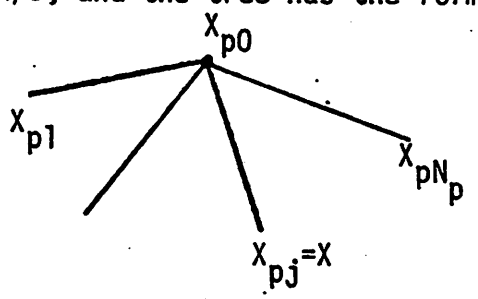
defined for j=0 and all $\alpha \in A_0(X_{p0})$, and for all $\underline{j}$ such that $1 \leq j \leq N_p$ and $\alpha \in A_1(X_{pj})$. $\underline{t}$ is some function $t(p,j,\alpha)$, and each $\alpha_i = \alpha_i(p,j,\alpha)$ is an attribute of one of the nonterminals in the production.

In words, corresponding to the above production, there is an attribute equation of the form

$$f_{pj\alpha}: V_{\alpha_1} \ldots \times V_{\alpha_t} \to V_\alpha \quad t=t(p,j,\alpha),\ \alpha_i=\alpha_i(p,j,\alpha)$$

for each synthesized attribute α of X_{p0}, and for each inherited attribute α of X_{pj}, $1 \leq j \leq N_p$.

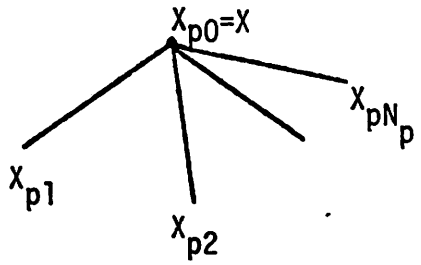
Given a parse tree for a program $\underline{P}$ in L, attribute equations can be used to define the values of the attributes at each node. Suppose the parse tree is labelled with the appropriate nonterminal and terminal symbols of $\underline{G}$. Let $\underline{X}$ be a node labelled with the symbol $\underline{X}$, and let $\alpha \in A(X)$ be an attribute of $\underline{X}$. If α is an inherited attribute, then $X \neq S$, and the tree has the form (locally)



for some $\underline{p}$. Hence, there is an equation

$$f_{pj\alpha}: V_{\alpha_1} \times \ldots \times V_{\alpha_t} \to V_\alpha \text{ defining the value of } \alpha.$$

If α is a synthesized attribute, i.e., $\alpha \in A_0(X)$, then $\underline{X}$ is a nonterminal symbol, and the tree has the local form



for some $\underline{p}$, and again there is an equation

$$f_{pj\alpha}: V_{\alpha_1} \ldots \times V_{\alpha_t} \to V_\alpha \text{ defining the value of } \alpha.$$

This equation can be used to define the value of α at $\underline{X}$ if the attributes $\alpha_1,\ldots,\alpha_t$, $t\geq 0$, at the adjacent nodes have already been evaluated.

This process of attribute definition can be repeatedly applied to all of the attributes in the parse tree for $\underline{P}$ until they are all defined. Clearly, some of the attribute functions must be constant or none of the attributes will ever get defined. Once the constant attributes are defined, then other attributes dependent on them will be defined, and so on, until all of the attributes in the tree are defined. This is true if the attribute equations are "well-defined". Knuth gives an algorithm for checking that attribute equations are well-defined in (Kn 68). Essentially, the algorithm checks for potential circularities of the form "attribute α is dependent on attribute β ...is dependent on attribute α." Clearly, in this case, α will never be defined. In practice, it is always easy to tell if such circularities exist in the semantic equations, although Knuth's algorithm is somewhat complicated.

2.3 Recursive Functional Equations and a Fixed-Point Theorem

The target language of the translation defined by the method of the preceding section will be sets of mutually dependent recursive functional equations; this "language" will be denoted by *R*. If $\underline{P}$ is a program written in language *L*, and $R=\text{Trans}_{L\text{-}R}(P)$ is the set of equations corresponding to $\underline{P}$, it will be shown that R satisfies conditions that guarantee the existance of a minimal fixed point F_R. The map from sets of recursive functional equations to

this minimal fixed point will be denoted by Sem_R. Sem_L will be defined by $Sem_L = Sem_R \circ Trans_{L-R}$. In this section, R will be defined, conditions that guarantee $R \varepsilon R$ has a minimal fixed point are given, and the fixed point theorem is proved. Also, an alternate characterization of the function defined by R is given, and this function is shown to be equal to F_R. The alternate characterization is by a computation rule; in Chapter 3, the implementation of a particular computation rule will be discussed.

The results in this section appear in references (Ma 72a) and (Vu 72). The results included here are essentially more detailed studies of work done originally by Kleene in 1952 (Kl 52).

Any functional model for general programming languages has to allow for the fact that a program may not halt for a particular input value. In this case, the function representing P should have the value "undefined" for argument x. A function that is not defined for some elements of its domain is called a partial function. To express the fact that f(x) is undefined, the special symbol ω is used, and f(x) is set equal to ω, $f(x)=\omega$. If f is defined for every element of its domain, it is a total function.

> Definition: If D is a set, D^+ denotes $D \cup \{\omega\}$. If D is a Cartesian product of sets, $D=A_1 \times A_2 \ldots \times A_N$, D^+ denotes $A_1^+ \times A_2^+ \ldots \times A_N^+$.

Clearly, a partial function $f:D_1 \rightarrow D_2$ can be considered a total function mapping D_1 into D_2^+. The following partial ordering $\leqq$ can be defined on any set D^+, corresponding to the intuitive notion "is

less defined than":

$$\omega \leqq b \text{ for every } b \in D$$

If $D^+ = A_1^+ \times A_2^+ \ldots \times A_N^+$, then

$$(\alpha_1,\ldots,\alpha_n) \leqq (\beta_1,\ldots,\beta_n) \text{ iff } \alpha_i \leqq \beta_i \quad 1 \leqq i \leqq n.$$

The condition arises where the output of one computer program (or subprogram) P_1 is the input for another program P_2. If f_{P_1} and f_{P_2} are functions representing P_1 and P_2, and P_1 loops for the input value $\underline{x}$, i.e., $f_{P_1}(x)=\omega$, then what should the value of $f_{P_2} \circ f_{P_1}(x)= f_{P_2}(\omega)$ be? That question will not be answered here, but our definitions will allow for functions defined on a domain including ω. A function $\underline{f}$ defined on $\omega \cup \{D\}=D^+$ is called an extension of $\underline{f}$ restricted to $\underline{D}$, denoted $f_{|D}$.

Definition: A function $f:D_1^+ \to D_2^+$ is monotonic iff

$$a \leqq b \Rightarrow f(a) \leqq f(b).$$

The set of monotonic functions from D_1^+ into D_2^+ is denoted by $(D_1^+ \to D_2^+)$. The partial ordering $\leqq$ on D_1^+ and D_2^+ can be used to define a partial ordering (also denoted by $\leqq$) on $(D_1^+ \to D_2^+)$. Again $\leqq$ on $(D_1^+ \to D_2^+)$ corresponds to the intuitive notion "is less defined than".

Definition: Let f_1, f_2 be elements of $(D_1^+ \to D_2^+)$, then

$$f_1 \leqq f_2 \text{ iff } f_1(a) \leqq f_2(a) \text{ for every } a \in D_1^+.$$

Theorem: Let $\{f_i\}$, $f_i \in (D_1^+ \to D_2^+)$ for even $\underline{i}$, be an increasing infinite sequence of functions (called a chain), i.e., $f_i \leqq f_{i+1}$ for $i=1,2,\ldots,\infty$. Then there is a unique limit function in $(D_1^+ \to D_2^+)$ denoted by $\lim(f_i)$, such that $f_i \leqq \lim\{f_i\}$ for every $i=1,2,\ldots,\infty$, and if $\underline{g}$ is another function such that $f_i \leqq g$ for every $i=1,2,\ldots\infty$, then $\lim\{f_i\} \leqq g$.

One way of regarding a program is as a function of functions; that is, a program combines a number of functions (such as addition, multiplication, built-in functions, functions defined by subprograms, etc.) and produces a new function, namely the function the program computes. One way of mathematically combining functions to get a new function is to use functional composition. In fact, if all programs were "straight line", functional composition could be used to describe the function computed by a program in terms of the function in each statement. When loops are present in a program, however, it is not possible to give an expression for the function computed by a program using only functional composition of the functions in each statement. One way of characterizing the function of a program with loops is to "simulate" the execution of the program. The following definitions and theorems will provide an alternate characterization.

Definition: A functional is a map $\tau:(D_1^+\to D_2^+)\to(D_1^+\to D_2^+)$. A functional is monotonic if $f\overset{\leq}{=}g \Rightarrow \tau(f)\overset{\leq}{=}\tau(g)$. A functional is continuous if for every chain $f_0\overset{\leq}{=}f_1\overset{\leq}{=}\ldots$

i) $\tau[f_0]\overset{\leq}{=}\tau[f_1]\overset{\leq}{=}\ldots$

and

ii) $\tau[\lim(f_i)]=\lim(\tau[f_i])$.

Note that the limit functions exist and are in $(D_1^+\to D_2^+)$ by the previous theorem.

The main result is the following

Theorem: Any continuous function τ: $(D_1^+ \to D_2^+) \to (D_1^+ \to D_2^+)$ has a unique minimal fixed point.

Proof: (This proof is included, since it is both easy and instructive)

Let Ω be the constant function $\Omega(x)=\omega$ for all $x \in D_1^+$.

Then $\Omega \leq \tau(\Omega)$ and hence, since τ is monotonic, $\tau^i[\Omega] \leq \tau \circ \tau^i[\Omega] = \tau^{i+1}(\Omega)$.

Hence, $\Omega \leq \tau[\Omega] \leq \tau^2[\Omega] \ldots$ is a chain, and $\lim(\tau^i[\Omega])$ exists.

Now, $\tau(\lim \tau^i[\Omega]) = \lim(\tau[\tau^i[\Omega]]) = \lim(\tau^i[\Omega])$ and therefore, $\lim(\tau^i[\Omega])$ is a fixed point of τ.

Suppose $\underline{g}$ is also a fixed point of τ. Then since $\Omega \leq g$, $\tau[\Omega] \leq \tau[g] = g$, etc., and $\tau^i[\Omega] \leq g$ for every $\underline{i}$, hence by the previous theorem $\lim(\tau^i[\Omega]) \leq g$. Therefore, $\lim(\tau^i[\Omega])$ is the unique minimal fixed point of τ.

Q.E.D.

This theorem can be generalized to sets of functional equations.

*Theorem: A continuous functional $\tau: (D_1^+ \to D_2^+)^n \to (D_1^+ \to D_2^+)^n$ given by coordinate functionals $\tau_1, \tau_2, \ldots \tau_n$ has a unique minimal fixed point

$$F_\tau = \langle F_{\tau_1}, \ldots, F_{\tau_n} \rangle$$

such that

a) $F_{\tau_i} = \tau_i[F_{\tau_1}, \ldots, F_{\tau_n}]$ for $1 \leq i \leq n$

and

b) for any fixed point $g=\langle g_1, \ldots, g_n \rangle$, i.e., $g_i = \tau_i[g_1, \ldots, g_n]$, for every $i=1, \ldots, n$, $F_{\tau_i} \leq g_i$.

Now a particular type of functional will be defined and it will be shown to be continuous.

Definition: A recursive functional equation is an expression of the form

$F=\tau[f]$

where $\tau[f]$ is defined by the composition of monotonic functions and the function variable $\underline{F}$.

A recursive functional equation defines a functional τ as follows:

for $g\varepsilon(D_1^+\rightarrow D_2^+)$,

$\tau[g](x)=\alpha(x)$ where α is formed from $\tau[F]$ by replacing each occurrence of $\underline{F}$ by $\underline{g}$.

Theorem: The functional defined by a recursive functional equation is continuous.

A set of mutually dependent recursive functional equations is a set of equations of the form $F_1=\tau_1[F_1,\ldots,F_n]$, $F_n=\tau_n[F_1,\ldots,F_n]$ where each τ_i, $i=1,\ldots,n$ is an expression defined by composition of monotonic functions and the function variables $F_1,\ldots,F_n$. Such a set determines a functional $\tau:(D_1^+\rightarrow D_2^+)^n\rightarrow(D_1^+\rightarrow D_2^+)^n$ which has a minimal fixed point by Theorem *. The minimal fixed point is denoted by $F_\tau=<F_{\tau_1},\ldots,F_{\tau_n}>$.

In view of the previous theorem, a set of recursive functional equations has a fixed point if the equations are defined using monotonic functions. Thus, when translating programs to sets of recursive functional equations, it is necessary to check that all

of the functions used are monotonic. Note that this is very easy to check, and $\underline{f}$ is monotonic if $f(\omega)=\omega$. In fact, all of the functions discussed in this thesis can easily be shown to be monotonic, and hence recursive functional equations containing them are continuous.

Suppose a program $\underline{P}$ has been translated into a set of recursive functional equations $R= F_1=\tau_1[F_1,\ldots,F_n],\ldots,\tau_n=[F_1,\ldots,F_n]$ such that each statement of $\underline{P}$ is represented by an equation in $\underline{R}$, and the first statement of $\underline{P}$ corresponds to $F_1=\alpha_1$. Then $\underline{R}$ has a minimal fixed point $F_\tau=<F_{\tau_1},\ldots,F_{\tau_n}>$ mapping $D_1^{+n}\rightarrow D_2^{+n}$. In this case, both D_1^+ and D_2^+ will be represented by state vectors, and we will define the effect of the program $\underline{P}$ on an input value $\underline{x}$ by $P(x)=F_{\tau_1}(x)$. This gives an invariant mathematical characterization of the state vector functions associated with $\underline{P}$, which is an elegant characterization of the semantics of $\underline{P}$ and is well suited to be used in mathematical proofs of properties of $\underline{P}$ [see (Ma 72)]. However, this characterization does not provide a way of calculating $F_{\tau_1}(x)$. Theorem * states that the fixed point F_τ exists but does not give an expression for F_{τ_1} or a method of calculating $F_{\tau_1}(x)$. It remains to be shown that there is an effective means for calculating F_{τ_1}.

Definition: Let $F=\tau[F]$ be a recursive functional equation, where $F_\tau:D_1^+\rightarrow D_2^+$. A computation sequence t_i, $i=0,1,\ldots,\infty$ for $a\in D_1^+$ is defined as follows:

1) The first term t_0 is $\tau[F](d)$.

2) For each i, $i \geq 0$, the term t_{i+1} is obtained from t_i in two steps:

   (a) Substitution: some occurrences of F in t_i are replaced by $\tau[F]$ simultaneously.

   (b) Simplification: known functions and predicates are replaced by their values, whenever possible, until no further simplifications can be made.

3) The sequence is finite and t_n is the final term in the sequence if no further substitutions or simplifications can be applied to t_n (that is, t_n is an element of D_2^+).

A computation rule tells us which occurrences of F to replace by $\tau[F]$ in the substitution phase (a) above. When the computation sequence was introduced by Kleene in (Kl 52), all occurrences of F were replaced simultaneously, and it was shown that the function determined by the computation sequence is identical to the minimal fixed point F_τ. This result can also be extended to sets of recursive functional equations. However, replacing all function variables simultaneously is not the only computation rule that can be used in part (a) above, and is in fact one of the less efficient rules to implement (not the least efficient, however, as some rules will determine non-terminating computation sequences where the above rule determines a terminating computation sequence).

In Chapter 3, we will discuss the implementation of a particular computation rule, different from Kleene's, and we will show that it also computes the minimal fixed point.

Definition: Let $\alpha[F^1,\ldots,F^k]$ denote any term, where superscripts are used to distinguish different occurrences of the function variable F in α. Suppose the occurrences $F^1,\ldots,F^i$ $(1\leq i\leq k)$ are chosen for substitution. This is a safe substitution if

$$\forall\, F^{i+1},\ldots,F^k\alpha[\Omega,\ldots,\Omega,F^{i+1},\ldots,F^k](d)=\omega \text{ for every } d\in D_1^+.$$

Definition: A computation rule is safe if it makes only safe substitutions.

If C is a computation rule, F_C will denote the function computed by C. The following theorem, proved in (Vu 72), states that if a computation rule satisfies a certain condition, then F_C= the minimal fixed point. Kleene's rule is an example of a rule satisfying this condition.

Theorem: Let R be a set of recursive functional equations $\{F_1=\tau_1[F_1,\ldots,F_n],\ldots,F_n=\tau_n[F_1,\ldots,F_n]\}$ and let C be a safe computation rule, then $F_C=F_{\tau_1}$.

This theorem will be discussed in greater detail in Chapter 3 where it is used to show the implementation of a particular computation rule computes the minimal fixed point.

2.4 A Correspondence Between "Flowcharts" and Recursive Functional Equations

The general strategy for the translation of a program P to a set of recursive functional equations is as follows: first, translate imperative statements in P to state vector functions which manipulate the values of variables in the same way as the statements they represent. Translate conditional statements to state vector predicates which map state vectors to the set {TRUE,FALSE,ω}. At this point, there are two equivalent directions that can be taken, as follows:

1) Each state vector function and predicate determines a recursive functional equation as follows:

   a) If c is an imperative statement, and f_c is the state vector function corresponding to c, then c determines the recursive functional equation

      $$F_c = F_d \circ f_c$$

      where d is the statement following c. Note that capital F_c, F_d, etc. are function variables.

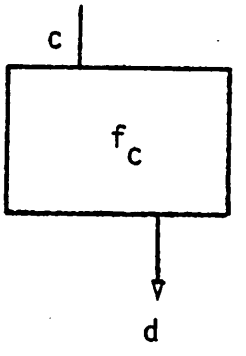
   b) If c is a conditional statement of the form IF(...)GOTO D ELSE GOTO E, and p is the predicate corresponding to c, then c determines the recursive functional

      $$F_c = \text{if } p_c \text{ then } F_d \text{ else } F_e$$

      The union of these equations is the set of mutually dependent recursive functional equations corresponding to P.

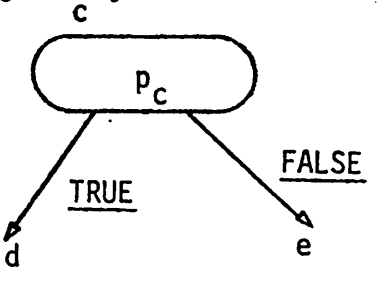
2) Each state vector function and predicate determines a node in a flowchart for P as follows:

a) If c is an imperative statement, and f_c the state vector function corresponding to c, the node corresponding to c is given by



labelled by c, where $\downarrow_d$ is an arrow to the node labelled by d, where d is the statement following c.

b) If c is a conditional statement of the form IF(...) GOTO D ELSE GOTO E, and p_c is the state vector predicate corresponding to c, the node corresponding to c is given by



The union of these nodes is a flowchart corresponding to the program P.

Thus, the recursive functional equations are in a one-to-one correspondence with the nodes of the flowchart for P.

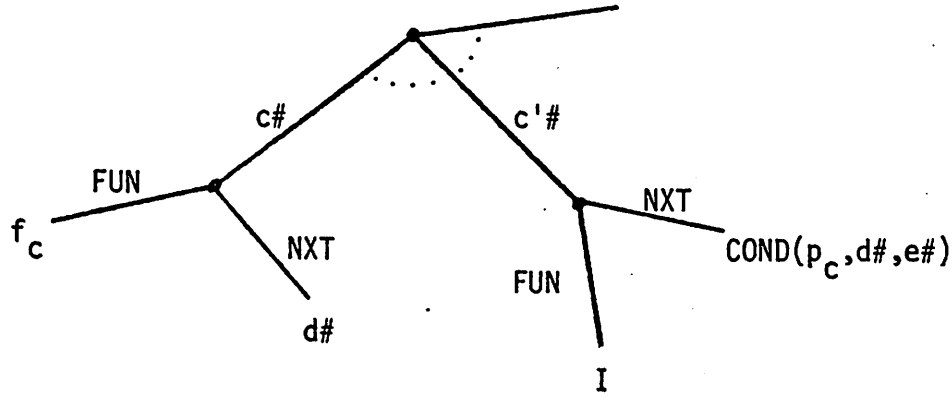
In order to further illustrate this correspondence, we will examine the structure of the functions and predicates a little more closely. Assume that $\underline{P}$ is written in a language allowing recursive procedure declarations and calls. Let P_1 be a procedure and let the statement $\underline{c}$ have a call of P_1 (recursive or not). How will f_c look if programs are translated to recursion equations, and how will f_c look if programs are translated to flowcharts?

1) Let R_{P_1} be the set of functional equations corresponding to P_1, and let Y be the minimal fixed point operator, i.e., $Y(R)$=the minimal fixed point of $\underline{R}$. Then f_c contains an application of $Y(R_{P_1})$ to the arguments in the call. Let the argument list be represented by the function f_{ARG}, i.e., given a state vector $\underline{S}$, the value of the argument list is $f_{ARG}(S)$. The only way of determining the value of $Y(R_{P_1})(f_{ARG}(S))$ is to apply a fixed point computation rule to R_{P_1}. The implementation of a particular computation rule APPLY is discussed in Section 3.3.7. Thus, $Y(R_{P_1})(f_{ARG}(S))$ can be represented in $\underline{f}$ by $APPLY(R_{P_1},f_{ARG}(S))$ and this is, in fact, what is done in the language description in Chapter 4.

2) Let F_{P_1} be the flowchart representing P_1; let the function f_{ARG} represent the arguments to P_1. Let $APPLY'(F_{P_1},f_{ARG}(S))$ be the function defined by applying the usual computation rule for flowcharts to the flowchart F_{P_1} with arguments given by $f_{ARG}(S)$. Then f_c will contain the expression

APPLY'(F_{P_1},f_{ARG}(S)) to represent the call of P_1 in c.
From Section 3.3.7 it can easily be seen that APPLY=APPLY'. Thus, it is only a matter of notation whether a program is translated to a set of recursive functional equations or to a flowchart as described above. In fact, the Vienna object representing the result of the translation can be interpreted either way.

Let c be an imperative statement in P (followed by d) and let c' be a conditional statement of the form IF(...) GOTO D ELSE GOTO E, and for any statement s let s# be a unique statement number for that statement. Then the VO representing the translation of P has the following form:



where COND(TRUE,d#,e#)=d#

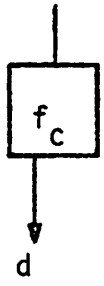
COND(FALSE,d#,e#)=e#

and COND(ω,d#,e#)=ω

The VO selected by c# corresponds to the recursive functional equation

$$F_c=F_d(f_c)$$

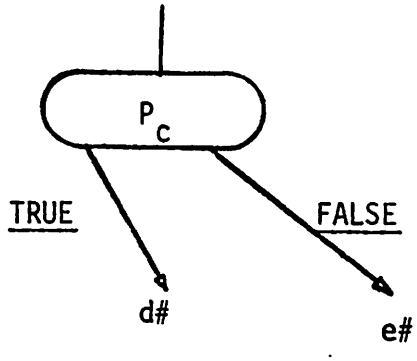
and to the flowchart node



The VO selected by c'# corresponds to the recursive functional equation

$$F_{c'} = \text{if } p_c \text{ then } F_d \text{ else } F_e$$

and to the flowchart node



In this section, assumptions have been made about the form of the statements in P to simplify the discussion. Numerous variations in the forms of statements appear in the languages defined in Chapter 4, and in some cases, slight variations have to be made in the forms of the recursion equations or flowcharts used to represent them. These will be discussed along with the language definitions themselves.

2.5 A New Method for Specifying Programming Language Semantics

A definition of a programming language *L* consists of two parts as follows:

1) The syntax of *L*. The syntax of *L* specifies which symbol

strings denote legal constructs (including programs) in *L*.

2) The semantics of *L*. The semantics of *L* provides a "meaning" for legal constructs in *L*.

It will be assumed that the syntax of *L* is given by an unambiguous context-free grammar. This deserves some comment. It is well known that some restrictions on programming language syntax cannot conveniently be specified using a context-free grammar. For example, in mini-language 6, each variable used in a program must be declared and given a type, and each expression in the program must be free of type errors. While it is impossible to distinguish between legal programs and illegal programs in mini-language 6 on the basis of syntax alone, the requirements that all identifiers be declared and all expressions be free of type errors can be incorporated into the semantic definition (see Section 4.6.2).

The "meaning" of a program P is defined as the function from input values to output values computed by P. Since the semantics of *L* provides a method of determining the meaning of every program in *L*, it can be expressed as a function

$$Sem_L : P_L \rightarrow (D_1^+ \rightarrow D_2^+)$$

where P_L is the set of syntactically legal programs in *L*, D_1 represents the set of possible input values to programs in P_L, and D_2 is the set of possible output values of programs in P_L [see Section 2.3 for an explanation of the notation $(D_1^+ \rightarrow D_2^+)$]. Since the meanings of the constructs in a particular program P are defined only in relation to the meaning of the entire program, Sem_L is de-

fined only for legal programs (i.e., P_L) and is clearly the first goal of a semantic definition.

Sem_L will be defined as follows: let G be a context-free grammar for L. Attribute equations translating programs in P_L to recursive functional equations will be associated with the productions in G (see Section 2.2). These attribute equations define a map $Trans_{L-R}$ translating legal programs in P_L to sets of recursive functional equations, i.e.,

$$Trans_{L-R}: P_L \rightarrow R$$

where $R \epsilon R$ is a set of recursive functional equations. The attribute equations also define for each construct in $P \epsilon P_L$ a set of attribute values which represent the meaning of that construct in P.

Let $Trans_{L-R}$ map P to R. The recursive functional equations in R generally correspond to "executable statements" in P on a one-to-one basis (although this is certainly not always so, e.g., mini-languages 9 and 10). The functional corresponding to a particular statement c will be one of the attribute values of the construct denoted by c. These functionals represent the manipulation of the program variables by explicit transformation of a state vector variable representing the current variables and their values (see Section 2.2). Declarative statements will likely have as attribute values tables which represent the information contained in the declaration organized in an easily accessible form.

The semantics of elements of R, i.e., set of recursive functional equations, is given by the fixed-point theorem in Section

2.3. This theorem defines

$$Sem_R: R \rightarrow (D_1^+ \rightarrow D_2^+)$$

as follows: let $R=\{F_1=\tau_1[F_1,\ldots,F_n],\ldots,F_n=\tau_n[F_1,\ldots,F_n]\}$ then

$$Sem_R(R)=F_{\tau_1}$$

where $F_\tau=\{F_{\tau_1},F_{\tau_2},\ldots,F_{\tau_n}\}$ is the minimal fixed point of R. Sem_L can now be defined by equation

$$Sem_L=Sem_R \circ Trans_{L-R}$$

This chapter contains the necessary theoretical tools for defining $Trans_{L-R}$ and Sem_R in Sections 2.2 through 2.4. Chapter 3 contains a detailed description of the format for specifying $Trans_{L-R}$. A program implementing $Trans_{L-R}$, and $Sem_R(R):D_1^+ \rightarrow D_2^+$, is listed in Appendix II and described in Chapter 3. Chapter 4 contains definitions of ten mini-languages, using this method. Chapter 5 contains advantageous comparisons of this method with other current methods.

Chapter 3. A Semantics-Based Interpreter

Abstract

An overview is given of a semantics-based interpreter and its relation to the definitional method. The format for a language definition is given in detail. The use of attribute values to represent information such as tables, flowcharts and functions is discussed, along with techniques for manipulating attribute values. The input phase, parsing algorithm, attribute definition algorithm, and computation rule of the interpreter are discussed.

3.0 Introduction

In Chapter 2, a method of semantic definition of programming languages was presented, which consists of giving for a language *L* a context-free grammar generating *L*, and a set of attribute equations (referred to as semantic rules), which when evaluated for the parse tree of a program P in *L*, effectively translate P into a set of recursive functional equations R. The function "computed" by P is defined to be the minimal fixed point of R, and this is shown to be equivalent to the function which results from applying a suitable (safe) computation rule to R.

A SNOBOL program, listed in Appendix I, has been written which does the following:

1) accepts as input a definition of a language *L*, consisting of a set of syntactic rules and associated semantic rules

(the format is given later in this Chapter), and a program P written in *L*.

2) using the syntactic rules in the definition of *L*, creates a parse tree for P.

3) using the semantic rules in the definition of *L*, evaluates all of the attributes for nodes in the parse tree of P, thereby translating P into a set of recursive functional equations R. All attribute values of the root node of the parse tree are listed.

4) applies a safe computation rule to R and lists the final state vector.

Given a definition of *L*, this program will determine the result of executing any program in *L*; hence, the program acts as an interpreter for any language for which it is provided a syntactic and semantic definition.

In this chapter, a complete account will be given of this program. First, what are its uses? The program has been used primarily for the purpose of checking out language definitions. That is, in defining a language, we have an intuitive understanding of what the language does, and we try to capture our intuition in syntactic and semantic rules. There will probably not be a standard against which we can compare the formal definition, other than the intuitive understanding we began with. One way of doing this is by "simulating" a program P in *L* according to our intuition and comparing the result with the function determined by translating

P to recursive functionals and taking the fixed point, based on the formal definition of L. If the results agree on a large enough set of test programs, we assume we have defined the language properly; if the results disagree, we must re-examine our intuition or the language definition.

3.1 Notation for Syntactic and Semantic Rules

In Chapter 4, there are ten examples of syntactic and semantic definitions of programming languages illustrating how a wide variety of programming language features can be handled. Examples from the first of these definitions, of mini-language 1, will be used here in explaining the notation used for syntactic and semantic rules. The notation used here is a variation of the notation first presented in Maurer (Mau 73); this paper contains a very complete description of the notation along with a self-description of a "language" used to define programming languages.

A language description consists of a set of rules. A rule consists of a syntactic rule, and its (possibly null) associated set of semantic rules.

Nonterminals appearing in syntax rules are enclosed in angle brackets, e.g., <STATEMENT>. Terminals are enclosed in quotes, e.g., 'END'. A syntactic rule resembles a Backus Normal Form rule in which letters used for identification follow each nonterminal. Thus, in

<PROGRAM>A=<BLOCK>B

A corresponds to <PROGRAM> and B corresponds to <BLOCK>. Alterna-

tives are separated (instead of by the usual |) by a semi-colon followed by a letter (which corresponds to the left-hand side nonterminal) followed by an equals sign (=). Thus, there are two alternate right-hand sides in the rule

<COMMAND LIST>A=<COMMAND>B";"<COMMAND LIST>C;D=<COMMAND>E

and the D corresponds to the <COMMAND LIST> nonterminal in the second production. Thus, one syntactic rule of this form incorporates all of the productions in the grammar having a particular nonterminal on the left side. The letters are necessary to identify a particular nonterminal in a particular production.

Syntactic rules always begin in column 1; the attribute rules associated with productions of the syntactic rule follow the syntactic rule and are indented 10 spaces. Each semantic rule refers to a production in the syntactic rule that precedes it. Semantic rules are divided into three classes, and the format for each class will be discussed separately. These incorporate the attribute equations as defined in Chapter 2, and also specify additional information to make the language definition easier to read.

Each attribute is given a name, which is chosen to, hopefully, convey some idea of what the attribute represents (this is optional, of course). The first type of attribute equation is simply the name, enclosed in brackets, of an inherited attribute of the nonterminal appearing on the left side of a syntactic rule. Each syntactic rule is followed by attribute equations of type 1 listing its inherited

attributes. Thus,

<BLOCK>A="BEGIN"<DECLARATION>B";"<COMMAND LIST>C";END"

<NEXT>N

<GLOBALS>G

indicates that the <BLOCK> nonterminal has inherited attributes named NEXT and GLOBALS. The letter following the name is used to identify the attribute in references to its value occurring in attribute equations.

The only time attribute names appear enclosed in brackets is indented 10 spaces, and at the front of a semantic rule. Hence, there is no confusion between attribute names and nonterminals in the grammar. In the text, nonterminals will be written with brackets, and attribute names will be written without brackets. Since inherited attributes are only defined when the nonterminal appears on the right side of a production, there are no attribute equations which define the value of the GLOBALS attribute of the <BLOCK> nonterminal associated with the above syntactic rule. Inherited attributes are defined by semantic equations of type 3.

There is a semantic rule of type 2 for every synthesized attribute of the left-hand nonterminal in the syntactic rule. A semantic rule of type 2 consists of an attribute name, enclosed in brackets, followed by equations defining the value of the attribute for each of the productions in the syntactic rule. An attribute equation defining this value will be of the form

attribute reference=attribute expression

An attribute reference is a string of the form $\alpha\uparrow\beta$ where α is a letter, and β is either a letter or the string VAL. α must appear in the syntactic rule following a nonterminal, and if β is a letter, it must be associated with one of the attributes of that nonterminal. In the case of rules of type 2, where there is an equation of the form $\alpha_1\uparrow\beta_1$=attribute expression, α_1 must identify the left side nonterminal in one of the productions, and the letter β_1 is associated with the attribute being defined. When an attribute reference is of the form $\alpha\uparrow$VAL, then the node associated with α must be a literal node, and the value of this attribute reference is the string matched by the literal (literals are discussed in Section 3.3.4). An attribute expression can involve constants, operators as discussed in Section 3.3.6, and references to other attribute values of nonterminals in the production. The following rule of type 2 also follows the syntactic equation defining <BLOCK>

<VARTABLE>A↑V=U(A↑G,B↑V)

U is one of the Vienna operators. The letter V is associated with the VARTABLE attribute, and A↑V represents this value. It is defined to be the result of applying the U operator to A↑G (the value of the GLOBALS attribute of <BLOCK>) and B↑V (the value of the VARTABLE attribute of <DECLARATION>). In this example, there is only one production in the syntactic rule; had there been more, the equations defining the VARTABLE attribute for all the productions would have been listed in the semantic rule, separated by semicolons. For example, the semantic rule for the FLOWCHART attribute

of the <COMMAND LIST> nonterminal is

<FLOWCHART>A↑F=U(B↑F,C↑F);D↑F=E↑F

In the descriptions in Chapter 4, the letter associated with an attribute is usually the first letter in the attribute name.

Semantic rules of type 3 define the values for inherited attributes of nonterminals appearing on the right side of a production. These rules consist of one or more attribute equations separated by semi-colons. The attribute reference on the left side of each equation must refer to an inherited attribute of one of the nonterminals on the right side of one of the productions in the associated syntactic rule. Clearly, there must be equations for each such attribute. The <BLOCK> nonterminal has inherited attributes NEXT and GLOBALS, thus associated with the syntactic rule

<UNLABELLED COMMAND>A

.
.
.

F=<BLOCK>G

is the semantic rule of type 3

G↑N=F↑N;G↑G=F↑V

This rule defines the values of the NEXT and GLOBALS attributes of <BLOCK> when it appears on the right side of the above syntactic rule. The value of the NEXT attribute is defined to be identical to the value of the NEXT attribute of the <UNLABELLED COMMAND> nonterminal, and the value of the GLOBALS attribute is defined to be equal to the value of the VARTABLE attribute of the <UNLABELLED COMMAND> nonterminal.

3.2 Attribute Values and Expressions

Attribute values will be represented by Vienna objects (see Section 2.1). Of course, these include elementary objects as a special case. Elementary objects that will be used in the descriptions in Chapter 4 include

1) integers

2) strings

3) functions

Elements from other domains could be used as elementary objects; for example, real numbers might be used. The elementary objects above were chosen simply because they suffice to represent all the attributes we will need in defining the ten languages in Chapter 4.

Integers and strings present no problems, and they can be represented by themselves (strings representing themselves will be enclosed in quotes). Functions will be represented by strings in the following way: if the function is elementary in the sense that it is not made up of more elementary functions using composition, it will be represented by its name. The only allowable operation on elementary functions is composition. The result of composing functions f and g could be represented by the string f ° g. However, in computer science literature it is more common to use a variation of lambda calculus notation and represent this composition by $\lambda x.f(g(x))$, or simply f(g). We will conform to this latter notation, that is, compositions will be represented by using lambda calculus notation with the following distinction: all of the functions that

will be defined will be functions of one state vector variable, the variable will always be represented by S, and the λS. prefix will be dropped from the lambda calculus notation. Thus, the composition of f and g will be represented by the string f(g(S)).

An attribute expression is defined recursively as follows:

1) A constant is an attribute expression.

2) An attribute reference is an attribute expression.

3) If $\alpha_1,\ldots,\alpha_n$ are attribute expressions, and f is a function of n variables, $f(\alpha_1,\ldots,\alpha_n)$ is an attribute expression.

When the attribute references in an attribute expression are given values, the attribute expression can be evaluated to produce a new attribute value.

The functions that can appear in forming attribute expressions using rule 3 above are:

1) the Vienna operators UO, U, U1, and SEL

2) the usual arithmetic operators

3) string concatenation

and we will assume that these functions are used only when they make sense.

In order to represent the composition of functions, we must concatenate their string representations in such a way as to produce the string representation for their composition; for example, consider

```
<FUNCTION>...; C↑F='SEL('SEL('D↑VAL', C↑V) ',S)'
```

String concatenation is represented by a blank as in SNOBOL4, thus we are concatenating the strings SEL(, whatever SEL('D↑VAL', C↑V) returns as its value, and ,S). SEL('D↑VAL', C↑V) in fact returns a number; thus the function we are creating has the form SEL(#,S), where S as always represents the state vector variable.

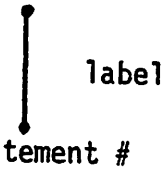
The Vienna operators can be used to form composite objects out of elementary objects, to combine composite objects, and to refer to or "select" components of composite objects. Composite objects are used to represent structured attribute values such as tables. For example, a number of the nonterminals in mini-language 1 have an attribute called XLABELS; this table is referenced with a label, represented by a string, and the value of such a reference is the statement number of the statement corresponding to the label. Entries to the table are created by the attribute equation

<XLABELS>A↑X=UO('B↑V',C↑S)...

referring to the syntactic equation

<COMMAND>A=<LABEL>A ' ' <UNLABELLED COMMAND>...

B↑V represents the VALUE attribute of the <$LABEL> nonterminal, and its value is the label itself; C↑S references the STATENO attribute of the <UNLABELLED COMMAND> nonterminal, its value is a number. Thus this equation creates a VO of the form

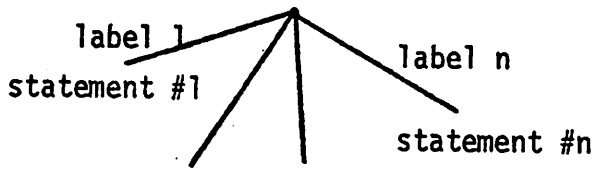


The attribute equation defining the XLABELS attribute for the

<COMMAND LIST> nonterminal is

<XLABELS>A↑X=U(B↑X,C↑X);

and the VO's created by attribute equation for <COMMAND> nonterminals are used to create a VO of the form



representing the value of the XLABELS attribute for <COMMAND LIST> nonterminals.

The set of recursive functional equations which is the ultimate objective of the entire attribute definition process is also represented by a VO. This representation is discussed in detail in Sections 2.4 and 3.3.7. This attribute will always be given the name FLOWCHART in the language definitions in Chapter 4, and the value of this attribute for each nonterminal for which it is defined is the set of functionals corresponding to that part of the program.

3.3 Overview of the Semantics-Based Interpreter

The program implementing the semantics-based interpreter is listed in Appendix II, and the results of numerous runs of this program interpreting programs in the ten mini-languages defined in Chapter 4 constitute Appendix I. The code for the program can be

broken down into seven sections as follows:

Function	Lines
1) Initialization	00100-10400
2) Initial processing of the syntactic and semantic rules	10500-19000
3) Control loop	19200-26600
4) The parsing algorithm	27300-36200
5) The algorithm for evaluating semantic attributes	36400-48300
6) Various primitive functions used in the ten mini-languages	48500-68300
7) The computation rule	68400-70700

3.3.1 Initialization

This section contains the SNOBOL function declarations and data declarations for the functions and data types used in the remainder of the program.

3.3.2 Initial Processing of the Syntactic and Semantic Rules

The primary function is to read the language definition and to store the information it contains in a tabular form that is especially adapted to the needs of the parsing algorithm and the attribute evaluation algorithm. The data is organized so that

1) Given a nonterminal of the grammar, the set of productions with that nonterminal as the left-hand side is immediately accessible as an array. For example, if <NON> is a nonterminal, then <NON>.ALT is an array containing the alternate productions of <NON>.

2) Given a production of the grammar, represented by a nonterminal and an index into the array of its alternate expansions, the set of attribute equations associated with that production is immediately accessible as an array. For example, if <NON> is a nonterminal, then <NON>.SEM.5 is an array of the attribute equations associated with the fifth alternate expansion of <NON>.
3) Given a nonterminal, a list of attributes of that nonterminal is immediately accessible as an array. For example, if <NON> is a nonterminal, <NON>.ATT is an array of the attributes of <NON>.

Once the language definition has been read in and stored in the above form, control is transferred to the control loop.

3.3.3 Control Loop

The program to be "interpreted" is read in, the parsing algorithm is called to create its parse tree, the semantic attribute evaluation is called to translate it to a set of recursive functional equations, and finally, the computation rule is applied to the recursive functional equations.

3.3.4 The Parsing Algorithm

The parsing algorithm is similar in principle to the general top-down parsing algorithm described in (Gr 71), except that the usual "scanner" is replaced by a special class of nonterminals, which match simple phrases like numbers and identifiers and which

are parsed in a different way than other nonterminals in the interest of efficiency. The nonterminals in this class are called literals, and they are identified by a dollar sign, $, as the first symbol following the left bracket, e.g., <$IDENT>, <$NUMBER>. Literals can only appear on the right side of a production, and their only attribute is the string or number they match.

The output of the parsing algorithm is a parse tree, where each node in the tree contains the following information:

1) the nonterminal represented by the node, and the production used to expand it (unless it is a literal).
2) pointers to the descendent nodes.
3) storage locations for attribute values, and a special marker, to be used by the semantic attribute evaluation algorithm.

The parsing algorithm is implemented as a recursive SNOBOL function, PARSE, which takes three arguments and returns a parse tree node. The arguments are:

1) a string of terminal symbols
2) a nonterminal symbol
3) a string of terminal and nonterminal symbols.

PARSE returns a node representing the second argument, if the sentential form consisting of the second argument followed by the third argument can be expanded to the first argument. If <PROGRAM> is the start symbol of a grammar G, then PARSE is initially called with the arguments P, <PROGRAM>, and null, to parse P.

3.3.5 The Algorithm for Evaluating Semantic Attributes

The algorithm for evaluating semantic attributes consists of scanning the parse tree, and at each node evaluating all attribute equations possible. Since attribute equations cannot be evaluated until all of the attribute references on the right of the equals sign have been assigned values, it is likely that each node of the parse tree will have to be scanned more than once before all of its attributes are evaluated. A special KEY field in each node of the parse tree is used to indicate when all of the attributes have been defined. The algorithm for evaluating the attributes is called, and when it returns, the KEY field of the root node is checked; if it is set, all attributes have been evaluated; if it is not set, the algorithm is again applied to the parse tree, which now has some attribute values filled in. This process is repeated until all attributes have been defined. The algorithm is implemented as a recursive SNOBOL sub-program, called with one argument which is a pointer to the node currently being scanned. In the following description, the parameter will be denoted by N.

SEMPASS

If KEY(N) is set, all attributes of this node have been defined, RETURN

Otherwise,

Let <NON> be the nonterminal represented by N, and let i be the production used to expand N. Then <NON>.SEM.i is the list of attribute equations for attributes defined by this production. Evaluate any of these attribute equations that

have not been evaluated previously, and which are now possible to evaluate.

Call SEMPASS for the descendants of N.

Again, evaluate any of the attribute equations which have not been evaluated previously, and which it is now possible to evaluate.

If all attributes of N are evaluated, set KEY(N).

RETURN

Like the parsing algorithm, this algorithm probably will not win many races. In (Kn 68), Knuth describes another algorithm which might be faster but this is not evident, and his method is more difficult, both to conceptualize and to program. An operating systems approach to evaluating attribute equations has been designed and implemented by Isu Fang, a student of Knuth's, in his thesis (Fa 72). In this approach, each attribute equation is considered a task, and tasks can communicate to each other via events. First, the parse tree is scanned, and when a task (attribute equation) is encountered, an attempt is made to evaluate it; if this is not possible because it depends on other attribute values that have not been evaluated, it is put on a wait list. Whenever a task is completed (i.e., an attribute equation is evaluated), all other tasks on the wait list waiting for this task are put on the ready list. New tasks are taken from the ready list until it is exhausted, which happens only when there are no more attributes that can be evaluated, and hence, the process is complete.

Fang used his method to implement a translator for translating SIMULA (a generalization and extension of ALGOL) to a variation of the Burroughs 6600 assembly language. The attribute equations defining the translation were given by W. Wilner (also a student of Knuth's) in his thesis (Wil 70). As a method of semantic definition, Wilner's and Fang's work is in the category of compiler-oriented methods discussed in Sections 1.3 and 5.1.1, and it demonstrates that the technique of attribute evaluation is sufficiently powerful to define a complete compiler of a large, complex language. The advantage of Fang's system to the above simple algorithm SEMPASS is, of course, efficiency. In Fang's system, an attribute equation is examined at most twice, when it is initially encountered, and possibly again when it is taken from the ready list to be executed.

3.3.6 Primitive Functions

The set of recursive functional equations corresponding to a program P written in language *L* is of the form $\{F_1=\tau_1[F_1,\ldots,F_n]$, $F_2=\tau_2[F_1,\ldots,F_n],\ldots,\tau_n[F_1,\ldots,F_n]\}$ where $\tau_i[F_1,\ldots,F_n]$ is an expression made up of constants, the function variable $F_1,\ldots,F_k$, the Vienna operators UO, U, U1 and SEL, and the functions representing basic functions in the language in which P is written. If *L* is an algebraic language, these primitive functions will no doubt include the usual arithmetic operators for real and integer numbers; if *L* is a string manipulation language, the primitive functions may include concatenation, a substring function, etc. This section of

the program contains the code implementing those primitive functions not already implemented in SNOBOL. The SNOBOL operators that are used as primitive functions in some of the language definitions are the arithmetic operators, and in mini-language 9, string concatenation.

Primitive functions represent operations in the language for which the semantics are given. The primitive functions for each of the mini-languages in Chapter 4 will be listed along with the definitions for the mini-languages. The primitive functions for each of the mini-languages include the Vienna operators, which are implemented in this section. Also implemented are the usual arithmetic predicates such as LTN (less than), EQU (equals), and GTN (greater than). The corresponding predicates in SNOBOL cannot be used because they do not return a value (predicates used in the mini-languages return the value TRUE or FALSE). Some of the primitive functions used in defining mini-languages 9 and 10 are more unusual, and they will be discussed along with the language definitions. This is important, because choosing reasonable and mathematically definable primitive functions is essential to the correct definition of a language. It is also desirable that the primitive functions be as simple as possible, since otherwise it is conceivable that the "hard part" of the semantics of a language could be obscured in the primitive functions used in its definition.

3.3.7 The Computation Rule

The computation rule is implemented as a SNOBOL function named APPLI (SNOBOL4 has a built-in function named APPLY), which takes two arguments. The first is a set of recursive functional equations $R=\{F_1=\tau_1[F_1,\ldots,F_n],\ldots,F_n=\tau_n[F_1,\ldots,F_n]\}$ represented by a VO as discussed in Section 2.4. The second argument is a state vector $\underline{S}$. APPLI(R,S) returns a state-vector result which is the result of applying the function defined by the set of recursive functionals to the state vector $\underline{S}$.

We will discuss the APPLI function in some detail because we want to show that the function computed using APPLI corresponds to the minimal fixed point of $\underline{R}$, that is, $\text{APPLI}(R,S)=F_{\tau_1}(S)$, using the notation F_{τ_1} to represent the minimal fixed point as in Chapter 2. Our only tool for showing this to be the case is the theorem stated in Chapter 2, Section 3. This theorem can be paraphrased as follows: a computation rule computes the minimal fixed point if, whenever the function variables $F^1,\ldots,F^k$ are substituted for in evaluating an expression $\gamma(F^1,F^2,\ldots,F^k,F^{k+1},\ldots,F^m)$, then if any of these function calls ultimately returns the value ω, the value of the entire expression γ is also ω. Formally,

$$\forall\,(F^{k+1},\ldots,F^m)\gamma[\Omega,\ldots,\Omega,F^{k+1},\ldots,F^m]=\omega$$

There are two types of recursive functional equations that can be in $\underline{R}$, as follows:

1) $F_k(x)=\alpha(x)$

2) $F_k(x)=$ if $\xi(x)$ then $\alpha(x)$ else $\beta(x)$

where α and β map state vectors to state vectors, and ξ maps state

vectors to {TRUE, FALSE,ω}. The expressions α, β, and ξ are functional compositions of basic functions and the function variables $F_1,\ldots,F_n$.

A number of fixed point computation rules are given in Manna (Ma 72) and any one of them could have been implemented in this part of the program, e.g., Kleene's rule, described in Section 2.3. The computation rule used is the leftmost-innermost rule defined in (Ma 72) as follows:

> Replace the leftmost innermost occurrence of a function variable F_i, by its expansion $\tau_i[F_1,\ldots,F_n]$.

This computation rule was chosen because using the CODE feature of SNOBOL, it can be executed relatively quickly (this becomes an issue as some of the sample programs executed on the order of one hour using this rule) and it has a very nice interpretation when R is viewed as a flowchart (this will be shown later in this section). However, there is a problem; Manna shows that this rule is in general not a fixed point rule! Fortunately, we can make one restriction that will guarantee that this rule does calculate the fixed point. We will therefore assume that

- *every basic function $f:A_1^+\times\ldots\times A_n^+$ is the natural extension of $f|_{A_1\times\ldots\times A_n}$, i.e., $f(a_1,\ldots,a_n)=\omega$ whenever any $a_1,\ldots,a_n=\omega$, and also that every expression $\tau_i[F_1,\ldots,F_n]$, i=1,...,n, is undefined whenever its argument is undefined, $\tau_i[F_1,\ldots,F_n](\omega)=\omega$ for i=1,...,n. This amounts to assuming $\tau_i[F_1,\ldots,F_n](x)=P2(x, {}_i[F_1,\ldots,F_n](x))$ where P2 is defined by

$$P2(x,y)=\begin{cases}\omega & \text{if } x=\omega\\ y & \text{otherwise}\end{cases}$$

With this assumption, it is easy to show that the leftmost-innermost computation rule calculates the minimal fixed point. The argument is as follows: for a substitution to be "unsafe", the variable substituted for must occur as part of an expression α that is an argument to a function $g(\ldots,\overset{k}{\alpha},\ldots)$ such that there are values of $a_1,\ldots,a_{k-1}$ and $a_{k+1},\ldots,a_n$ such that $g(a_1,\ldots,a_{k-1},\omega,a_{k+1},\ldots,a_n)\neq\omega$. Since this is the case, g is not the natural extension of its restriction to $A_1\times\ldots\times A_n$. All that is necessary is to show that this can never happen using the leftmost-innermost rule when the above assumption * holds. Since each primitive function is a natural extension, and $F_k(\omega)=\omega$ for every k, g must be the if-then-else function, which is not a natural extension since if TRUE then a else ω=a. However, since we are substituting for the leftmost function variable, it must be in the predicate (if there are no function variables in the predicate, it is evaluated using simplification to obtain the next term in the computation sequence, with no substitutions performed), and if-then-else$(\omega,a_1,a_2)=\omega$ regardless of a_1 and a_2. Hence, under the assumption *, the leftmost-innermost rule is safe.

The VO representing the first argument to APPLI has two interpretations, first as a flowchart, and second, as a set of recursive functional equations (see Section 2.4). As an illustration of this correspondence, consider the following set of recursive functional

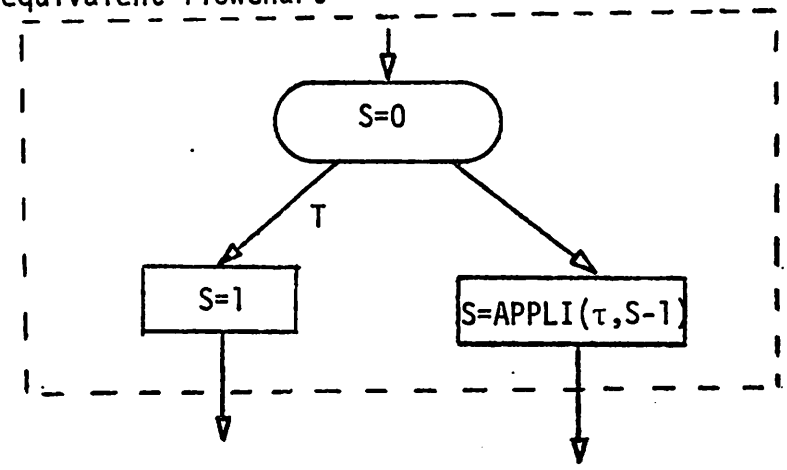
equations

$$F_1(S)=\text{if } S=0 \text{ then } F_2(S) \text{ else } F_3(S)$$

$$F_2(S)=1$$

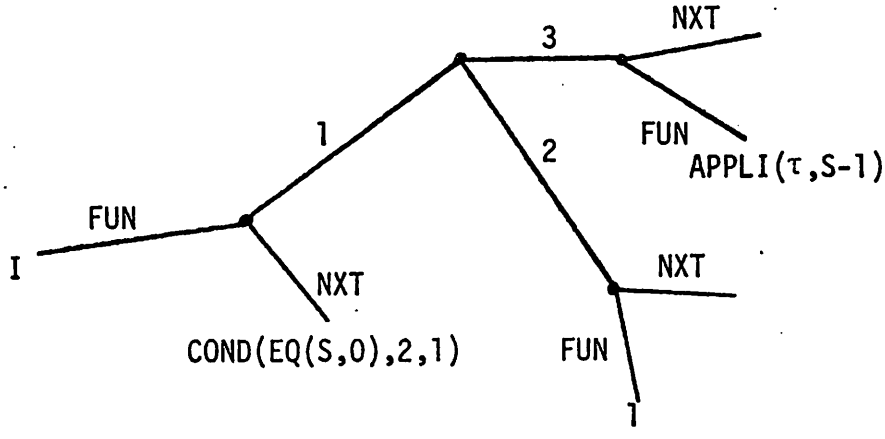
$$F_3(S)=F_1(S-1)$$

and the equivalent flowchart



where τ is a reference to the flowchart itself.

The actual data object which the program would store to represent both the set of recursive functionals and the flowchart would be



The actual code for APPLI can best be understood keeping the flowchart interpretation in mind. It corresponds to executing the flowchart the way one normally would, that is, at node x of the flow-

chart, setting the new state vector S'=FUN(x)(S) where S is the current state vector, then executing the node given by x'=NXT(x)(S'). If x' is null, the computation is complete, and the result is the current state vector.

Chapter 4. Definitions of Ten Mini-Languages

Abstract

Ten programming languages especially designed to isolate and illustrate the notions of assignment, locations, transfer of control, functions, parameter passing, static type checking, dynamic type checking, data structures, string manipulation, and input/output, are given formal definitions using the method described in Chapters 2 and 3.

4.0 Introduction

In this chapter, complete syntactic and semantic definitions are given for ten model programming languages designed to isolate and illustrate a number of the complex features found in current programming languages. The ten languages are taken from the paper "Ten Mini-Languages: A Study of Topical Issues in Programming Languages" by Henry F. Ledgard (Le 71). The ten mini-languages are defined informally by Ledgard, and for each language, Ledgard's informal definition will be given verbatim. Following Ledgard's description will be:

1) a list of the notational changes made in the language in order to satisfy the restricted character set of most (in particular, our) line printers.
2) a general discussion of the strategy for representing the

features of the language by recursive functional equations.

3) a detailed discussion of the more important and/or obscure semantic equations, and the formal definition of the mini-language.

In Ledgard's paper, included with each mini-language are a set of programs written in the language and a discussion of the action and/or result when the programs are executed. These will be included with Ledgard's description of the language, as they are essential in aiding the reader to understand the mini-language. Each of these programs was also "run" on the semantics-based interpreter, described in Chapter 3, and the results of these runs constitute Appendix I. Each of the ten sections in Appendix I consists of:

1) the formal definition of the mini-language
2) for each sample program:
    a) a listing of the program
    b) the values of the attributes for the root node of the parse tree of the sample program
    c) the final state vector after "execution" of the program
    d) timing information for the various phases of the interpreter

The features of the ten mini-languages are listed in the following table:

SUMMARY OF TOPICS TREATED IN EACH MINI-LANGUAGE

Mini-language		Topic
1	Simple assignment, transfer of control, and block structure	
2	Generalized assignment and the notion of locations	Commands
3	Generalized transfer of control	
4	Functions	
5	Passing of parameters	Procedures
6	Static type checking	
7	Dynamic type checking	Type checking
8	Structured data	
9	String manipulation	Applications
10	Input/output	

In the case of each mini-language, we are faced with the problem of defining formally what Ledgard defines informally. The syntax is, for practical purposes, given a complete definition in Ledgard's paper. Although the rules are stated in English and scattered through the description, they can be gathered together and formalized to constitute a context-free grammar for the mini-language. Thus, the syntax of a mini-language never poses a problem.

There is, of course, no correspondingly simple way to formalize the semantics of the mini-languages. Ledgard presents the semantics informally in the manner of the ALGOL report [(Na 60) or see Section 1.2]. That is, after the syntax of a construct is given, there follow a number of statements (i.e., a general discussion) giving the semantics of the construct, that is, how it affects the values of variables when it is encountered in executing the program. Of course, these statements cannot be translated on a one-to-one basis to semantic equations; instead, they give an intuitive understanding of the language which is then formulated in the semantic equations. Often these sentences will have specific counterparts among the semantic equations.

The definitions of the ten mini-languages are in Sections 1 through 10 of this chapter; Section 11 contains a discussion of the "correctness" of these definition.

4.1 Mini-Language 1: Simple Assignment, Transfer of Control, and Block Structure

4.1.1 Ledgard's Description (Verbatim)

Description of Language Elements

Commands in mini-language 1 are of three varieties:

1) a simple assignment command, which consists of a string of the form i:=e, where i is an identifier, and e is a numeral or an identifier;

2) a hopto command, which consists of a string of the form hopto ℓ, where ℓ is a label; and

3) a block (defined below)

A command may be prefixed by a label.

A declaration consists of a string of the form iden ℓ, where ℓ is a list of identifiers, each of which must be different.

A block consists of a string of the form begin d c end, where d is a declaration, and c is a sequence of commands such that:

1) each label prefixing a command in c must be different; and

2) each label in a hopto command in c must be identical to some label prefixing a command in c.

An identifier given in a declaration is said to have a "scope", which consists of the block in which the identifier is declared and all encompassed blocks that do not contain another declaration of the same identifier. (Note that a label prefixing a command in c may be said to have a "scope" similar to that of an identifier. Because of requirement (2) above, it is not necessary in mini-language 1 to extend the concept of the scope of a label to more deeply nested blocks. This issue will be treated in mini-language 3.)

Program Execution

Each identifier declared in a program refers to an (initially unspecified) object. The execution of a program begins with the execution of the outermost block and proceeds according to the following rules:

1) Each command in a block is sequentially executed until a hopto command (defined below) is encountered.

2) The execution of a simple assignment command of the form i:=e

proceeds as follows:

a) the expression e is evaluated, and if e is an identifier, its value is the object currently assigned to the identical, declared identifier whose scope includes the assignment command;

b) the object obtained in (a) is assigned as the value of the identifier i whose scope includes the assignment command.

3) The execution of a hopto command of the form hopto ℓ causes execution to continue (sequentially) from the command that is prefixed by the label and that occurs in the same block (not a more deeply nested block) as the hopto command.

The execution of a program terminates when the execution of the outermost block is terminated (unless, as mentioned earlier, a violation condition arises from an attempt to evaluate an identifier whose value is unspecified).

EXAMPLE 1:

```
begin iden A,B
  A:=1
  B:=2
  begin iden B,C
    B:=A
    C:=7
  end
  A:=3
  B:=3
end
```

EXAMPLE 2:

```
begin iden A,B
  A:=1
  B:=2
  begin iden B,C
    B:=A
    C:=7
  end
  hopto L1
  A:=3
L1 B:=3
end
```

EXAMPLE 3
(Syntactically illegal):

```
begin iden A,B
  A:=1
  B:=2
  begin iden B,C
    B:=A
    D:=7
  end
L1 A:=3
L1 C:=3
end
```

In Examples 1 and 2, there are four declared identifiers: A and B in the outer block, and B and C in the inner block. Let us refer to these four identifiers as A_1, B_1, B_2, and C_2. In Example 1 the final values of A_1, B_1, B_2, and C_2 will be 3, 3, 1, and 7; whereas in Example 2 the final values will be 1, 3, 1, and 7. The program of Example 3 is syntactically quite illegal, notably in that identifier D is not declared, the label L1 prefixes two commands in a single block, and identifier C in the last command is not declared for the block in which it occurs.

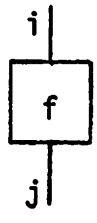
4.1.2 Discussion

All three of the features - assignment, transfer of control, and block structure - are very interesting from the definitional point of view; in spite of their commonness. In fact, the first two concepts, assignment and transfer of control, probably present the greatest problems to designers of definitional methods. The reason is simple: the notions of assignment and transfer of control do not exist in mathematics. This is the reason that most definitional methods, rather than giving a mathematical model for semantics, finally rely on a real or abstract computer. That is, a computer "contains" the notions of assignment and transfer of control; they are "wired in", so to speak; so if a computer is used in the definitional model, it is not necessary to give a mathematical treatment of these concepts. The difficulty in handling assignment and transfer of control is evidenced by the number of papers written which present definitional methods which include only languages lacking one or both of these two fundamental operations, e.g., McCarthy (McC 64), Scott and Strachey (Sc 70b), Wegner (We 72), Reynolds (Re 73)*.

---

*The reason GOTO's, in particular, wreak such havoc with definitional methods [see, for example, (Re 73)], is that, when a programming language allows GOTO's, it is difficult to assign a semantic interpretation to sub-units of the language corresponding to phrases in the language representing nonterminals in the grammar. This is because the semantics of any phrase containing a GOTO might depend on the entire program. When GOTO's are not present, however, it is possible to "localize" the semantic definition. The underlying reason for the difficulty is that a program is taken to be a linear representation of its parse tree, but if the program contains GOTO's,

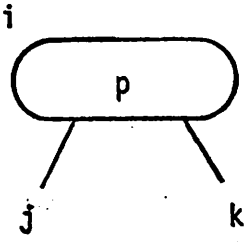
The methods of handling assignment and transfer of control are most easily seen in the absence of block structure; so, assume that there is only one block in every program in mini-language 1. Then, corresponding to every numeral e, we have the constant state vector function $f_e(S)=e$; for every identifier e we have the state vector function $f_e=SEL(e,S)$; and for each assignment statement "i=e" we can generate the state vector function $f_{i=e}=U1(S,i,f_e)$. To create a flowchart node, the only thing more we need to know is what node to transfer to. In the case of a HOPTO command, this is also all we need to know to generate an appropriate flowchart node. The method of determining which node to execute next will be explained in the next section; let us now assume that we do know the next node to execute. Then the flowchart nodes can be created for each of these types of statements, and the union of these nodes will constitute a "flowchart" for the whole program. This, in fact, is precisely what the semantic equations do. As was discussed in Chapter 2, Section 4, such a flowchart corresponds to a set of recursive functional equations, one for each node. Recalling this discussion, if the flowchart node is of the form



---

the parse tree is entirely inadequate to represent the control structures of the program. The method of this thesis avoids this problem by first translating a program to a graph-structured object, and defining the semantics for the translated object.

the recursive functional is $F_i=F_j(f)$; and if the flowchart node is of the form



the recursive functional is F_i=if p then F_j else F_k.

The technique of associating state vector functions mapping state vectors to state vectors with imperative statements, and state vector predicates mapping state vectors to {TRUE, FALSE, ω} with conditional transfer statements (here the constant predicate TRUE corresponds to the HOPTO statement) is used in all of the language descriptions that follow. Additional features like block structure, in the case of mini-language 1, complicate the way in which the state vector functions and predicates are arrived at, but can be seen as embellishments of the underlying technique outlined above.

Block structure is handled as follows: consider the program

```
BEGIN
IDEN A,B
A=1
  BEGIN
  IDEN A
  A=2
  END
* B=A
END
```

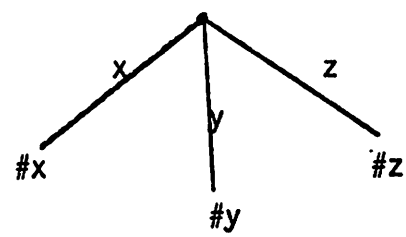
At the point in the program marked *, the value assigned to B must be 1. If we translated the statements (as suggested above)

$F_1=F_2 \circ f_{A=1}$, $\quad f_{A=1}=U(S,UO(A,1))$

$F_2=F_3 \circ f_{A=2}$, $\quad f_{A=2}=U(S,UO(A,2))$

$F_3=f_{B=A}$, $\quad f_{B=A}=U(S,UO(B,SEL(A,S))$

then B would be assigned the value 2, which, of course, is not what we want. The very easy solution is to use the semantic equations to associate with each declared identifier a unique name, and to pass the identifier/unique name to every statement in the scope of the declaration. Then, instead of using the name itself in constructing the functions f_e and $f_{e=\alpha}$, the unique name associated with e is used. Name-unique name associations constitute the VARTABLE attribute, which has the form



where x, y, and z are identifiers and #x, #y, and #z represent unique names. Note that no recursion equations are introduced by block structure to handle block entry and exit.

4.1.3 Formal Definition

The primitive functions used in the formal definition of mini-language 1 are

1) the Vienna operators UO, U, U1, and SEL.

Note: Mini-language 1, like a number of other mini-languages, lacks the usually expected arithmetic operators (+,-,*,/) and comparative operators (<,>,=,etc.). Hence, these functions are not needed as primitive functions in the definition of the language.

Following the formal definition, the most significant attribute equations will be discussed.

MINI-LANGUAGE 1

SYNTACTIC AND SEMANTIC DESCRIPTION

```
<PROGRAM>A=<BLOCK>B
         <FLOWCHART>A↑F=B↑F
         <SYMBOLTABLE>A↑S=B↑V
         B↑N='NUL';B↑G='NUL'

<BLOCK>A='BEGIN ' <DECLARATION>B ';' <COMMAND LIST>C ';END'
         <NEXT>N
         <GLOBALS>G
         <VARTABLE>A↑V=U(A↑G,B↑V)
         <FLOWCHART>A↑F=C↑F
         <STATENO>A↑S=C↑S
         C↑N=A↑N;C↑V=A↑V;C↑L=C↑X

<DECLARATION>A='IDEN ' <VARLIST>B
         <VARTABLE>A↑V=B↑V

<VARLIST>A=<$LETTR>B ',' <VARLIST>C;D=<$LETTR>E
         <VARTABLE>A V=U(UO('B VAL',UNO()),C V);D V=UO('E VAL',UNO())

<COMMAND LIST>A=<COMMAND>B ';' <COMMAND LIST>C;D=<COMMAND>E
         <NEXT>N
         <VARTABLE>V
         <LABELTABLE>L
         <XLABELS>A↑X=U(B↑X,C↑X);D↑X=E↑X
         <FLOWCHART>A↑F=U(B↑F,C↑F);D↑F=E↑F
         <STATENO>A↑S=B↑S;D↑S=E↑S
         B↑V=A↑V;B↑N=C↑S;B↑L=A↑L
         C↑V=A↑V;C↑N=A↑N;C↑L=A↑L
         E↑V=D↑V;E↑N=D↑N;E↑L=D↑L

<COMMAND>A=<LABEL>B ' ' <UNLABELLED COMMAND>C;D=<UNLABELLED COMMAND>E
         <NEXT>N
         <VARTABLE>V
         <LABELTABLE>L
         <XLABELS>A↑X=UO('B↑V',C↑S);D↑X='NUL'
         <FLOWCHART>A↑F=C↑F;D↑F=E↑F
         <STATENO>A↑S=C↑S;D↑S=E↑S
         C↑N=A↑N;C↑V=A↑V;C↑L=A↑L
         E↑N=D↑N;E↑V=D↑V;E↑L=D↑L

<LABEL>A='L' <$NUMBR>B
         <VALUE>A↑V='L' B↑VAL

<UNLABELLED COMMAND>A=<$LETTR>B '=' <VALUE>C;
.D='HOPTO ' <LABEL>E;
.F=<BLOCK>G
         <NEXT>N
         <VARTABLE>V
         <LABELTABLE>L
         <STATENO>A↑S=UNO();D S=UNO();F↑S=G↑S
         <FLOWCHART>A↑F=UO(A↑S,U(UO('FUN','U(S,UO(' SEL('B↑VAL',A↑V) ',C↑F))'),
          .UO('NXT',A↑N)));
          .D F=JO(D↑S,U(UO('FUN','S'),UO('NXT',SEL('E↑V' ,D↑L))));
          .F F=G↑F
         C↑V=A↑V
```

```
          G↑N=F↑N;G↑G=F↑V

<VALUE>A=<$NUMBR>B;C=<$LETTR>D
          <VARTABLE>V
          <FUNCTION>A↑F=B↑VAL;C↑F='SEL(' SEL('D↑VAL',C↑V) ',S)'
END
```

NEXT - the value of this attribute for a <COMMAND> nonterminal is the statement number of the next command in the command list.

- - - - - - - -

VARTABLE - this is a table of identifier-unique name pairs. Entries to VARTABLE are created by the rule

```
<VARLIST>A=<$LETTR>B ',' <VARLIST>C;D=<$LETTR>E
        <VARTABLE>A↑V=U(UO('B↑VAL',UNO()),C↑V);D↑V=UO('E↑VAL',UNO())
```

UNO( ) is a function which always returns a unique name. The VARTABLE of a <BLOCK> nonterminal is given by

```
<BLOCK>A='BEGIN ' <DECLARATION>B ';' <COMMAND LIST>C ';END'
:
        <VARTABLE>A↑V=U(A↑G,B↑V)
```

This value for VARTABLE is passed to each command in the block.

- - - - - - - -

GLOBALS - when a <BLOCK> is parsed as an <UNLABELLED COMMAND>, its set of global variables is the VARTABLE of the command. Thus,

```
<UNLABELLED COMMAND>A=<$LETTR>B '=' <VALUE>C;
.D='HOPTO ' <LABEL>E;
.F=<BLOCK>G
:
:
:
:
:
:
:
          G↑N=F↑N;G↑G=F↑V
```

- - - - - - - -

LABELTABLE - is a table of label-statement number pairs local to a block. This table is formed from the XLABELS attribute and

passed to each command in the command list of a <BLOCK>, thus

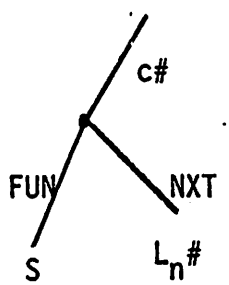
```
<BLOCK>A='BEGIN ' <DECLARATION>B ';' <COMMAND LIST>C ';END'
:
          C↑N=A↑N;C↑V=A↑V;C↑L=A↑L
```

- - - - - - - - -

FLOWCHART - the flowchart nodes are created by the equation

```
<UNLABELLED COMMAND>A=<$LETTR>B '=' <VALUE>C;
.D='HOPTO ' <LABEL>E;
.F=<BLOCK>G
:
        <FLOWCHART>A↑F=UO(A↑S,U(UO('FUN,'U(S,UO('SEL('B↑VAL',A↑V) ',C↑F))'),
        .UO('NXT',A↑N)));
        .D↑F=UO(D↑S,U(UO('FUN','S'),UO('NXT',SEL('E↑V' ,D↑L))));
        .F↑F=G↑F
```

If the statement c has the form i=e, then the corresponding flowchart node is



where L_n# is the statement number of the statement labelled by L_n.

- - - - - - - - -

FUNCTION - if a value v is a number n, the state vector function associated with v is the constant function η. If v is an identifier i, the function associated with v is

SEL(α,S)

where α is the unique name associated with i.

```
<VALUE>A=<$NUMBR>B;C=<$LETTR>D
        <VARTABLE>V
        <FUNCTION>A↑F=B↑VAL;C↑F='SEL(' SEL('D↑VAL',C↑V) ',S)'
```

4.2 Mini-Language 2: Generalized Assignment and the Notion of Locations

4.2.1 Ledgard's Description (Verbatim)

Description of Language Elements

The primitive objects in mini-language 2 are the literals 'A', 'B',...,'Z'.

A left-hand expression is either an identifier or a left-hand expression prefixed by the symbol "↓".

A right-hand expression is either a literal, an identifier, or a right-hand expression prefixed by the symbol "↓".

An assignment command is a string of the form x:=y, where x is a left-hand expression and y a right-hand expression. An assignment command is executed by performing the steps described below as follows: steps (1) and (2), in either order, then step (3).

1) "Right-mode" evaluation of y: a) if y is a literal, its value is the literal itself; b) if y is an identifier i, its value is the literal currently stored in the location for i; and c) if y is a right-hand expression e prefixed by ↓, its value is the literal "ℓ" currently stored in the location of the identifier i, where the literal "i" is the right-mode value of e.
2) "Left-mode" evaluation of x: a) if x is an identifier i, its value is the location in which the object associated with i is stored; and b) if x is a left-hand expression e prefixed by ↓, its value is the location of the identifier i, where "i" is the right-mode value of e.

3) The literal obtained in (1) is copied into the location obtained in (2).

Program Execution

A program consists of a sequence of assignment commands, each to be executed sequentially.

EXAMPLE 1:

```
X:='A'
Y:='X'
Z:=Y
↓Z:='B'
```

EXAMPLE 2 (in violation):

```
X:='A'
Y:=↓X
```

In Example 1, there are three identifiers: X, Y, and Z. At the end of the execution of the program the literals associated with X, Y, and Z will be 'B', 'X', and 'X'. Example 2 is in violation upon execution of the second assignment command because the evaluation of ↓X results in an attempt to obtain a literal from the location of A, which does not have a specified value.

4.2.1.a Notational Changes

Literals are denoted (A), (B), (C),...,(Z), rather than 'A', 'B', 'C',...,'Z' (single quotes are symbols in the meta language for the syntax descriptions).

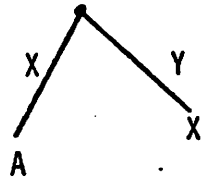
The non-printing symbol ↓ is replaced by the printing symbol ?.

4.2.2 Discussion

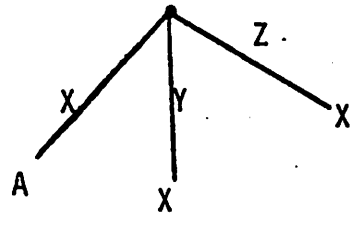
Mini-language 2 has only one type of statement, the assignment statement, and the basic strategy for representing programs in mini-language 2 will be the same as for programs in mini-language 1.

Since mini-language 2 does not have block structure, no name conflicts can arise; hence it is not necessary to associate unique names with the identifiers in mini-language 2 programs.

In mini-language 2, a literal can be assigned as a value, and later used to denote a "location" in memory. This presents no problems for a Vienna Definition Language representation, since string selectors play a similar role with respect to state vectors. For example, in the first program above, when "X" is assigned to Y, the state vector has the form



The next statement transforms the state vector to



Then the reference ↓Z treats the value of Z, namely X, as a location, and the statement

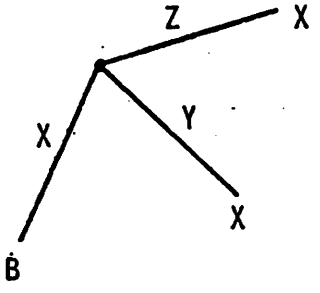
```
↓Z='B'
```

modifies the value stored at that location. In terms of state vectors and selectors, the component corresponding to the selector X is changed. In short, "locations in memory" can be represented by selectors and manipulated by the usual VDL operators. The above

statement has the associated state vector function

U1(S,SEL('Z',S),'B').

The final state vector is



4.2.3 Formal Definition

The primitive functions used in the formal definition of mini-language 2 are

1) the Vienna operators U0, U, U1, and SEL.

Following the formal definition, the most significant attribute equations will be discussed.

MINI-LANGUAGE 2

SYNTACTIC AND SEMANTIC DESCRIPTION

```
<PROGRAM>A=<ASSIGNMENT LIST>B ';END'
         <FLOWCHART>A↑F=B↑F
         B↑N='NUL'

<ASSIGNMENT LIST>A=<ASSIGNMENT COMMAND>B ';' <ASSIGNMENT LIST>C;
.D=<ASSIGNMENT COMMAND>E
         <NEXT>N
         <FLOWCHART>A↑F=U(B↑F,C↑F);D↑F=E↑F
         <LINENO>A↑L=B↑L;D↑L=E↑L
         B↑N=C↑L
         C↑N=A↑N
         E↑N=D↑N

<ASSIGNMENT COMMAND>A=<LEFT HAND EXPRESSION>B '=' <RIGHT HAND EXPRESSION>C
         <NEXT>N
         <LINENO>A↑L=UNO()
         <FLOWCHART>A↑F=UO(A↑L,U(UO('FUN','U1(S,B↑L,C↑R)'),UO('NXT',A↑N)))

<LEFT HAND EXPRESSION>A=<$LETTR>B;
.C='?' <LEFT HAND EXPRESSION>D
         <LHSFUN>A↑L='"B↑VAL"';
         .C↑L='SEL(D↑L,S)'

<RIGHT HAND EXPRESSION>A=<LITERAL>B;
.C=<$LETTR>D;
.E='?' <RIGHT HAND EXPRESSION>F
         <RHSFUN>A↑R=B↑V;
         .C↑R='SEL("D↑VAL",S)';
         .E↑R='SEL(F↑R,S)'

<LITERAL>A='(' <$LETTR>B ')'
         <VALUE>A↑V='"B↑VAL"'
END
```

LHSFUN - is the function which corresponds to evaluating an expression on the left of an assignment command.

RHSFUN - is the function which corresponds to evaluating an expression on the right of an assignment command.

- - - - - - - -

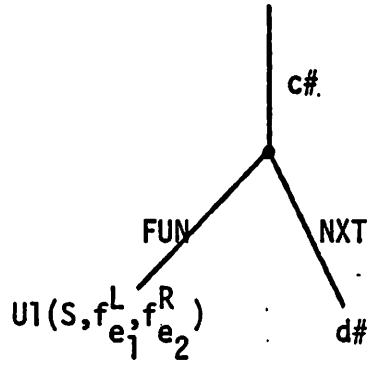
FLOWCHART - flowchart nodes are created by the rule

<ASSIGNMENT COMMAND>A=<LEFT HAND EXPRESSION>B '=' <RIGHT HAND EXPRESSION>C

⋮

<FLOWCHART>A↑F=UO(A↑L,U(UO('FUN','U1(S,B↑L,C↑R)'),UO('NXT',A↑N)))

The node corresponding to the command $e_1=e_2$ is



where d# is the statement number of the command following c, $f^L_{e_1}$ is the LHSFUN for e_1, and $f^R_{e_2}$ is the RHSFUN for e_2.

4.3 Mini-Language 3: Generalized Transfer of Control

4.3.1 Ledgard's Description (Verbatim)

Description of Language Elements

Commands in mini-language 3 are of three varieties:

1) an assignment command, which consists of a string of either the form i:=e or i:=ℓ, where i is an identifier, e is a numeral or

an identifier, and ℓ is a label;

2) a goto command, which consists of a string of the form goto x, where x is an identifier or a label; and

3) a block (defined below).

A command may be prefixed by a label.

A declaration consists of a string of the form iden $i_1,i_2,\ldots,i_n$, where $i_1, i_2,\ldots,i_n$ are identifiers, each of which must be different.

A block consists of a string of the form (1) begin d c end, where d is a declaration and c is a sequence of commands such that each label prefixing a command in c must be different. An identifier given a declaration or a label prefixing a command is said to be "declared", and to have a "scope" consisting of block (1) above, in which the identifier or label is declared, and all encompassed blocks in c that do not contain another declaration of the same identifier or label.

A program consists of a block such that each occurrence of an identifier or label in a command lies within the scope of an identical, declared identifier or label identifier.

The value of a label ℓ in an assignment or goto command consists of two parts: 1) the (unique) command c that is prefixed by ℓ and lies within the scope of ℓ; and 2) the "environment" for c; i.e., a pairing of the identifiers whose scope includes c with the locations in which their current values are stored.

Program Execution

Each identifier in a program refers to an (initially unspecified) object. The execution of a program begins with the execution

of the outermost block and proceeds according to the following rules:

1) The execution of a block results in the allocation of new locations for identifiers that are local to the block, followed by the sequential execution of each command in the block until a goto command (defined below) is encountered.
2) The execution of an assignment command is the same as for mini-language 1, except that for a label assignment command, such as i:=ℓ, the value of the label ℓ is assigned to i.
3) The execution of a goto command of the form goto x is as follows: a) the value of x is obtained, and if it is not that of a label, the command is in violation; and b) execution continues (sequentially) at the command c given with the value of x, and the environment for execution is set to that given with the value of x.

Note that the execution of a goto command may result in the transfer of execution to a block not encompassing the goto command (see Example 2).

EXAMPLE 1:

```
begin iden A,B
      A:=L1
      B:=2
      begin iden B,C
            B:=3
            C:=4
            goto A
      end
      A:=6
L1    A:=1
end
```

EXAMPLE 2:

```
begin iden A,B
      B:=2
      begin iden B,C
            B:=3
            A:=L3
            goto L1
      L3    C:=4
            goto L2
      end
      A:=6
L1    goto A
L2    A:=1
end
```

In both examples 1 and 2 there are four names: A and B in the outer block, and B and C in the inner block. Let us refer to the four names as A_1, B_1, B_2, and C_2. In both examples, the final values of A_1, B_1, B_2, and C_2 will be 1, 2, 3, and 4. Note that in Example 2 the command "goto A" transfers execution from the outer block into the inner block.

4.3.1.a Restrictions

All labels in mini-language 3 are required to be unique. In the above definition, labels have scope rules similar to variables; these rules could be incorporated into the definition by associating a unique name with each label, but this introduces no new ideas and obscures the definition of the language in additional detail. Hence, it is not included.

4.3.2 Discussion

While this language illustrates the use of labels as values, it is not nearly as general as it might be; in particular, there

are no functions, and as a result, no label-valued arguments to functions.* Nevertheless, Ledgard thinks that this language poses serious problems for the following reason: in languages with block structure, a variable may have a number of different values, depending on its location in the program. If unrestricted GOTO's are al-

---

*Our strategy is to translate programs and procedures to sets of recursive functional equations and then apply the fixed point operator to obtain functions representing the programs and procedures (see, for example, the definitions of mini-languages 4, 5, 6 and 7). Let p be a procedure, and f_p the function associated with p, and let c be a statement of the form "CALL p". This statement will be represented by a functional of the form

$$F_{c\#}=F_{d\#}\ \ f_p\ \ (f_{ARG})$$

where $f_{ARG}(S)$ is the list of parameter values passed to p, and d is the statement following c. However, if it is possible to pass a label-valued parameter to p, then it is also possible that execution will never resume at the statement d. In this case, the above functional equation does not accurately represent the action of the statement c. The only "solution" to this problem known by the author involves using what is commonly known as van Wijngaarden's device to translate the original program to a program free of labels altogether [this technique is presented by van Wijngaarden (vanW 66)].

Suppose mini-language 3 did have functions, and one of the parameters to p is label-valued. Suppose the label passed is ℓ_k and the functional corresponding to the statement labelled by ℓ_k is

$$F_k=\alpha_k$$

Then, the value that will be passed to p will be a function represented by a name (in this case, F_k) and a VO representing the current bindings of the variables (see Section 4.4.2). Also, as per van Wijngaarden's device, an extra parameter, the "continuation", will be passed to p. The continuation will consist of the functional name F_d, and the VO representing the current bindings of the variables. The functional corresponding to c would be

$$F_c=f_p\ (f_{ARG}\ ,U(UO(NAME,F_d),UO(ENV,SEL(ENV,S))).$$

lowed, there is the problem of making sure that a variable is associated with its correct value after the GOTO.

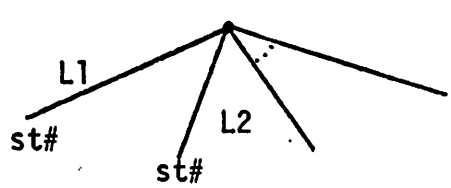
From the semantic perspective of this thesis, the above problem does not exist; that is, we regard a block structured language as possibly having a number of different variables represented by the same identifier. The variable-identifier correspondence is determined strictly by the scope rules of the language (scope rules are the same for mini-language 1 and mini-language 3). Hence, regardless of where execution resumes after a GOTO, the variable represented by an identifier is the same variable as determined by the scope rules. In Ledgard's paper, the issue is clouded by a discussion of environments, and how environments should be associated with label variables, etc. Clearly, from the above discussion, environments are not an essential part of the semantics of mini-language 3.

In mini-language 3, labels can be assigned to a variable, and transferred to by a statement of the form

GOTO V

where V is a variable. This introduces several considerations, as follows:

1) How are labels to be represented? In the semantic description, labels are represented by themselves, i.e., the label L3 is represented by the string L3. The LABELTABLE attribute of the root node of a program is of the form

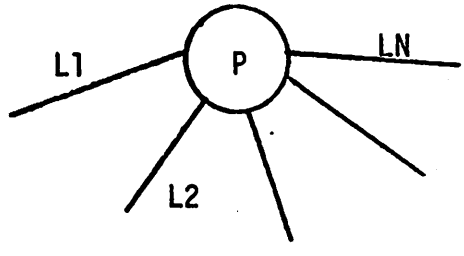


where for each label L, SEL(L,LABELTABLE) is the statement number of the node it labels. Recall that the flowchart for the program is represented by the list of recursion equations or flowchart nodes, each selected by its "statement number". The value of the LABELTABLE attribute will be treated as a constant that can be used in the recursion equations; its value is referred to in the equations by the string LABELTABLE.

2) What type of node corresponds to

GOTO V

where V is a variable? Since V can have as its value any label in the program, the node should have the form



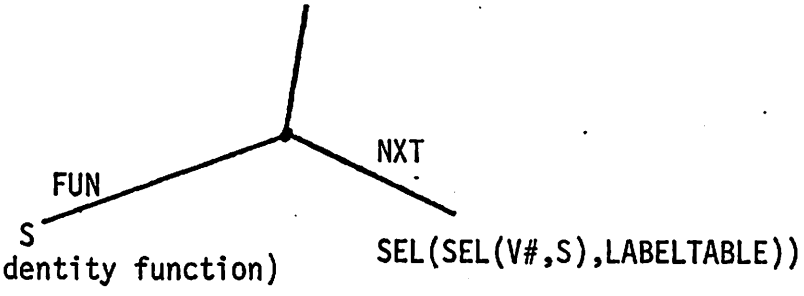
where P is a function from state vectors to {L1,...,LN, ω} and the recursion equation would have the form

$F_{\#}$=if V=L1 then $F_{L1\#}$ else (if V=L2 then $F_{L2\#}$ else (if...else (if V=LN then $F_{LN\#}$ else ω))...)

The representation used corresponds to the flowchart interpretation most closely; the node for

GOTO V

will have the form



where V# is the unique name associated with V. Note that SEL(V#,S) is the current value of V, a label L; and SEL(L,LABELTABLE) is the statement number of the next statement to be executed; LABELTABLE is the constant discussed above.

4.3.3 Formal Definition

The basic functions used in the definition of mini-language 3 are

1) the Vienna operators U0, U, U1 and SEL.

Following the formal definition, the most significant attribute equations will be discussed.

MINI-LANGUAGE 3

SYNTACTIC AND SEMANTIC DESCRIPTION

```
<PROGRAM>A=<BLOCK>B
          <FLOWCHART>A↑F=B↑F
          <LABELTABLE>A↑L=B↑X
          <SYMBOLTABLE>A↑S=B↑V
          B↑N='NUL';B↑G='NUL';B↑L=B↑X

<BLOCK>A='BEGIN ' <DECLARATION>B ';' <COMMAND LIST>C ';END'
          <NEXT>N
          <GLOBALS>G
          <LABELTABLE>L
          <VARTABLE>A↑V=U(A↑G,B↑V)
          <XLABELS>A↑X=C↑X
          <FLOWCHART>A↑F=C↑F
          <STATENO>A↑S=C↑S
          C↑N=A↑N;C↑V=A↑V;C↑L=A↑L

<DECLARATION>A='IDEN ' <VARLIST>B
          <VARTABLE>A↑V=B↑V

<VARLIST>A=<$LETTR>B ',' <VARLIST>C;D=<$LETTR>E
          <VARTABLE>A↑V=U(UO('B↑VAL',UNO()),C↑V);D↑V=UO('E↑VAL',UNO())

<COMMAND LIST>A=<COMMAND>B ';' <COMMAND LIST>C;D=<COMMAND>E
          <VARTABLE>V
          <NEXT>N
          <LABELTABLE>L
          <XLABELS>A↑X=U(B↑X,C↑X);D↑X=E↑X
          <FLOWCHART>A↑F=U(B↑F,C↑F);D↑F=E↑F
          <STATENO>A↑S=B↑S;D↑S=E↑S
          B↑V=A↑V;B↑N=C↑S;B↑L=A↑L
          C↑V=A↑V;C↑N=A↑N;C↑L=A↑L
          E↑V=D↑V;E↑N=D↑N;E↑L=D↑L

<COMMAND>A=<LABEL>B ' ' <UNLABELLED COMMAND>C;D=<UNLABELLED COMMAND>E
          <VARTABLE>V
          <NEXT>N
          <LABELTABLE>L
          <XLABELS>A↑X=UO('B↑V',C↑S);D↑X=E↑X
          <FLOWCHART>A↑F=C↑F;D↑F=E↑F
          <STATENO>A↑S=C↑S;D↑S=E↑S
          C↑N=A↑N;C↑V=A↑V;C↑L=A↑L
          E↑N=D↑N;E↑V=D↑V;E↑L=D↑L

<LABEL>A='L' <$NUMBR>B
          <VALUE>A↑V='L' B↑VAL

<UNLABELLED COMMAND>A=<$LETTR>B '=' <VALUE>C;
.D='GOTO ' <LABEL REF>E;
.F=<BLOCK>G
          <NEXT>N
          <VARTABLE>V
          <LABELTABLE>L
          <STATENO>A↑S=UNO();D↑S=UNO();F↑S=G↑S
          <XLABELS>A↑X='NUL';D↑X='NUL';F↑X=G↑X
          <FLOWCHART>A↑F=UO(A↑S,U(UO('FUN','U(S,UO(' SEL('B↑VAL',A↑V) ',C↑F))'),
```

```
            .UO('NXT',A↑N)));
        .D F=UO(D↑S,U(UO('FUN','S'),UO('NXT','E F')));
            .F↑F=G↑F
            C↑V=A↑V;C↑L=A↑L
            E↑V=D↑V;E↑L=D↑L
            G↑N=F↑N;G↑G=F↑V;G↑L=F↑L

<LABEL REF>A=<LABEL>B;C=<$LETTR>D
            <VARTABLE>V
            <LABELTABLE>L
            <FUNCTION>A F=SEL('B↑V',A↑L);C↑F='SEL(SEL(' SEL('D↑VAL',C↑V)
            . ',S),LABELTABLE)'

<VALUE>A=<$NUMBR>B;C=<$LETTR>D;E=<LABEL>F
            <VARTABLE>V
            <LABELTABLE>L
            <FUNCTION>A↑F=B↑VAL;C↑F='SEL(' SEL('D↑VAL',C↑V) ',S)';
            .E↑F='"F↑V"'
END
```

Corresponding to a <LABEL REF> we have the rule

```
<LABEL REF>A=<LABEL>B;C=<$LETTR>D
:
:
        <FUNCTION>A↑F=SEL('B↑V',A↑L);C↑F='SEL(SEL(' SEL('D↑VAL',C↑V)
        . ',S),LABELTABLE)'
```

If the label reference ℓ is a label, then the function is the constant function whose value is the statement number of the statement labelled by ℓ. If the label reference is a variable $\underline{v}$, the associated function is

SEL(SEL(α,S),LABELTABLE)

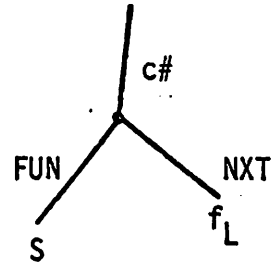
where α is the unique name associated with $\underline{v}$. The value of this function of $\underline{S}$ is the statement number associated with the label which is the current value of $\underline{v}$.

- - - - - - - -

FLOWCHART - flowchart nodes are created for the <UNLABELLED COMMAND> nonterminal just as in mini-language 1, except for an unlabelled command of the form

GOTO L

where $\underline{L}$ is a label reference. For this command, the flowchart node has the form



where f_L is the function associated with the label reference $\underline{L}$.

- - - - - - - -

LABELTABLE - the value of this attribute is the table of all label-statement number pairs for the entire program. It is passed to every command in the program.

4.4 Mini-Language 4: Functions

4.4.1 Ledgard's Description (Verbatim)

Description of Language Elements

Primitive objects in mini-language 4 include, in addition to the natural numbers: 1) a binary function, which maps two natural numbers into the natural number that is their numerical sum; and 2) a quarternary function, which maps four objects - of which the first two are natural numbers and the second two are arbitrary objects - into the third object if the first natural number is greater than the second and into the fourth object if the opposite is true. These binary and quarternary functions are represented by the symbols "+" and "select," respectively.

A simple expression is either one of the primitive symbols {+ select 0 1 2...} or an identifier. The value of a primitive symbol is the primitive object represented by the symbol. The value of an identifier is the object currently linked with the identifier (for linking of identifiers to objects, see definition and evaluation of "let" expressions, below).

A list expression is a list of expressions. The value of a list expression $e_1,e_2,\ldots,e_n$, where the e_i $(1 \leq i \leq n)$ are expressions, is the list of objects $a_1,a_2,\ldots,a_n$ obtained by evaluating (in any

order) each of the component expressions $e_1, e_2, \ldots, e_n$.

A combination is a string of the form $r(\ell)$, where r is an identifier or one of the primitive symbols + or select, and ℓ is a list expression. The value of a combination is obtained by evaluating r and ℓ and then applying the value of r to the value of ℓ. This evaluation is well-defined (i.e., not in violation) only if: the value of r is a function; the number of the components of list expression ℓ is identical to the number of arguments of the function; and if r is one of the symbols + or select, the values of the first two arguments of ℓ must be natural numbers.

The following alternate notations may be used for combinations:

$$(e_1+e_2)$$

in place of

$$+(e_1,e_2)$$

and

$$(\text{if } e_1>e_2 \text{ then } e_3 \text{ else } e_4)$$

in place of

$$\text{select}(e_1,e_2,e_3,e_4).$$

A let expression is a string of either the form

1) $\quad \text{let } i=e_1 \text{ in } e_2$

or

2) $\quad \text{let } f(x_1,\ldots,x_n)=e_1 \text{ in } e_2$

where i is an identifier, f, $x_1,\ldots, x_n$ are identifiers each of which must be different, and e_1 and e_2 are expressions. The value of a let expression of form (1) is computed by evaluating e_1, link-

ing i with the value of e_1, and then evaluating e_2. The value of a let expression of form (2) is computed by forming the function mapping n arguments into the value of the expression e_1 obtained by linking the identifiers $x_1,\ldots,x_n$ with their respective arguments, and linking identifiers other than $x_1,\ldots,x_n$ with their current values; linking f with the function thus formed; and then evaluating e_2.

If the identifier f itself appears in e_1 and is not linked by another let expression that is a subexpression of e_1, then the occurrence of f is taken as part of a recursive definition of f. For example, the let expressions:

1) let Y=3

2) let F(X)=(X+X)

3) let G(N)=(if N>3 then 1 else G(5))

link Y, F, and G with objects whose effective values are respectively: 1) the natural number three; 2) the function mapping a natural number into its double; and 3) the function mapping a natural number into the constant value one.

An expression is either a simple expression, a list expression, a let expression, or a combination.

Program Execution

A program is an expression. The value of a program is the value of the expression. Note that (as mentioned in the first section, "General Considerations") if the evaluation of a program leads to the evaluation of an identifier that is not linked to a value, the program is in violation.

EXAMPLE 1:

```
let F(Y)=(Y+3)
in (F(1)+F(2))
```

EXAMPLE 2:

```
let F(X)     =(X+X)
in let G(P,X)=(P(X)+P(1))
   in G(F,2)
```

EXAMPLE 3:

```
let F(X)=(if X>3 then X
                 else (X+F(X+1))
in F(2)
```

EXAMPLE 4:

```
let Y=2
in let F(X)=(X+Y)
   in F
```

EXAMPLE 5

```
let P(F,G,N)=(if N>100 then 1 else
              G(F,G(N+1)))
in let Q(F,G,N)=(if N>100 then 1 else
                 F(F,G(N+1)))
   in P(P,Q,86)
```

The values of the example programs above are: Example 1, the natural number 9; Example 2, the natural number 6; Example 3, the natural number 9; Example 4, the function mapping some object X (presumably a natural number) into a summation of X and the natural number 2; and Example 5, the natural number 1.

4.4.1 Notation

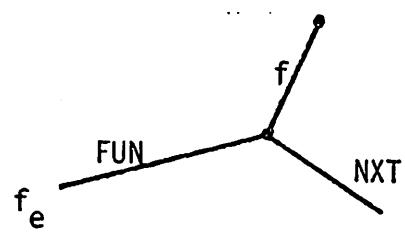
All functions and variable names are assumed to be unique.

4.4.2 Discussion

Mini-language 4 allows functions to be defined, to be passed as arguments to other functions, and to be returned as the value of a function call. Thus, it appears that the data space manipulated by programs in mini-language 4 includes functions. The first question this raises is "how are functions to be represented in the data space?" The mathematical definition of a function, as a set of ordered pairs, cannot be used to represent the function since the set of ordered pairs might be infinite. To simplify the discussion, we will first assume that nested function definitions are not allowed. This restriction will be removed in the next paragraph. In mini-language 4, a function is always defined by an expression, hence the expression itself could be used to represent the function. Equivalently, a table of all declared functions can be defined (in the formal definition, this is MFUNLIB) and the function name used to represent the function. Suppose there is a LET expression of the form:

LET $f(x)=e_1$ in e_2.

Then when the state vector function corresponding to e_1 is defined, references to $\underline{x}$ are replaced by $\underline{SEL(1,SEL(f,S))}$. That is, the selector composed of the function name, and the index of the parameter list, is used to select the value corresponding to the parameter. Let f_{e_1} be the state vector function corresponding to e_1, then an entry is put into MFUNLIB of the form



Note that the "flowchart" representation is maintained even if there is only one node; this is for purposes of uniformity. Whenever the function is passed as an argument, or returned as a value, actually its name is passed. Semantically, what is passed is nothing more than a key, which can be used to reference MFUNLIB. When the function is called, in an expression of the form $f(e_S)$, then the state vector function corresponding to this expression is

$$\text{APPLI}(\text{SEL}(f,\text{MFUNLIB}),\text{UI}(S,1\ f,f_{e_3})$$

(For an explanation of APPLI, see Section 3.3.7.)

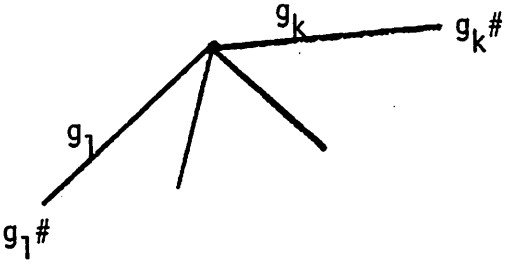
The state vector is modified by assigning the parameter value to the selector 1→f; note above that references to the parameter were made using this selector. In all of the mini-languages with functions, this is the method used to accomplish "parameter passing".

Since nested function definitions (i.e., LET expressions) are allowed in mini-language 4, the evaluation of a function call may result in the evaluation of an inner LET expression and the dynamic definition of a new function. However, the functions defined by the evaluation of a particular LET expression of the form

$$\text{LET } f(x_1,\ldots,x_n)=e_1 \text{ in } e_2$$

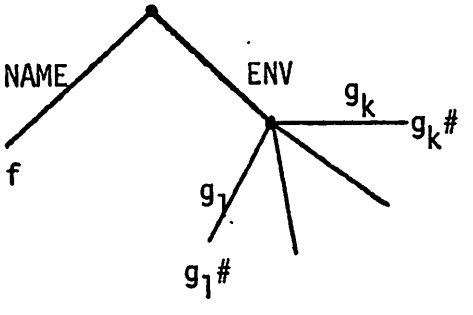
differ only in the bindings of the variables in e_1. Note that actual variables corresponding to the formal parameters are created whenever a function is called, and initialized to the values of the arguments of the function call. Since recursive function calls are allowed, at some point in a computation there may be a number of actual variables corresponding to a formal variable (or parameter).

A binding of the variables in e_1 is a map from formal variables to actual variables. If $G = g_1,\ldots,g_n$ is the set of variables in e_1, a binding of these variables will be represented by a VO of the form



where g_1#,...,g_k# are actual variable names.

Functions will be presented by their names, but when a function is passed as an argument, along with the function name we must pass the current bindings of the variables in the expression defining the function. Thus, the evaluation of a function name yields a VO of the form



where the NAME component is the name of the function, and the ENV component is the current bindings of the variables in the expression defining the function.

When a function is called, the VO representing the bindings of the variables in the expression defining the function is assigned to the ENV component of the state vector. If $\underline{x}$ is a variable in e_1,

references to x are represented by the function

SEL(SEL(X,SEL(ENV,S),S)

Note: Each "variable" in mini-language 4 is actually a formal parameter of some function definition. Let x be a formal parameter of the function f; the arguments to f are passed as a list, and the name of the actual parameter assigned this list is assigned to the f component of the ENV component of the state vector. Thus, if k is the index of x in the formal parameter list, the function representing x has the form

SEL(K,SEL(SEL(F,SEL(ENV,S)),S),S)

A sticky wicket arises in interpreting the built-in function SELECT. What we would like is for SELECT to be one of the basic functions implemented in SNOBOL, and to replace calls to SELECT in the program with calls to the basic function SELECT in the state vector function corresponding to the expression SELECT(e_1,e_2,e_3,e_4)= (if e_1>e_2 then e_3 else e_4). The problem here is, SNOBOL uses call-by-value, and in mini-language 4, SELECT uses call-by-name. Thus, our SNOBOL implemented SELECT will not work correctly. The solution is to define SELECT properly and incorporate the definition of SELECT into the definition of mini-language 4, and this is what was done, as follows, for each expression:

(if e_1>e_2 then e_3 else e_4)

An entry (selected by a unique name #) is made into EXPLIB of the

form

$$F_0 = \text{if } f_{e_1} > f_{e_2} \text{ then } F_1 \text{ else } F_2$$
$$F_1 = f_{e_3}$$
$$F_2 = f_{e_4}$$

and the expression replaced by

APPLI(SEL(#,EXPLIB),S)

This set of recursion equations defines a function identical to the function determined by the expression

$$(\text{if } e_1 > e_2 \text{ then } e_3 \text{ else } e_4)$$

of mini-language 4.

4.4.3 Formal Definition

The basic functions used in the definition of mini-language 4 are

1) the Vienna operators U0, U, U1 and SEL.

2) COND(α,β,γ), where

$\alpha \in \{$TRUE, FALSE, $\omega\}$, β and $\gamma \in N$

COND("TRUE",β,γ)=β and

COND("FALSE",β,γ)=γ

COND(ω,β,γ)= ω

3) GTN(α,β) where α and $\beta \in N$

GTN(α,β)=TRUE if α is greater than β

GTN(α,β)=FALSE otherwise.

Following the formal definition, the most significant attribute equations will be discussed.

MINI-LANGUAGE 4

SYNTACTIC AND SEMANTIC DESCRIPTION

```
4

<PROGRAM>A=<EXPRESSION>B ';END'
         <FLOWCHART>A↑F=U0(0,U(U0('FUN','B↑F'),U0('NXT',)))
         <MFUNLIB>A↑M=B↑M
         <EXPLIB>A↑E=B↑E
         B↑D='NUL'

<EXPRESSION>A=<$NUMBR>B;
.C=<LET EXPRESSION>D;
.E=<COMBINATION>F;
.G=<IDENT>H
          <DECLNAMES>D
          <MFUNLIB>A↑M='NUL';C↑M=D↑M;E↑M=F↑M;G↑M='NUL'
          <EXPLIB>A↑E='NUL';C↑E=D↑E;E↑E=F↑E;G↑E='NUL'
          <FUNCTION>A↑F=B↑VAL;C↑F=D↑F;E↑F=F↑F;G↑F=H↑F
          D↑D=C↑D
          F↑D=E↑D
         H↑D=G↑D

<IDENT>G=<$ALPHA>H
         <DECLNAMES>D
         <FUNCTION>G↑F=COND(EQU(SEL('TYPE',SEL('H↑VAL',G↑D)),'FUN'),
         .'U(U0("NAME","H↑VAL"),U0("ENV",SEL("ENV",S)))',
          .COND(EQU(SEL('TYPE',SEL('H↑VAL',G↑D)),'EXP'),
         .'APPLI(SEL("H↑VAL",MFUNLIB),S)',
         .'SEL(' SEL('NO',SEL('H↑VAL',G↑D)) ',SEL(SEL('" SEL('FNAME',
         .SEL('H↑VAL',G↑D)) '",SEL("ENV",S)),S)'))

<LET EXPRESSION>A='LET ' <$ALPHA>B '=' <EXPRESSION>C 'IN ' <EXPRESSION>D;
.E='LET ' <$ALPHA>F '(' <PARAM LIST>G ')=' <EXPRESSION>H 'IN ' <EXPRESSION>I
         <DECLNAMES>D
         <FUNCTION>A↑F=D↑F;E↑F=I↑F
        <MFUNLIB>A↑M=J(U0('B↑VAL',U0(D,U(U0('FUN','C↑F'),U0('NXT',NUL)))))
         .,D↑M,C↑M);
         .E↑M=U(U0('F↑VAL',U0(0,U(U0('FUN','H↑F'),U0('NXT',NUL)))))
         .,U(H↑M,I↑M))
         <EXPLIB>A↑E=U(C↑E,D↑E);E↑E=U(H↑E,I↑E)
         C↑D=U(A↑D,U0('B↑VAL',U0('TYPE','EXP')));
         .D↑D=C↑D;
         .H↑D=U(E↑D,U(G↑D,U0('F↑VAL',U0('TYPE','FUN'))));
         .I↑D=H↑D
         G↑I=1;G↑N='F↑VAL'

<PARAM LIST>A=<$ALPHA>B ',' <PARAM LIST>C;D=<$ALPHA>E
         <NAME>N
         <INDEX>I
         <DECLNAMES>A↑D=U(C↑D,U0('B↑VAL',U(U(U0('TYPE','PARAM'),U0('NO',A↑I)),
         .U0('FNAME',A↑N))));
         .D↑D=U0('E↑VAL',U(U(U0('TYPE','PARAM'),U0('NO',D↑I)),U0('FNAME',
         .'D↑N')))
         C↑N=A↑N;C↑I=A↑I + 1

<COMBINATION>A='(' <EXPRESSION>B '+' <EXPRESSION>C ')';
.D='(IF ' <EXPRESSION>E '>' <EXPRESSION>F ' THEN ' <EXPRESSION>G ' ELSE '
```

```
.<EXPRESSION>H ')';
.L=<IDENT>M '(' <EXPRESSION LIST>N ')'
          <DECLNAMES>D
           <UNO>A↑U='NUL';D↑U=UNO();L↑U='NUL'
          <MFUNLIB>A↑M=U(B↑M,C↑M);
          .D↑M=U(U(U(E↑M,F↑M),G↑M),H↑M);
          .L↑M=N↑M
          <EXPLIB>A↑E=U(B↑E,C↑E);
          .D E=U(U(U(E↑E,F↑E),G↑E),H↑E,
           .UO('SELECT' D↑U,U(MKFLC(0,'S','COND(GTN(E↑F ,F↑F),1,2)'),
          .MKFLC(1,'G↑F',),MKFLC(2,'H↑F',))));
          .L↑E=N↑E
          <FUNCTION>A↑F='B↑F + C↑F';
           .D↑F='APPLI(SEL("SELECT" D U,EXPLIB),S)';
          .L↑F='APPLI(SEL(SEL("NAME",M↑F),MFUNLIB),
          .U(S,UO("ENV",U(SEL("ENV",M↑F),SEL("NAME",M↑F),UNO())),
          .UO(UNIQUENUMBER,N↑F)))'
          B↑D=A↑D
          C↑D=A↑D
          E↑D=D↑D
          F↑D=D↑D
          G↑D=D↑D
          H↑D=D↑D
          M↑D=L↑D
          N↑D=L↑D;N↑I='1'

<EXPRESSION LIST>A=<EXPRESSION>B ';' <EXPRESSION LIST>C;D=<EXPRESSION>E
          <DECLNAMES>D
          <INDEX>I
          <MFUNLIB>A↑M=U(B↑M,C↑M);D↑M=E↑M
          <EXPLIB>A↑E=U(B↑E,C↑E);D↑E=E↑E
          <FUNCTION>A↑F='U(C↑F,UO(A↑I,B↑F))';D↑F='UO(D↑I,E↑F)'
          B↑D=A↑D
          C↑D=A↑D;C↑I=A↑I + 1
          E↑D=D↑D
END
```

The function f_i corresponding to the evaluation of an identifier $\underline{i}$ is given by the rule

```
<IDENT>G=<$ALPHA>H
        <FUNCTION>G↑F=COND(EQU(SEL('TYPE',SEL('H↑VAL',G↑D)),'FUN'),
       .'U(UO(,NAME,"H↑VAL"),UO(ENV,SEL("ENV",S)))',
         .COND(EQU(SEL('TYPE',SEL('H↑VAL',G↑D)),'EXP'),
       .'APPLI(SEL("H↑VAL",MFUNLIB),S)',
       .'SEL(' SEL('NO',SEL('H↑VAL',G↑D)) ',SEL(SEL(" SEL('FNAME',
       .SEL('H↑VAL'H↑VAL',G↑D)) '",SEL("ENV",S)),S))'))
```

There are three cases, as follows:

1) $\underline{i}$ is a function name; in this case, f_i has the form

   U(UO(NAME,i),UO(ENV,SEL(ENV,S))).

2) If $\underline{i}$ has been associated with an expression, then the flowchart for this expression is in MFUNLIB, and the function is

   APPLI(α,S)

   where α is the flowchart in MFUNLIB.

3) If $\underline{i}$ is a parameter, then the function f_i has the form

   SEL(α,SEL(SEL(β,SEL(ENV,S)),S))

   where α is the index of $\underline{i}$ in the parameter list for the function named β.

- - - - - - - -

MFUNLIB - is created by the rule

```
<LET EXPRESSION>A='LET ' <$ALPHA>B '=' <EXPRESSION>C 'IN ' <EXPRESSION>D;
.E='LET ' <$ALPHA>F '(' <PARAM LIST>G ')=' <EXPRESSION>H 'IN ' <EXPRESSION>I

:
:

      <MFUNLIB>A↑M=U(UO('B↑VAL',UO(0,U(UO('FUN','C↑F'),UO('NXT',NUL))))
       .,D↑M,C↑M);
       .E↑M=U(UO('F↑VAL',UO(0,U(UO('FUN','H↑F'),UO('NXT',NUL))))
       .,U(H↑M,I↑M))
```

This table contains the flowchart of the function corresponding to each declared identifier other than a parameter. Note that each flowchart has only one node, since there is no sequencing in mini-language 4.

- - - - - - - -

EXPLIB is created by the rule

```
<COMBINATION>A='(' <EXPRESSION>B '+' <EXPRESSION>C ')';
.D='(IF ' <EXPRESSION>E ' ' <EXPRESSION>F ' THEN ' <EXPRESSION>G ' ELSE '
.<EXPRESSION>H ')';
.L= <IDENT>M '(' <EXPRESSION LIST>N ')'

:
:

       <EXPLIB>A↑E=U(B↑E,C↑E);
       .D↑E=U(U(U(E↑E,F↑E),G↑E),G↑E,
        .UO('SELECT' D↑U,U(MKFLC(0,'S','COND(GTN(E↑F ,F↑F),1,2)'),
       .MKFLC(1,'G↑F',),MKFLC(2,'H↑F',))));
       .L↑E=N↑E
```

EXPLIB is fully discussed in the preceding section. Corresponding to the same syntactic rule is the semantic rule defining the FUNCTION attribute

```
<FUNCTION>A↑F='B↑F   C↑F';
.D↑F='APPLI(SEL(,SELECT D↑U,EXPLIB),S)';
.L↑F='APPLI(SEL(SEL("NAME",M↑F), MFUNLIB),
.U(S,UO("ENV",U1(SEL("ENV",M↑F),SEL("NAME",M↑F),UNO())),
.UO(UNIQUENUMBER,N↑F)))'
```

If the combination is of the form $i(\ell)$, where $\underline{i}$ is an identifier and

ℓ a list of expressions, the function corresponding to this combination is

APPLI(SEL(α,MFUNLIB),U(S,U1(β,α,#),UO(#,f_L)))

where α is the name of the function corresponding to i, β is the VO representing the bindings of the variables in the expression defining α and # is a unique name.

Note: Mini-language 4 is the only mini-language in which UNO( ) appears in the recursive functionals corresponding to a program. UNO( ) is implemented by incrementing the SNOBOL variable UNIQUENUMBER and taking its value, and thus is clearly not a monotonic function of the state vector; however, it is easy to implement. UNO could be defined as follows:

UNO(S)=the least integer not labelling any branch in S.

With this definition, UNO(S) is a monotonic function of S, and in the above semantic equation, UNO( ) and UNIQUENUMBER would be replaced by UNO(S).

4.5 Mini-Language 5: Passing of Parameters

4.5.1 Ledgard's Description (Verbatim)

Description of Language Elements

Array identifiers are composed of the symbols A1 A2.... The value of an array identifier is a one-dimensional array with ten (initially unspecified) elements. Procedure identifiers comprise the symbols P1 P2..., and parameter identifiers comprise the symbols a b ... z.

A parameter specification is a string of one of the following forms: exp ℓ, copy val ℓ, or loc ℓ, where ℓ is a list of parameter identifiers. A procedure declaration is a string of the form proc p(ℓ) s c end, where p is a procedure identifier, ℓ a list of parameter identifiers, s a sequence of parameter specifications, and c is a sequence of assignment commands (defined below) such that: each parameter identifier in ℓ is different; each parameter identifier in s is different and occurs as one of the parameters in ℓ; and each identifier in c is a parameter identifier. For example, the following declaration violates each of the above requirements

```
proc P1(a,b,a)
     exp a,d
     loc a
     D:=1
end
```

An expression is either a numeral or a named expression. A named expression is either an identifier, a parameter identifier, or an array identifier followed by an expression enclosed in square brackets.

An assignment command is a string of the form e_1:=e_2, where e_1 is a named expression, and e_2 is an expression. The execution of an assignment command results in assigning the value of e_2 to the location denoted by e_1. (For a more detailed discussion of assignment, see mini-language 2.) In particular, if e_1 or e_2 contains an expression of the form a[e], where a is an array identifier and e is an expression whose value is the natural number n, then

a[e] denotes the n-th element of the array a.*

A procedure command is a string of the form p(ℓ), where p is a procedure identifier, and ℓ is a list of expressions. The evaluation of the procedure command of this form is as follows:

1) Obtain the sequence of assignment commands c given in the declaration of p.
2) For each parameter i called by "expression", replace each occurrence of i in c by the corresponding expression e in ℓ.
3) For each parameter i called by "copy value", prefix to c the command i:=e, where e is the corresponding expression for i in ℓ.
4) For each parameter i called by "location," obtain the corresponding expression e for i: if e is an identifier, let L_i denote its location; and if e is an array expression A[e'], let L_i denote the location of the n-th element of A, where n is the value e'. Then replace each occurrence of i in c by L_i.
5) Execute the sequence of assignment commands c', formed by the above rules, with the following interpretation for parameters called by location: if an L_i is evaluated on the right-hand side of an assignment command, its value is the object currently stored in the location L_i; and if an L_i appears on the left-hand side of an assignment command, the assignment of the value of the right-hand side is made to the location L_i.

---

*For example, if the value of the array name A1 is the array whose ten elements are 4,A,A,5,5,A,1,2,3,4 (where A denotes an unspecified element), then the value of A1[7] is 1, and the value of A1[A1[7]] is 4.

Program Execution

A program consists of a sequence d of procedure declarations followed by a sequence c of assignment and procedure commands such that:

each declared procedure identifier in d is different;

each procedure identifier in c is declared in d;

the number of expressions in the expression list of a procedure command is identical to the number of parameters in the corresponding procedure declaration;

no parameter identifiers occur in c; and

no procedure declaration in d contains a parameter that is called by location, appears on the left side of an assignment statement, and corresponds to an expression that is a numeral - i.e., a condition that leads to an assignment to a numeral is syntactically illegal.

EXAMPLE 1:

```
proc P1(a,b)
     copy val a,b
     x:=a
     a:=b
     b:=x
     end
I    :=3
A[I] :=6
P1(I,A[i])
```

EXAMPLE 2:

```
proc P2(a,b)
     location a,b
     x:=a
     a:=b
     b:=x
     end
I    :=3
A[I] :=6
P2(I,A[I])
```

EXAMPLE 3:

```
proc P3(a,b)
     exp a,b
     x:=a
     a:=b
     b:=x
     end
I    :=3
A[I] :=6
P3(I,A[I])
```

Suppose we wish to define and use a procedure that swaps the values of its two arguments, and we must select one of the three options above. Which one shall we choose? The three examples are identical except that the parameter specifications in procedures P1, P2, and P3 are different.

In Example 1: the values of I and A[3] will be set to 3 and 6, respectively; and the procedure call P1(I,A[I]) will be executed - i.e., the values of a and b will be set to 3 and 6, respectively, and the values of x, a, and b will be set to 3, 6, and 3, respectively. The values of I and A[3] will remain unchanged, so we shall not elect P1.

In Example 2: the values of I and A[3] will be set to 3 and 6, respectively; and the procedure call P2(I,A[I]) will be executed -

i.e., the locations L_a of I and L_b of A[I] will be determined, and the commands $x:=L_a$, $L_a:=L_b$, and $L_b:=x$ will be executed, which will result in settling (sic) the values of I and A[3] to 6 and 3, respectively. Thus, procedure P2 of Example 2 yields the desired outcome.

In Example 3 we have a rather surprising result:

1) the values of I and A[3] will be set to 3 and 6, respectively; and
2) the procedure call P3(I,A[I]) will be executed - i.e., the commands

```
x:=I
I:=A[I]
A[I]:=x
```

will be executed, which will result in 3 being assigned to x, 6 to I, and then 3 to A[6].

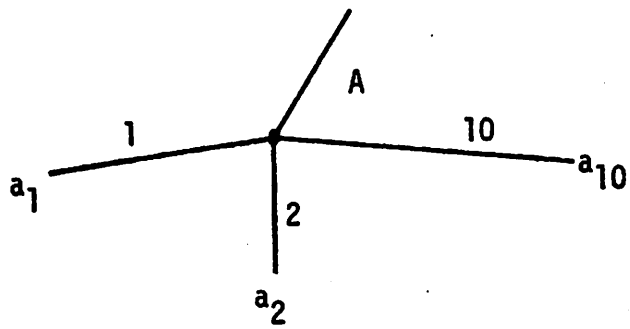
Therefore, we elect only procedure P2.

4.5.2 Discussion

Mini-language 5 has two new features that have not been encountered in the previous four mini-languages - arrays, and a variety of parameter passing mechanisms.

First arrays will be discussed. Suppose A is an array and the elements of A are $a_1, a_2, \ldots, a_{10}$. A will be represented by a Vienna

object having the form



In a statement I=A[5], the state vector function corresponding to the expression A[5] is

$$f_{A[5]}=SEL(5,SEL(A,S))$$

and the state vector function corresponding to I=A[5] is

$$U1(S,I,f_{A[5]})$$

If the statement has the form I=A[e], where e is an arbitrary expression, the state vector function corresponding to A[e] is

$$SEL(f_e,SEL(A,S))$$

where f_e is the state vector function associated with the expression e. The state vector function corresponding to the statement

$$A[e_1]=e_2$$

is

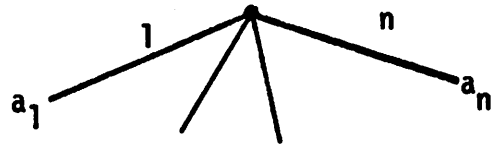
$$U1(S,f_{e_1}\rightarrow A,f_{e_2}).$$

The expression $f_{e_1}\rightarrow A$ evaluates to the composite selector $v\rightarrow A$ where v is the value of f_{e_2}.

Like mini-language 2, mini-language 5 emphasizes the difference between evaluation of expressions occurring on right and left sides of the equals sign. Consequently, each named expression in mini-language 5 (array references and identifiers) has two state vector

functions associated with it (LHSFUN and RHSFUN), one which corresponds to evaluating the expression on the left of an equals sign, and one which corresponds to evaluating the expression on the right.

Parameters are passed basically as in mini-language 4. Let Pl(ℓ) be an expression, where Pl is a function name, and ℓ is a list $\ell_1,\ldots,\ell_n$ of expressions. Then the state vector function f_e corresponding to ℓ will, when applied to a state vector $\underline{S}$, produce the value



i.e., a Vienna object representing the list of argument values. The state vector function corresponding to Pl(ℓ) will be

APPLI(SEL(Pl,PROCLIB),Ul(S,PLIST,f_e)).

Parameters may be passed in three ways in mini-language 5: call-by-value, call-by-location, and call-by-expression. Let $x_1,\ldots,x_n$ be the formal parameters of $\underline{f}$ above; then

if x_i is called by value, $a_i=f^R_{e_i}$, where
$f^R_{e_i}$ is the function corresponding to evaluating e_i on the right of an equals sign.

if x_i is called by location, $a_i=f^L_{e_i}$, where
$f^L_{e_i}$ is the function corresponding to evaluating e_i on the left of an equals sign

if x_i is called by expression, the case is more complicated. As in mini-language 4, a name for e_i will be passed as

the argument. In order to do this, the semantic equations create a name for e_i, and e_i is placed in the table ARGLIB. The argument passed is the name, thus

a_i=name of e_i in ARGLIB

In order to be able to determine how each argument is passed, the SPECTABLE attribute is added to whenever a procedure is declared. This table can be referenced with a function name and parameter index (its position in the parameter list) to determine how that parameter is passed.

There are four possible interpretations for an identifier occurring in the body of a procedure declaration, as follows:

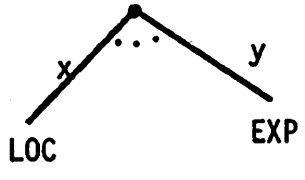
1) It can denote a simple variable.

2) It can denote a parameter called by value.

3) It can denote a parameter called by location.

4) It can denote a parameter called by expression.

In each of these cases, the LHSFUN and the RHSFUN corresponding to the identifier are determined differently. To keep track of what each identifier denotes, the PARAMTABLE attribute is created for each procedure declaration; this table has the form



where x, y, are the formal parameters to the procedure, and the component corresponding to a formal parameter is the string VAL, LOC, or EXP, depending on whether the parameter is passed by value,

location or expression. PARAMTABLE is an inherited attribute for each statement in the body of the procedure declaration. If an identifier is not in PARAMTABLE, then it denotes a simple identifier.

4.5.3 Formal Definition

The basic functions used in the definition of mini-language 5 are

1) the Vienna operators UO, U, U1 and SEL.

Following the formal definition, the most significant attribute equations will be discussed.

MINI-LANGUAGE 5

SYNTACTIC AND SEMANTIC DESCRIPTION

```
<PROGRAM>A=<DEC LIST>B ';' <COMMAND LIST>C ';END'
        <FLOWCHART>A↑F=C↑F
        <PROCLIB>A↑P=B↑P
        <ARGLIB>A↑A=C↑A
        C↑N='NUL';C↑P='NUL';C↑S=B↑S

<DECLARATION>A='PROC P' <$NUMBR>B '(' <VARLIST>C ');' <SPEC LIST>D ';'
. <COMMAND LIST>F ';END'
        <PROCLIB>A↑P=UO('B↑VAL',F↑F)
        <SPECTABLE>A↑S=UO('B↑VAL',D↑S)
        C↑I=1
        D↑P=C↑V
        F↑N='NUL';F↑P=C↑V;F↑S=D↑S

<DEC LIST>A=<DECLARATION>B ';' <DEC LIST>C;D=<DECLARATION>E
        <PROCLIB>A↑P=U(B↑P,C↑P);D↑P=E↑P
        <SPECTABLE>A↑S=U(B↑S,C↑S);D↑S=E↑S

<VARLIST>A=<$ALPHA>B ',' <VARLIST>C;D=<$ALPHA>E
        <INDEX>I
        <VARLIST>A↑V=U(UO('B↑VAL',A↑I),C↑V);D↑V=UO('E↑VAL',D↑I)
        C↑I=A↑I + 1

<SPEC LIST>A=<SPEC>B ',' <SPEC LIST>C;D=<SPEC>E
        <PARAMTABLE>P
        <SPECTABLE>A↑S=U(B↑S,C↑S);D↑S=E↑S
        B↑P=A↑P
        C↑P=A↑P
        E↑P=D↑P

<SPEC>A='EXP ' <SVARLIST>B;
.C='COPY VAL ' <SVARLIST>D;
.E='LOC ' <SVARLIST>F
        <PARAMTABLE>P
        <SPECTABLE>A↑S=B↑S;C↑S=D↑S;E↑S=F↑S
        B↑P=A↑P;B↑T='EXP'
        D↑P=C↑P;D↑T='COPY VAL'
        F↑P=E↑P;F↑T='LOC'

<SVARLIST>A=<$ALPHA>B ',' <SVARLIST>C;D=<$ALPHA>E
        <PARAMTABLE>P
        <TYPE>T
        <SPECTABLE>A↑S=U(UO(SEL('B↑VAL',A↑P),'A↑T'),C↑S);
        .D S=UO(SEL('E↑VAL',D↑P),'D↑T')
        C↑P=A↑P;C↑T=A↑T

<COMMAND LIST>A=<COMMAND>B ';' <COMMAND LIST>C;D=<COMMAND>E
        <NEXT>N
        <PARAMTABLE>P
        <SPECTABLE>S
        <LINENO>A↑L=B↑L;D↑L=E↑L
        <FLOWCHART>A↑F=U(B↑F,C↑F);D↑F=E↑F
        <ARGLIB>A↑A=U(B↑A,C↑A);D↑A=E↑A
        B↑N=C↑L;B↑P=A↑P;B↑S=A↑S
        C↑N=A↑N;C↑P=A↑P;C↑S=A↑S
```

```
          E↑N=D↑N;E↑P=D↑P;E↑S=D↑S

<COMMAND>A=<ASSIGNMENT COMMAND>B;C=<PROCEDURE COMMAND>D
          <NEXT>N
          <PARAMTABLE>P
          <SPECTABLE>S
          <LINENO>A↑L=B↑L;C↑L=D↑L
          <FLOWCHART>A↑F=B↑F;C↑F=D↑F
          <ARGLIB>A↑A='NUL';C↑A=D↑A
          B↑N=A↑N;B↑P=A↑P;B↑S=A↑S
          D↑N=C↑N;D↑S=C↑S

<ASSIGNMENT COMMAND>A=<EXPRESSION>B '=' <EXPRESSION>C
          <NEXT>N
          <PARAMTABLE>P
          <SPECTABLE>S
          <LINENO>A↑L=UNO()
          <FLOWCHART>A↑F=UO(A↑L,U(UO('FUN','U1(S,B↑L,C↑R)'),UO('NXT',A↑N)))
          B↑P=A↑P;B↑S=A↑S
          C↑P=A↑P;C↑S=A↑S

<PROCEDURE COMMAND>A='P' <$NUMBR>B '(' <ARGLIST>C ')'
          <NEXT>N
          <SPECTABLE>S
           <LINENO>A↑L=UNO()
           <FLOWCHART>A↑F=UO(A↑L,U(UO('FUN',' APPLI(SEL("B↑VAL",PROCLIB),
           .U(S,UO(PLISTL,C↑F))) '),UO('NXT',A↑N)))
           <ARGLIB>A↑A=C↑A
           C↑N=B↑VAL;C↑I=1;C↑S=A↑S

<ARGLIST>A=<ARG>B ',' <ARGLIST>C;D=<ARG>E
          <NAME>N
          <INDEX>I
          <SPECTABLE>S
          <FUNCTION>A↑F='U(B↑F,C↑F)';D↑F=E↑F
          <ARGLIB>A↑A=U(B↑A,C↑A);D↑A=E↑A
          B↑N=A↑N;B↑I=A↑I;B↑S=A↑S
          C↑N=A↑N;C↑I=A↑I + 1;C↑S=A↑S
          E↑N=D↑N;E↑I=D↑I;E↑S=D↑S

<EXPRESSION>A=<$NUMBR>B;
.C=<$ALPHA>D;
.E=<$ALPHA>F '=' <EXPRESSION>G '-'
          <PARAMTABLE>P
          <SPECTABLE>S
          <LHSFUN>A↑L='NUL';
          .C↑L=COND(EQU(SEL(SEL('D↑VAL',C↑P),C↑S),'COPY VAL'),'NUL',
           .COND(EQU(SEL(SEL('D↑VAL',C↑P),C↑S),'LOC'),
           .'SEL(' SEL('D↑VAL',C↑P) ',SEL(PLISTL,S))',
           .COND(EQU(SEL(SEL('D↑VAL',C↑P),C↑S),'EXP'),
           .'SEL("LHSVAL",APPLI(SEL(SEL(' SEL('D↑VAL',C↑P) ',SEL(PLISTL,S))
           .,ARGLIB),S))',
           .'"D↑VAL"')));
           .E↑L='COMPOS(G↑R,"F↑VAL")'
           <RHSFUN>A↑R=B↑VAL;
          .C↑R=COND(EQU(SEL(SEL('D↑VAL',C↑P),C↑S),'COPY VAL'),
           .'SEL(' SEL('D↑VAL',C↑P) ',SEL(PLISTL,S))',
           .COND(EQU(SEL(SEL('D↑VAL',C↑P),C↑S),'LOC'),'SEL(C↑L,S)',
           .COND(EQU(SEL(SEL('D↑VAL',C↑P),C↑S),'EXP'),
           .'SEL("RHSVAL",APPLI(SEL(SEL(' SEL('D↑VAL',C↑P) ',SEL(PLISTL,S))
```

```
                .,ARGLIB),S))',
                .'SEL("D↑VAL",S)')));
                .E↑R='SEL(E↑L,S)'
                G↑P=E↑P;G↑S=E↑S

<ARG>A=<EXPRESSION>B
                <NAME>N
                <INDEX>I
                <SPECTABLE>S
                <UNO>A↑U=UNO()
                <FUNCTION>A↑F=COND(EQU(SEL(A↑I,SEL(A↑N,A↑S)),'COPY VAL'),'UO(A↑I, B↑R)'
               .,COND(EQU(SEL(A↑I,SEL(A↑N,A↑S)),'LOC'),'UO(A↑I, B↑L)',
                .'UO(A↑I,A↑U)'))
                <ARGLIB>A↑A=COND(EQU(SEL(A↑I,SEL(A↑N,A↑S)),'EXP'),
                .UO(A U,UO(O,U(UO('FUN','U(UO("LHSVAL",B↑L),UO("RHSVAL",B↑R))'),
                .UO('NXT',NUL))))),
                .'NUL')
                B↑P='NUL';B↑S='NUL'
END
```

PROCLIB is a table containing the flowchart for each declared procedure. The PROCLIB is created by the rule

```
<DECLARATIONB>A='PROC P' <$NUMBR>B '(' <VARLIST>C ');' <SPEC LIST>D ';'
. <DECLARATIONA>E ';' <COMMAND LIST>F ';END'
        . <PROCLIB>A↑P=UO('B↑VAL',U(E↑F,F↑F))
```

- - - - - - - -

ARGLIB is a table of all expressions declared by virtue of being passed as an argument called by expression. Such an expression is given a unique name and placed in ARGLIB. Expressions are put in ARGLIB by the rule

```
<ARG>A=<EXPRESSION>B
:
      <ARGLIB>A↑A=COND(EQU(SEL(A↑I,SEL(A↑N,A↑S)),'EXP'),
      .UO(A↑U,UO(0,U(UO('FUN','U(UO("LHSVAL",B↑L),UO("RHSVAL",B↑R))'),
      .UO('NXT',NUL)))),
      .'NUL')
```

The SPECTABLE contains a declaration for each parameter. It is created as follows, in a procedure declaration of the form

```
<DECLARATIONB>A='PROC P' <$NUMBR>B '(' <VARLIST>C ');' <SPEC LIST>D ';'
. <DECLARATIONA>E ';' <COMMAND LIST>F ';END'
:
                  <SPECTABLE>A↑S=UO('B↑VAL',D↑S)
```

The list of formal parameters is passed to the specification list to be used in creating the SPECTABLE. The SPECTABLE is passed from the specification list to the declaration and to every command in the program.

- - - - - - - -

The LHSFUN corresponding to an identifier i is given by

```
<EXPRESSION>A=<$NUMBR>B;
.C=<$ALPHA>D;
.E=<$ALPHA>F '[' <EXPRESSION>G ']'
:
:
            <LHSFUN>A↑L='NUL';
            .C↑L=COND(ELEM('D↑VAL',C↑V),'.D↑VAL',
            .COND(EQU(SEL(SEL('D↑VAL',C↑P),C↑S),'COPY VAL'),'NUL',
            .COND(EQU(SEL(SEL('D↑VAL',C↑P),C↑S),'LOC'),
            .'SEL(' SEL('D↑VAL',C↑P) ',SEL(PLISTL,S))',
            .'SEL("LHSVAL",APPLI(SEL(SEL(' SEL('D↑VAL',C↑P) ',SEL(PLISTL,S))
            .,ARGLIB),S))'))));
            B↑L='COMPOS(G↑R,"F↑VAL")'
```

There are four cases, as follows:

1) i is a parameter called by value; in this case, the LHSFUN corresponding to i is null.

2) i is a parameter called by location; in this case, the function corresponding to i is equivalent to

   SEL(α,S)

   where α is the composite selector corresponding to i.

3) i is a parameter called by expression; in this case, the function corresponding to i is the LHSFUN of the passed parameter, given by

   SEL("LHSVAL",APPLI(SEL(SEL(α,S),ARGLIB),S))

   where again, α is the composite selector corresponding to i.

4) If i is a simple variable, the function is the constant function with value i.

RHSFUN for an identifier is defined similarly.

- - - - - - - -

The FUNCTION corresponding to an argument e called by value is f_e^R, where f_e^R corresponds to evaluating e on the right of an assignment. If e is called by location, the function corresponding to the argument is f_e^L, where f_e^L corresponds to evaluating e on the left of an assignment. If e is called by expression, e is given a unique name e# and entered into ARGLIB; the function corresponding to the argument e is the e#. These functions are defined by the rule

```
<ARG>A=<EXPRESSION>B
:
    <FUNCTION>A↑F=COND(EQU(SEL(A↑I,SEL(A↑N,A↑S)),'COPY VAL'),'UO(A↑I,B
.,COND(EQU(SEL(A↑I,SEL(A↑N,A↑S)),'LOC'),'UO(A↑I, B↑L)',
.'UO(A↑I,A↑U)'))
```

- - - - - - - -

4.6 Mini-Language 6: Static Type Checking

4.6.1 Ledgard's Description (Verbatim)

Description of Language Elements

A type designation is either: 1) the symbol "N," in which case it denotes the class of natural numbers; or 2) a string of the form $(t_1,\ldots,t_n \rightarrow t_{n+1})$, where the t_i, $1 \leq i \leq n+1$, are type designations, in which case it denotes a class of functions whose domain is the Cartesian product of the classes denoted by $t_1,\ldots,t_n$ and whose range is the class denoted by t_{n+1}.

Primitive objects in mini-language 6 include, in addition to the natural numbers, a unary function, SQ, for computing the square of a natural number, and an infix binary function, +, for computing the sum of two natural numbers. The natural numbers have a type N,

and the functions SQ and + have types (N→N) and (N,N→N).

A declaration specifies an identifier as representing a class of objects of only one type.* A declaration consists of a string of either of the following forms:

1) dec ℓ type t

2) dec f(ℓ) := e where ℓ_t

where ℓ is a list of identifiers, t a type designation, f an identifier, e an expression (defined below), and ℓ_t is a list of type designations such that each identifier in ℓ is different; each identifier in e occurs in ℓ; and the number of type designations in ℓ_t is identical to the number of identifiers in ℓ.

A declaration of form (1) specifies that each identifier in ℓ has the type t. A declaration of form (2): a) assigns to f the function from arguments in the domain, denoted by $t_1,\ldots,t_n=\ell_t$, into the value of the expression e, obtained by replacing each identifier in ℓ by its corresponding argument; and b) specifies that the type of f is $(t_1,\ldots,t_n\rightarrow t_{n+1})$, where $t_1,\ldots,t_n=\ell_t$ are the type designations given in the declaration of f, and t_{n+1} is the type of the result expression e (defined below), obtained by using the types $t_1,\ldots,t_n$ for the corresponding identifiers of ℓ in e.

An expression consists of a string of one of the following forms:

1) p

2) i

*Note that this requirement is not made in mini-language 7, where the types of objects assigned to identifiers may vary.

3) SQ(e)

(provided e is of type N)

4) (d+e)

(provided d and e are of type N)

5) $f(e_1,\ldots,e_n)$

(provided f is of type $(t_1,\ldots,t_n \rightarrow t_{n+1})$, and $e_1,\ldots,e_n$ are of corresponding type $t_2,\ldots,t_n$)

6) (if d>e then e_1 else e_2)

(provided d and e are of type N, and e_1 and e_2 are of the same type),

where p is a primitive symbol, i and f are identifiers, and d, e, $e_1,\ldots,e_n$ are expressions.

Expressions are of the following types:

1) an expression of form (1) is of the same type as p;

2) an expression of form (2) is of the type declared for i;

3) if e is of type N, then an expression of form (3) is of type N;

4) if d and e are of type N, an expression of form (4) is of type N;

5) if the types of $e_1,\ldots,e_n$ are $t_1,\ldots,t_n$, and if f is declared to be of type $(t_1,\ldots t_n \rightarrow t_{n+1})$, then an expression of form (5) is of type t_{n+1}; and

6) if d, e are of type N, and e_1, e_2 are of the same type, then an expression of form (6) is of the same type as e_1 or e_2.

An assignment command consists of a string of the form i:=e, where i is an identifier, and e is an expression such that the

types of i and e must be identical.

A program consists of a sequence d of declarations followed by a sequence c of assignment commands such that each identifier occurring in c is declared once and only once in d, and all of the above requirements on types are satisfied.

Program Execution

The commands of a program are executed sequentially, with the conventional menaing for addition, the squaring operation, assignment, conditional expressions, and functional application.

EXAMPLE 1:

```
dec A,B              type N
dec F(X):=SQ(SQ(X)) where N
A:=2
B:=F(A)
```

EXAMPLE 2:

```
dec X                    type N
dec A,B                  type (N→N)
dec F(X)  :=SQ(SQ(X)) where N
dec G(X,Y):=X(Y)         where
                         (N→N),N
A:=SQ
B:=F
X:=(G(A,1)+G(B,2))
```

EXAMPLE 3 (syntactically illegal):

```
dec A,B.           type N
dec F(X)  :=(X+X) where N
dec G(X,Y):=(X+Y) where N,N
dec H(X,Y):=X(Y)  where (N→N),N
A:=H(F,1)
B:=H(G,1)
```

EXAMPLE 4 (syntactically illegal):

```
dec A              type N
dec B              type (N→N)
dec F(X)  :=X      where N
dec G(X)  :=X      where (N→N)
dec H(X,Y):=X(Y) where ((N→N)→
                   (N→N)),(N→N)
B:=H(G,SQ)
A:=B(2)
B:=H(F,2)
```

Example 1 results in setting the value of B to 16. Example 2 results in setting the value of X to 17.

Example 3 is illegal because of the statement "B:=H(G,1)," where H, a function of type ((N→N),N→N), is applied to arguments G, 1 of types (N,N→N),N. Using the rules given previously, we can determine that the program of Example 3 is syntactically illegal:

1) A and B are declared to be of type N;

2) F is declared to be of type (N→N);

3) G is declared to be of type (N,N→N);

4) H is declared to be of type ((N→N), N→N);

5) H(F,1), where F, 1 are of types (N→N), N and H is of type ((N→N), N→N), is of type N;

6) A:=H(F,1) is type-wise of the form N:=N, hence is type-wise correct; therefore,

7) H(G,1), where G, 1 are of types (N, N→N), N and H is of type ((N→N), N→N), is illegal because the type (N,N→N) of G is not identical to the type (N→N) of the first argument of H.

Example 4 is illegal because of the statement "B:=H(F,2)," where H, a function of type (((N→N)→(N→N)), (N→N)→(N→N)), is ap-

plied to arguments F and 2 of types (N→N) and N. Note, however, that the application of H to F and 2 is semantically well-defined, i.e., results in applying F to 2, which returns the value 2. (Programs like that of Example 4 will be allowed in mini-language 7.) Note, also, that in all syntactically legal programs in mini-language 6, no condition leading to a type error can arise during execution.

4.6.2 Discussion

In mini-language 6, a program containing an expression having a type conflict is considered by Ledgard to be syntactically illegal. However, there is no context-free grammar that generates only the programs in mini-language 6 that are free of type conflicts, i.e., the restriction of type compatibility is not enforceable by a context-free grammar. The approach taken here is to give a grammar for mini-language 6 which generates all programs in mini-language 6 that are free of type conflicts, and also generates programs having type conflicts. Type conflicts are regarded as semantic errors; recall that

$$Sem_{\text{mini-language 6}}(P)$$

is the map computed by any syntactically legal program P in mini-language 6. $Sem_{mini\text{-}6}$ will be defined so that if P contains a type error, $Sem_{mini\text{-}6}(P)$ will be the empty function. In order to make things more explicit, the nonterminal corresponding to a complete program is given a special attribute, called CHECKTYPE. Also, each nonterminal that may generate a phrase having a type error has this attribute. If the program (phrase) is free of type errors, this

attribute will have the value "LEGAL". If a type error occurs in the program (phrase), CHECKTYPE will have the value "ILLEGAL". If the value of CHECKTYPE for a program is "ILLEGAL", the value of the FLOWCHART attribute will be null.

Type checking occurs in three places: where an expression is evaluated, where a value is assigned to a variable, and where an argument is passed to a procedure. Each expression, assignment, and argument is checked for type correctness and compatibility; the result of this check becomes the value for the expression, assignment, or argument. The semantic equations for each production higher in the parse tree check the CHECKTYPE attributes of the descendent nodes, and if at least one of them has the value "ILLEGAL", the value of the CHECKTYPE attribute for the node representing the left side nonterminal is "ILLEGAL". Thus, an "ILLEGAL" value for the CHECKTYPE attribute is passed up to the root node whenever a type error occurs.

The global SYMBOLTABLE attribute is a list of all identifiers with their declared types. This attribute is inherited by all of the command statements in the programs, and is the reference for checking type compatibility.

4.6.3 Formal Definition

The basic functions used in the definition of mini-language 6 are

1) the Vienna operators UO, U, U1 and SEL.

2) SQ(α), where $\alpha \epsilon N$.

   $SQ(\alpha)=\alpha^2$

3) addition, $\alpha+\beta$ where α and $\beta \epsilon N$. Addition has its usual interpretation.

Following the formal definition, the most significant attribute equations will be discussed.

MINI-LANGUAGE 6

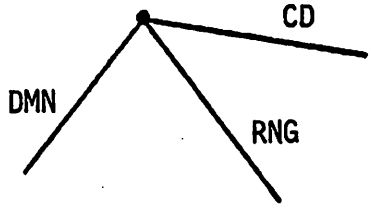
SYNTACTIC AND SEMANTIC DESCRIPTION

```
<PROGRAM>A=<DEC LIST>B ';' <COMMAND LIST>C ';END'
        <FLOWCHART>A↑F=COND(EQU('A↑C','LEGAL'),C↑F,'NUL')
        <PROCLIB>A↑P=U(B↑P,UO('SQ',MKFLC(0,'SQ(SEL(1,S))',)))
        <CHECKTYPE>A↑C=COND(EQU('B↑C','LEGAL'),'C↑C','ILLEGAL')
        C↑N='NUL';C↑S=B↑S

<TYPE DESIGNATION>A='N';B='(' <TYPE LIST>C '→' <TYPE DESIGNATION>D ')'
        <CODE>A↑C='N';B↑C='(C↑C→D↑C)'
        <TYPE>A↑T=U(UO('RNG','N'),UO('CD','N'));
        .B↑T=U(UO('DMN',C↑L),UO('RNG','D↑C'),UO('CD','B↑C'))
        C↑I=1

<TYPE LIST>A=<TYPE DESIGNATION>B ',' <TYPE LIST>C;D=<TYPE DESIGNATION>E
        <INDEX>I
        <CODE>A↑C=SEL('CD',B↑T) ',' SEL('CD',C↑L);D C=SEL('CD',E↑T)
        <LIST>A↑L=U(UO('CD','A↑C'),UO(A↑I,B↑T),C↑L);
        .D L=U(UO('CD',SEL('CD',E↑T)),UO(D↑I,E↑T))
        C↑I=A↑I + 1

<DECLARATIONA>A='DEC ' <VARLIST>B ' TYPE ' <TYPE DESIGNATION>C
        <SYMBOLTABLE>A↑S=B↑S
        B↑T=C↑T

<VARLIST>A=<$ALPHA>B ',' <VARLIST>C;D=<$ALPHA>E
        <TYPE>T
        <SYMBOLTABLE>A↑S=U(UO('B↑VAL',U(UO('TYPE', A↑T ),UO('KIND','VAR'))),
        .C↑S);
        .D↑S=UO('E↑VAL',U(UO('TYPE', D↑T ),UO('KIND','VAR')))
        C↑T=A↑T

<DECLARATIONB>A='DEC ' <$ALPHA>B '(' <FPLIST>C ')=' <EXPRESSION>D ' WHERE '
. <TYPE LIST>E
      <SYMBOLTABLE>A↑S=UO('B↑VAL',U(UO('TYPE',U(UO('DMN',E↑L),UO('RNG','D↑T'),
        .UO('CD','(E↑C→D↑T)'))),
        .UO('KIND','NAME')))
        <PROCLIB>A↑P=UO('B↑VAL',UO(0,U(UO('FUN','D↑F'),UO('NXT',NUL)))).
        <CHECKTYPE>A↑C=D↑C
        C↑I=1;C↑T=E↑L
        D↑S=C↑S
        E↑I='1'

<FPLIST>A=<$ALPHA>B ',' <FPLIST>C;D=<$ALPHA>E
        <INDEX>I
        <TYPELIST>T
        <SYMBOLTABLE>A↑S=U(UO('B VAL',U(U(UO('NO',A↑I),UO('KIND','PARAM')),
        .UO('TYPE',SEL(A↑I,A↑T)))),C↑S);
        .D↑S=UO('E↑VAL',U(U(UO('NO',D↑I),UO('KIND','PARAM')),UO('TYPE',SEL(D↑I,
        .D↑T))))
        C↑I=A↑I + 1;C↑T=A↑T

<DECLARATION>A=<DECLARATIONA>B;C=<DECLARATIONB>D
        <SYMBOLTABLE>A↑S=B↑S;C↑S=D↑S
        <PROCLIB>A↑P='NUL';C↑P=D↑P
        <CHECKTYPE>A↑C='LEGAL';C↑C=D↑C
```

```
<DEC LIST>A=<DECLARATION>B ';' <DEC LIST>C;D=<DECLARATION>E
        <SYMBOLTABLE>A↑S=U(B↑S,C↑S);D↑S=E↑S
        <PROCLIB>A↑P=U(B↑P,C↑P);D↑P=E↑P
        <CHECKTYPE>A↑C=COND(EQU('B↑C','LEGAL'),'C↑C','ILLEGAL');
        .D↑C=E↑C

<PRIMITIVE OBJECT>A=<$NUMBR>B;C='SQ';D='+'
        <VALUE>A↑V=B↑VAL;C↑V='"SQ"';D↑V='"ADD"'
        <TYPE>A↑T='N';C↑T='(N-N)';D↑T='(N,N-N)'

<IDENTIFIER>A=<$ALPHA>B
        <SYMBOLTABLE>S
        <FUNC>A↑F=COND(EQU(SEL('KIND',SEL('B↑VAL',A↑S)),'VAR'),'SEL("B↑VAL",S)'
        .,
        .COND(EQU(SEL('KIND',SEL('B↑VAL',A↑S)),'NAME'),'"B↑VAL"',
        .'SEL(' SEL('NO',SEL('B↑VAL',A↑S)) ',S)'))
        <TYPE>A↑T=SEL('TYPE',SEL('B↑VAL',A↑S))
        <VALUE>A↑V='B↑VAL'

<EXPRESSION>A=<PRIMITIVE OBJECT>B;
.C=<IDENTIFIER>D;
.E='SQ(' <EXPRESSION>F ')';
.G='(' <EXPRESSION>H '+' <EXPRESSION>I ')';
 .J=<IDENTIFIER>K '(' <ARGLIST>L ')';
.M='(IF ' <EXPRESSION>N '>' <EXPRESSION>O ' THEN ' <EXPRESSION>P ' ELSE '
. <EXPRESSION>Q ')'
         <SYMBOLTABLE>S
         <TYPE>A↑T=B↑T;
        .C↑T=SEL('CD',D↑T);
         .E↑T='N';
         .G↑T='N';
        .J↑T=SEL('RNG',K↑T);
         .M↑T=P↑T
        <FUNCTION>A↑F=B↑V;
        .C↑F=D↑F;
        .E↑F='SQ(F↑F)';
        .G↑F='(H↑F + I↑F)';
        .J↑F='APPLI(SEL(K↑F,PROCLIB),U(S,L↑F))';
        .M↑F='SELECT(N↑F,O↑F,P↑F,Q↑F)'
        <CHECKTYPE>A↑C='LEGAL';
        .C↑C='LEGAL';
        .E↑C=COND(EQU('F↑T F↑C','N LEGAL'),'LEGAL','ILLEGAL');
        .G↑C=COND(EQU('H↑T H↑C I↑T I↑C','N LEGAL N LEGAL'),'LEGAL','ILLEGAL');
        .J↑C=L↑C;
        .M↑C=COND(EQU('N↑T N↑C O↑T O↑C P↑C Q↑C','N LEGAL N LEGAL LEGAL LEGAL'),
        .COND(EQU('P↑T','Q↑T'),'LEGAL','ILLEGAL'),'ILLEGAL')
        D↑S=C↑S
        F↑S=E↑S
        H↑S=G↑S
        I↑S=G↑S
        K↑S=J↑S
        L↑S=J↑S;L↑I=1;L↑A=SEL('DMN',SEL('TYPE',SEL('K↑V',J↑S)))
        N↑S=M↑S
        O↑S=M↑S
        P↑S=M↑S
        Q↑S=M↑S

<ARGLIST>A=<EXPRESSION>B ',' <ARGLIST>C;D=<EXPRESSION>E
        <SYMBOLTABLE>S
        <INDEX>I
```

```
        <ARGTYPES>A
        <FUNCTION>A↑F='U(UO(A↑I,B↑F),C↑F)';D↑F='UO(D↑I,E↑F)'
        <CHECKTYPE>A↑C=COND(EQU('B↑C C↑C','LEGAL LEGAL'),
        .COND(EQU('B↑T',SEL('CD',SEL(A↑I,A↑A))),'LEGAL','ILLEGAL'),'ILLEGAL');
        .D C=COND(EQU('E↑T',SEL('CD',SEL(D↑I,D↑A))),'E↑C','ILLEGAL')
        B↑S=A↑S
        C↑S=A↑S;C↑I=A↑I + 1;C↑A=A↑A
        E↑S=D↑S

<ASSIGNMENT COMMAND>A=<$ALPHA>B '=' <EXPRESSION>C
        <NEXT>N
        <SYMBOLTABLE>S
        <LINENO>A↑L=UNO()
        <FLOWCHART>A↑F=UO(A↑L,U(UO('FUN','UI(S,"B↑VAL",C↑F)'),UO('NXT',A↑N)))
        <CHECKTYPE>A↑C=COND(EQU(SEL('CD',SEL('TYPE',SEL('B↑VAL',A↑S))),
        .'C↑T'),'C↑C' ,'ILLEGAL')
        C↑S=A↑S

<COMMAND LIST>A=<ASSIGNMENT COMMAND>B ';' <COMMAND LIST>C;
.D=<ASSIGNMENT COMMAND>E
        <NEXT>N
        <SYMBOLTABLE>S
        <LINENO>A↑L=B↑L;D↑L=E↑L
        <FLOWCHART>A↑F=U(B↑F,C↑F);D↑F=E↑F
        <CHECKTYPE>A↑C=COND(EQU('B↑C','LEGAL'),'C↑C','ILLEGAL');D↑C=E↑C
        B↑N=C↑L;B↑S=A↑S
        C↑N=A↑N;C↑S=A↑S
        E↑N=D↑N;E↑S=D↑S
END
```

TYPE's are represented by Vienna objects of the form

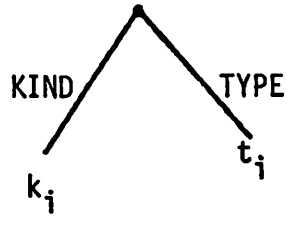


where the CD component is the string representing the type, and the DMN and RNG components are the types of the domain and range, respectively. Types are created by the rule

```
<TYPE DESIGNATION>A='N';B='(' <TYPE LIST>C '-' <TYPE DESIGNATION>D ')'
:
        <TYPE>A↑T=U(UO('RNG','N'),UO('CD','N'));
        .B↑T=U(UO('DMN',C↑L),UO('RNG','D↑C'),UO('CD','B↑C'))
```

- - - - - - - -

The SYMBOLTABLE has an entry for each identifier in the program, and the component selected by an identifier i is a VO of the form



where t_i is the type of the object associated with i, and

k_i=("NAME" if i is a function name
("PARAM" if i is a formal parameter
("VAR" if i is a simple variable

The SYMBOLTABLE is passed to practically every node in the parse tree.

- - - - - - - -

Each expression has a type given by the following rule:

```
<EXPRESSION>A=<PRIMITIVE OBJECT>B;
.C=<IDENTIFIER>D;
.E='SQ(' <EXPRESSION>F ')';
.G='(' <EXPRESSION>H '+' <EXPRESSION>I ')';
 .J=<IDENTIFIER>K '(' <ARGLIST>L ')';
.M='(IF ' <EXPRESSION>N '>' <EXPRESSION>O ' THEN ' <EXPRESSION>P ' ELSE '
. <EXPRESSION>Q ')'
:
:
:
```

```
<TYPE>A↑T=B↑T;
.C↑T=SEL('CD',D↑T);
 .E↑T='N';
 .G↑T='N';
.J↑T=SEL('RNG',K↑T);
 .M↑T=P↑T
```

Corresponding to the same syntactic rule, the CHECKTYPE attribute is defined by

```
<CHECKTYPE>A↑C='LEGAL';
.C↑C='LEGAL';
.E↑C=COND(EQU('F↑T F↑C','N LEGAL'),'LEGAL','ILLEGAL');
.G↑C=COND(EQU('H↑T H↑C I↑T I↑C','N LEGAL N LEGAL'),'LEGAL','
                                                    ILLEGAL');
.J↑C=L↑C;
.M↑C=COND(EQU('N↑T N↑C O↑T O↑C P↑C Q↑C','N LEGAL N LEGAL  LEGAL
                                                        LEGAL'),
.COND(EQU('P↑T','Q↑T'),'LEGAL','ILLEGAL'),'ILLEGAL')
```

- - - - - - - -

CHECKTYPE is also defined for each actual argument passed to a procedure. The rule checking that arguments are of the correct type is

```
<ARGLIST>A=<EXPRESSION>B ',' <ARGLIST>C;D=<EXPRESSION>E
:
     <CHECKTYPE>A↑C=COND(EQU('B↑C C↑C','LEGAL LEGAL'),
     .COND(EQU('B↑T',SEL('CD',SEL(A↑I,A↑A))),'LEGAL','ILLEGAL'),'ILLEGAL');
     .D↑C=COND(EQU('E↑T',SEL('CD',SEL(D↑I,D↑A))),'E↑C','ILLEGAL')
```

- - - - - - - -

CHECKTYPE is also defined when an assignment to a variable is made; the rule checking for type compatibility in an assignment statement is

```
<ASSIGNMENT COMMAND>A=<$ALPHA>B '=' <EXPRESSION>C
:
        <CHECKTYPE>A↑C=COND(EQU(SEL('CD',SEL('TYPE',SEL('B↑VAL',A↑S))),
        .'C↑T'),'C↑C'  ,'ILLEGAL')
```

- - - - - - - -

Whenever a CHECKTYPE attribute has the value "ILLEGAL", this value is passed up the parse tree and becomes the value of the CHECKTYPE attribute for the root node. For example, the following rule defines the value of CHECKTYPE for a command list to be "ILLEGAL" if any single command has the value ILLEGAL for CHECKTYPE:

```
<COMMAND LIST>A=<ASSIGNMENT COMMAND>B ';' <COMMAND LIST>C;
.D=<ASSIGNMENT COMMAND>E
:
    <CHECKTYPE>A↑C=COND(EQU('B↑C','LEGAL'),'C↑C','ILLEGAL');D↑C=E↑C
```

4.7 Mini-Language 7: Dynamic Type Checking

4.7.1 Ledgard's Description (Verbatim)

Description of Language Elements

Primitive objects in mini-language 7 are the same as in mini-language 6.

A declaration is a string of the form dec f(ℓ):=e, where f is an identifier, ℓ is a list of identifiers, and e is an expression (defined below) such that each identifier in ℓ must be different, and each identifier in e must occur in ℓ. A declaration of this form assigns to f the function from arguments whose domain includes any object into the value of the expression e obtained by replacing the identifiers in ℓ by their corresponding arguments. No single type is associated with an identifier denoting a function.

Expressions are of any of the following forms:

1) p
2) i
3) SQ(e)
4) (d+e)
5) $f(e_1,\ldots,e_n)$
6) (if d>e then e_1 else e_2),

where p is a primitive symbol, i and f are identifiers, and d, e, $e_1,\ldots,e_n$ are expressions.

Expressions and their types are evaluated as follows:

1) The value of a symbol p denoting a primitive object x is the primitive object x. The type of the value is the type of x.

2) The value of an identifier i is the object x currently assigned to i. The type of the value is the type of x.

3) If e is an expression whose current value is of type N, then the value of SQ(e) is the numerical square of the value of e, and the result is of type N.

4) If d and e are expressions whose current values are of type N, then the value of (d+e) is the numerical sum of the values of d and e, and the result is of type N.

5) If $e_1,\ldots,e_n$ are expressions whose current values are $v_1,\ldots,v_n$ and whose types are $t_1,\ldots,t_n$, then the type and value of $f(e_1,\ldots,e_n)$ are the type and value of the result of applying the value of f to $v_1,\ldots,v_n$.

6) If d and e are expressions whose values are of type N, then the value and type of (if d>e then e_1 else e_2) are either the value and type of e_1, if the numerical value of d is greater than the numerical value of e, or of e_2, if the numerical value of e is not greater than the numerical value of d.

A program (defined below) is in violation if, in the evaluation of an expression, any of the following conditions arises:

1) the value of the expression e in SQ(e) is not of type N;

2) the value of either expression d or e in (d+e) is not of type N;

3) the number n of arguments in an expression $f(e_1,\ldots,e_n)$ does not match the number of arguments in the declaration of the function associated with f; or

4) the value of either expression d or e in (if d>e then e_1 else e_2) is not of type N.

An assignment command consists of a string of the form i:=e, where i is an identifier, and e is an expression. The execution of an assignment command results in assigning the value of e to i.

Program Execution

A program consists of a sequence d of declarations followed by a sequence c of assignment commands such that each function identifier occurring in c is declared once and only once. The conventional meaning for execution of programs is assumed.

EXAMPLE 1:

```
dec F(X):=SQ(SQ(X))
A:=1
B:=2
C:=F
D:=C(B)
```

EXAMPLE 2:

```
dec F(P,X,Y):=P(X,Y)
dec G(X,Y)  :=(X+Y)
dec H(P,Q)  :=P(Q(2))
A:=F(G,1,2)
B:=F(H,SQ,SQ)
```

EXAMPLE 3:

```
dec F(X,Y):=(if Y>3 then (X+Y)
                    else X(Y))
A:=5
B:=6
C:=F(A,B)
A:=SQ
B:=2
D:=F(A,B)
```

EXAMPLE 4 (leads to a type violation):

```
dec F(X,Y):=(if Y>3 then (X+Y)
                    else X(Y))
A:=5
B:=6
C:=F(A,B)
A:=SQ
D:=F(A,B)
```

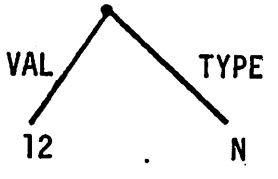
In Example 1, the final values of A, B, C, and D are, respectively: the natural number 1, the natural number 2, the function for computing X^4, and the natural number 16. In Example 2, the final values of A and B are the natural numbers 3 and 16, respectively.

In Example 3, the final values of A, B, C, and D are: the function for squaring a natural number, the natural number 2, the natural number 11, and the natural number 4, respectively. Note here that F is successively applied to objects of different types. Furthermore, the declaration of F could not be syntactically legal in mini-language 6 since no type specification for X and Y could lead to identical types for both (X+Y) and X(Y); i.e., X cannot be both a natural number and a function.

The final values of A, B, and C in Example 4 are: the function for squaring a natural number, the natural number 6, and the natural number 11, respectively. The final value of D is undefined since execution of the statement "D:=F(A,B)" leads to a type violation. The type violation occurs in the attempt to add the square function denoted by A to the number 6 denoted by B.

4.7.2 Discussion

Values manipulated by mini-language 7 are either integers or functions. As usual, functions will be represented by their names. Each value will be represented by a Vienna object which also includes type information. Specifically, each value will be represented by a VO having two components, a component selected by VAL and a component selected by TYPE. The evaluation of an expression or a function call in mini-language 7 also produces a value representation of this type. The VAL component is the value one would expect, and the TYPE component is the type of the VAL component, in the same notation used for expressing types in mini-language 6. For example, evaluation of the expression (5+7) in mini-language 7 produces



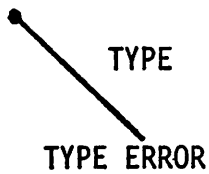
The above representation provides the information necessary for checking for type compatability. Types must be checked at the following times (corresponding to the four kinds of type errors):

1) Whenever SQ(e) is evaluated, e must be of type N.
2) Whenever an expression (d+e) is evaluated, both d and e must be of type N.
3) Whenever a function $f(e_1,\ldots,e_n)$ is called, the number of parameters in the parameter list must be identical to the

number of formal parameters in the declaration of f.*

4) When an expression of the form (if d>e then e_1 else e_2) is evaluated, both d and e must be of type N.

Undoubtedly, the easiest way of accomplishing the checks of types 1, 2, and 4 above would be to assume the existence of basic functions SQ', ADD', and SELECT' for mini-language 7, which take as arguments VO's having two components (VAL and TYPE) and return the desired result if there is no type conflict, or the VO



if a type conflict occurs. These functions could be used to define the corresponding operations in the language, and there would be little more to do. Essentially this was the approach taken, except, rather than assuming the existence of basic functions SQ', ADD', and SELECT', their definitions are incorporated into the definition of mini-language 7.

Functions are normally defined by expressions; however, by this time, the reader should be familiar with the concept of defining functions using recursive functional equations (equivalently, flowcharts). This is the way in which we will define ADD',

---

*This check could be made before execution; however, this did not dawn on the author before the formal definition was written, and hence it was done the hard way, dynamically. Since this exercise was somewhat pointless, it will not be discussed in this section any further.

SQ', and SELECT'. The explicit semantic equations constructing the recursion equations for the functions will be discussed in the next section. For example, the set of functionals defining SQ' is, letting t'=the type of the argument, and u'=the value of the argument,

F_0=if t'=N then F_1 else F_2

F_1=U(UO(TYPE N),UO(VAL,SQ(t'))

F_2=UO(TYPE,TYPE ERROR)

where SQ is a basic function defined for integers and returning the square of the value of the argument. The function definitions for SQ', ADD', and SELECT' are incorporated into the PROCLIB attribute of the root node of the program, which also includes flowcharts for each function defined by a function declaration. The state vector function for expressions of the form SQ(e), (d+e), (if d>e then f else g) are formed by applying the corresponding function SQ', ADD', SELECT' defined in PROCLIB. For example, the state vector function corresponding to SQ(e) will be

APPLI(SEL(SQ,PROCLIB),f_e)

where f_e is the state vector function corresponding to $\underline{e}$.

4.7.3 Formal Definition

The basic functions used in defining mini-language 7 are the same ones used in defining mini-language 6 (see Section 4.6.3).

Following the formal definition, the most significant attribute equations will be discussed.

MINI-LANGUAGE 7

SYNTACTIC AND SEMANTIC DESCRIPTION

```
<PROGRAM>A=<DEC LIST>B ';' <COMMAND LIST>C ';END'
        <FLOWCHART>A↑F=C↑F
        <PROCLIB>A↑P=U(B↑P,
        .U(
        .MAKESUB('SQ',1,'EQU(SEL(TYPEL,SEL(1,S)),"N")','U(UO(TYPEL,"N"),
        .UO(VALL,SEL(VALL,SEL(1,S)) * SEL(VALL,SEL(1,S))))')
        .,U(
        .MAKESUB('ADD',2,'AND(EQU(SEL(TYPEL,SEL(1,S)),"N"),EQU(SEL(TYPEL,
        .SEL(2,S)),"N"))','U(UO(TYPEL,"N"),UO(VALL,SEL(VALL,SEL(1,S)) + SEL(VAL
        .L,
        .SEL(2,S)))))')
        .,U(
        .MAKESUB('SELECT',4,'AND(EQU(SEL(TYPEL,SEL(1,S)),"N"),
        .EQU(SEL(TYPEL,SEL(2,S)),"N"))',
        .'SELECT(SEL(VALL,SEL(1,S)),SEL(VALL,SEL(2,S)),SEL(3,S),SEL(4,S))'),
        .MAKESUB('APPLI',2,'EQU(SEL("ARGNO",SEL(SEL(VALL,SEL(1,S)),PROCLIB)),
        .SEL("NO",SEL(2,S)))',
        .'APPLI(SEL("FLOWCHART",SEL(SEL(VALL,SEL(1,S)),PROCLIB)),SEL(2,S))')
        .))))
        C↑N='NUL';C↑S=B↑S

<DECLARATION>A='DEC ' <$ALPHA>B '(' <FPLIST>C ')=' <EXPRESSION>D
        <SYMBOLTABLE>A↑S=UO('B↑VAL',UO('KIND','NAME'))
        <PROCLIB>A↑P=UO('B↑VAL',U(UO('ARGNO',C↑N),UO('FLOWCHART',
        .UO(0,U(UO('FUN','D↑F'),UO('NXT',NUL))))))
        C↑I=1
        D↑S=C↑S

<DEC LIST>A=<DECLARATION>B ';' <DEC LIST>C;D=<DECLARATION>E
        <PROCLIB>A↑P=U(B↑P,C↑P);D↑P=E↑P
        <SYMBOLTABLE>A↑S=U(B↑S,C↑S);D↑S=E↑S

<FPLIST>A=<$ALPHA>B ',' <FPLIST>C;D=<$ALPHA>E
        <INDEX>I
        <NO>A↑N=1 + C N;D↑N=1
        <SYMBOLTABLE>A↑S=U(UO('B↑VAL',U(UO('NO',A↑I),UO('KIND','PARAM'))),C↑S);
        .D S=UO('E↑VAL',U(UO('NO',D↑I),UO('KIND','PARAM')))
        C↑I=A↑I + 1

<IDENTIFIER>A=<$ALPHA>B
        <SYMBOLTABLE>S
        <VALUE>A↑V='B↑VAL'
        <FUNCTION>A↑F=COND(EQU(SEL('KIND',SEL('B↑VAL',A↑S)),'NAME'),
        .'U(UO(VALL,"B↑VAL"),UO(TYPEL,"NAME"))',
        .COND(EQU(SEL('KIND',SEL('B↑VAL',A↑S)),'PARAM'),
        .'SEL(' SEL('NO',SEL('B↑VAL',A↑S)) ',S)',
        .'SEL("B↑VAL",S)'))

<ASSIGNMENT COMMAND>A=<$ALPHA>B '=' <EXPRESSION>C
        <NEXT>N
        <SYMBOLTABLE>S
        <LINEND>A↑L=UNO()
        <FLOWCHART>A↑F=UO(A↑L,U(UO('FUN','U1(S,"B↑VAL",C↑F)'),UO('NXT',A↑N)))
        C↑S=A↑S
```

```
<COMMAND LIST>A=<ASSIGNMENT COMMAND>B ';' <COMMAND LIST>C;
.D=<ASSIGNMENT COMMAND>E
          <NEXT>N
          <SYMBOLTABLE>S
          <LINENO>A↑L=B↑L;D↑L=E↑L
          <FLOWCHART>A↑F=U(B↑F,C↑F);D↑F=E↑F
          B↑N=C↑L;B↑S=A↑S
          C↑N=A↑N;C↑S=A↑S
          E↑N=D↑N;E↑S=D↑S

<PRIMITIVE OBJECT>A=<$NUMBR>B;C='SQ';D='+'
          <VALUE>A↑V=B↑VAL;C↑V='"SQ"';D↑V='"ADD"'
          <TYPE>A↑T='N';C↑T='(N_N)';D↑T='(N,N_N)'

<EXPRESSION>A=<PRIMITIVE OBJECT>B;
.C=<IDENTIFIER>D;
.E='SQ(' <EXPRESSION>F ')';
.G='(' <EXPRESSION>H '+' <EXPRESSION>I ')';
.J=<IDENTIFIER>K '(' <ARGLIST>L ')';
.M='(IF ' <EXPRESSION>N '>' <EXPRESSION>O ' THEN ' <EXPRESSION>P ' ELSE '
. <EXPRESSION>Q ')'
          <SYMBOLTABLE>S
          <FUNCTION>A↑F='U(UO(TYPEL,"B↑T"),UO(VALL,B↑V ))';
          .C↑F=D↑F;
          .E↑F='APPLI(SEL("FLOWCHART",SEL("SQ",PROCLIB)),UO(1,F↑F))';
          .G↑F='APPLI(SEL("FLOWCHART",SEL("ADD",PROCLIB)),U(UO(1,H↑F),UO(2,I↑F)))
          .';
          .J↑F='APPLI(SEL("FLOWCHART",SEL("APPLI",PROCLIB)),U(UO(1,K↑F),
          .UO(2,L↑F)))';
          .M↑F='APPLI(SEL("FLOWCHART",SEL("SELECT",PROCLIB)),
          .U(U(UO(1,N↑F),UO(2,O↑F)),U(UO(3,P↑F),UO(4,Q↑F))))'
          Q↑S=M↑S
          D↑S=C↑S
          F↑S=E↑S
          H↑S=G↑S
          I↑S=G↑S
          K↑S=J↑S
          L↑S=J↑S
          N↑S=M↑S
          O↑S=M↑S
          P↑S=M↑S

<ARGLIST>A=<EXPLIST>B
          <SYMBOLTABLE>S
          <FUNCTION>A F='U(UO("NO",B N),B F)'
          B↑S=A↑S;B↑I=1

<EXPLIST>A=<EXPRESSION>B ',' <EXPLIST>C;D=<EXPRESSION>E
          <SYMBOLTABTE>S
          <INDEX>I
          <NO>A↑N=1 + C↑N;D↑N=1
          <FUNCTION>A↑F='U(UO(A↑I,B↑F),C↑F)';D↑F='UO(D↑L,E↑F)'
          B↑S=A↑S
          C↑S=A↑S;C↑I=A↑I + 1
          E↑S=D↑S
END
```

The functions SQ', ADD', and SELECT' discussed in the preceding section are defined by the rule

```
<PROGRAM>A=<DEC LIST>B ';' <COMMAND LIST>C ';END'
:
        <PROCLIB>A↑P=U(B↑P,
        .U(
        .MAKESUB('SQ',1,'EQU(SEL(TYPEL,SEL(1,S)),"N")','U(UO(TYPEL,"N"),
        .UO(VALL,SEL(VALL,SEL(1,S)) * SEL(VALL,SEL(1,S))))')
        .,U(
        .MAKESUB('ADD',2,'AND(EQU(SEL(TYPEL,SEL(1,S)),"N"),EQU(SEL(TYPEL,
        .SEL(2,S)),"N"))','U(UO(TYPEL,"N"),UO(VALL,SEL(VALL,SEL(1,S)) + SEL
        .(VALL,
        .SEL(2,S))))')
        .,U(
        .MAKESUB('SELECT',4,'AND(EQU(SEL(TYPEL,SEL(1,S)),"N"),
        .EQU(SEL(TYPEL,SEL(2,S)),"N"))',
        .'SELECT(SEL(VALL,SEL(1,S)),SEL(VALL,SEL(2,S)),SEL(3,S),SEL(4,S))'),
        .MAKESUB('APPLI',2,'EQU(SEL("ARGNO",SEL(SEL(VALL,SEL(1,S)),PROCLIB)),
        .SEL("NO",SEL(2,S)))',
        .'APPLI(SEL("FLOWCHART",SEL(SEL(VALL,SEL(1,S)),PROCLIB)),SEL(2,S))')
```

The set of recursive equations defining each of these functions has the form

$$F_0 = \text{if } p \text{ then } F_1 \text{ else } F_2$$

$$F_1 = f_1$$

$$F_2 = UO(TYPE, TYPE\ ERROR)$$

where $\underline{p}$ is a predicate and f_1 corresponds to SQ, ADD, and SELECT. The Vienna expression defining such a set of equations is complex, and rather than writing it for each of the functions SQ', ADD', and SELECT', the function MAKESUB was defined by the expression, and used in its place. MAKESUB is found in lines 60700-61100 in Appendix II. Note that the use of MAKESUB is entirely a matter of convenience.

- - - - - - - -

State-vector functions corresponding to the operations of SQ, ADD, and SELECT in mini-language 7 are defined by the rule

```
<PROGRAM>A=<DEC LIST>B ';' <COMMAND LIST>C ';END'
        <FLOWCHART>A↑F=C↑F
        <PROCLIB>A↑P=U(B↑P,
        .U(
        .MAKESUB('SQ',1,'EQU(SEL(TYPEL,SEL(1,S)),"N")','U(UO(TYPEL,"N"),
        .UO(VALL,SEL(VALL,SEL(1,S)) * SEL(VALL,SEL(1,S))))')
        .,U(
        .MAKESUB('ADD',2,'AND(EQU(SEL(TYPEL,SEL(1,S)),"N"),EQU(SEL(TYPEL,
        .SEL(2,S)),"N"))','U(UO(TYPEL,"N"),UO(VALL,SEL(VALL,SEL(1,S)) + SEL(VAL
        .L,
        .SEL(2,S))))')
        .,U(
        .MAKESUB('SELECT',4,'AND(EQU(SEL(TYPEL,SEL(1,S)),"N"),
        .EQU(SEL(TYPEL,SEL(2,S)),"N"))',
        .'SELECT(SEL(VALL,SEL(1,S)),SEL(VALL,SEL(2,S)),SEL(3,S),SEL(4,S))'),
        .MAKESUB('APPLI',2,'EQU(SEL("ARGNO",SEL(SEL(VALL,SEL(1,S)),PROCLIB)),
        .SEL("NO",SEL(2,S)))',
        .'APPLI(SEL("FLOWCHART",SEL(SEL(VALL,SEL(1,S)),PROCLIB)),SEL(2,S))')
        .))))
```

In each case, the state vector function consists of applying the appropriate element of PROCLIB.

4.8 Mini-language 8: Structured Data

4.8.1 Ledgard's Description (Verbatim)

Description of Language Elements

Along with the natural numbers, the primitive objects of mini-language 8 include a class of objects called "pointers." The value of a pointer is the location at which another object is stored. In diagrams, a pointer is represented by a directed arrow terminating at the location of an object. Also included in the class of pointers is a special object, represented by the symbol A, which is interpreted as the "null" pointer - i.e., a pointer with no associated location.

A structure is either a primitive object or an n-tuple of objects ($n \geq 2$) each of which is, itself, a structure. A structure is represented in diagrams by a block, as shown in Figure 1, in which each element a_i in the block is either the representation of a primitive object, if the element denotes a primitive object; or a "dotted arrow" to another block, if the element denotes an n-tuple of objects or a subcomponent of the object being represented.

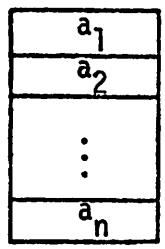


FIG. 1. Block representation of an n-tuple.

A structure identifier consists of a string of two or more capital English letters.

A component identifier consists of a string of two or more lower-case English letters.

A unit descriptor is a symbol whose value is a class of objects. "Primitive" unit descriptors are the symbols "num," "ptr," or "prim," which denote, respectively, the class of natural numbers, pointers, and primitive objects (natural numbers and pointers). A "named" unit descriptor is a structure identifier and denotes the class of objects defined in the declaration of the structure identifier (structure declarations are defined below). A unit descriptor is either a primitive unit descriptor or a named unit descrip-

tor.

A component descriptor is a string of the form

$$[c_1:u_1\ c_2:u_2\ \ldots\ c_n:u_n]$$

where the c_i, $1 \leq i \leq n$, are component identifiers, and u_i, $1 \leq i \leq n$, are unit descriptors. A component descriptor denotes the class of n-tuples of objects obtained from the Cartesian product of the sets denoted by $u_1, u_2, \ldots, u_n$.

A structure declaration is a string of either the form

dec s=d

or

dec $s=s_1d_1$ v s_2d_2 v ... v s_nd_n

where s, $s_1, \ldots, s_n$ are structure identifiers, and d, $d_1, \ldots, d_n$ are component descriptors. Structure declarations of either of these forms define s as representing members of the class of structures defined either by d or jointly by $d_1, d_2, \ldots, d_n$, and define each structure identifier s_i, $1 \leq i \leq n$, prefixing a component descriptor as representing the class of structures defined by the component descriptor d_i. For example, consider the following structure declarations.

1) dec INFO=[birthday:num|salary:num|period:num]

2) dec ACCOUNT=[iden:num|employee:INFO]

3) dec LIST=UNILIST[atom:prim] v PAIR[head:prim tail:LIST]

4) dec TREE=UNITREE[leaf:num] v BINTREE[node:num ℓb:TREE|rb:TREE]

The first declaration defines a set of triples each of which contains three natural numbers, such as the objects represented in Figure 2. The second declaration defines a set of pairs of which

the first element is a natural number (perhaps a social security number) and the second is an object defined in the first declaration, such as the objects represented in Figure 3. The third defines a class of objects called "lists," some of whose members are represented in Figure 4. The encircled object in Figure 4 is, strictly speaking, not part of the list, but represents one object to which the pointer might point. The fourth declaration defines a class of obfects called "trees," some of whose members are represented in Figure 5.

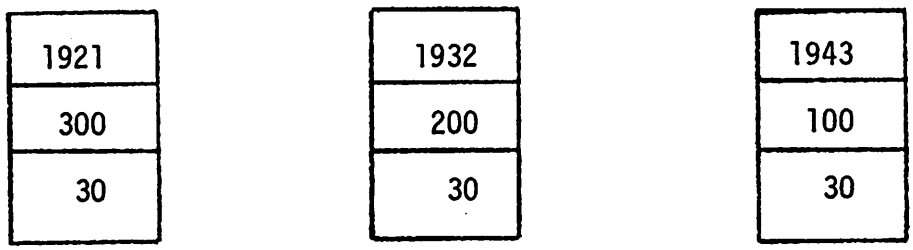


FIG. 2. Some objects defined by Declaration (1).

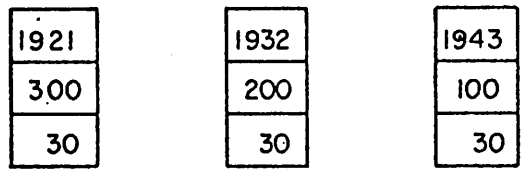


FIG. 3. Some objects defined by Declaration (2).

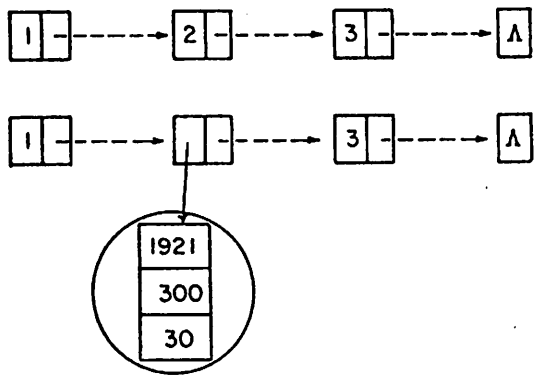


FIG. 4. Some objects defined by Declaration (3).

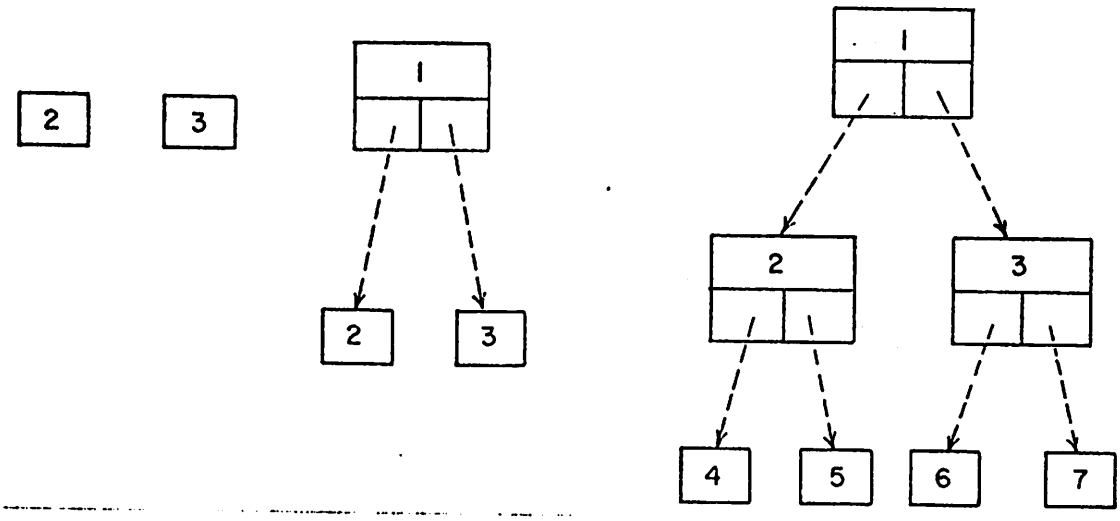


FIG. 5. Some objects defined by Declaration (4).

A constructor expression is a string of the form (cons s)(ℓ), where s is a structure identifier declared in a structure declaration, and ℓ is a list of expressions (an expression is defined below) whose value is a list of objects. The value of a constructor expression is an instance of the use of the objects of the list by the structure s. For example, using the structure declarations given previously, the evaluation of the following four constructor expressions results in the construction of the four objects of Figure 6:

1) (cons INFO)(1921,300,30)

2) (cons ACCOUNT)(022325795,(cons INFO)(1921,300,30))

3) (cons PAIR)(1,(cons PAIR)(2,(cons PAIR)(3,A)))

4) (cons BINTREE)(1,2,3)

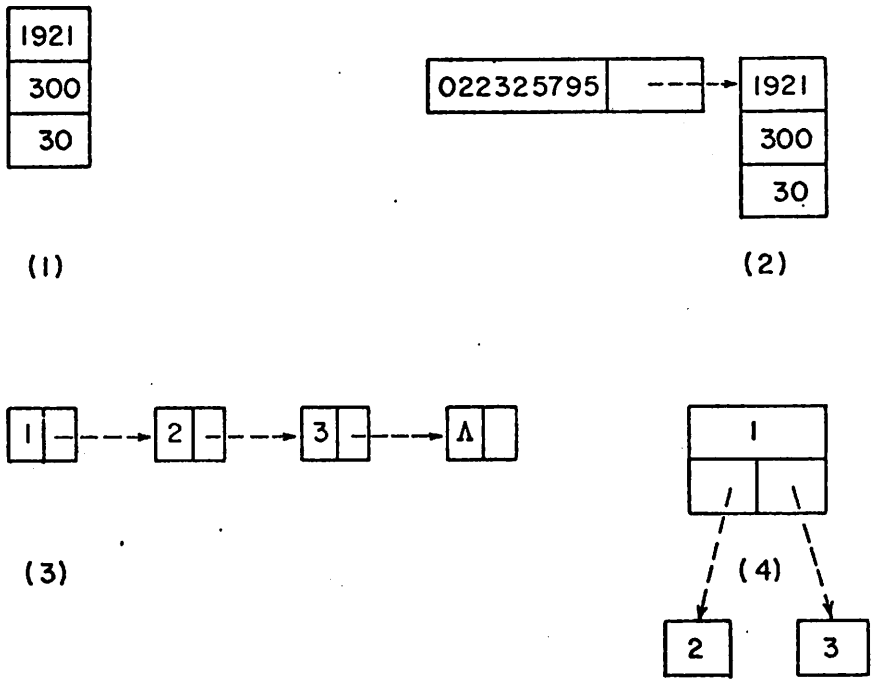


FIG. 6. Objects constructed by 4 constructor expressions.

In general, the value of a constructor expression (cons s)(ℓ) is obtained as follows:

1) The expression list ℓ is evaluated yielding a list of objects $\ell' = x_1, x_2, \ldots, x_n$.

2) Let d be the component descriptor

$$[c_i:u_i | c_2:u_2 | \ldots | c_n:u_n]$$

for s.

3) The value of (cons s)(ℓ) is the n-tuple of objects constructed from the x_i.

4) The case where more than one component descriptor is given for s or where the n-tuple constructed from the x_i is not a member of the set denoted by s is in violation.

A selector expression is a string of the form $c_i(e)$, where c_i is a component identifier, and e is an expression. If c_i is the i-th component identifier for a component descriptor d, and if the

value of e is an object x contained in the class represented by d, then the value of the selector expression $c_i(e)$ is the i-th component of x. The case where the value of e is not a member of the class represented by d is in violation.

A pointer expression is a string of the form ptr(e), where e is an expression. The value of a pointer expression is a pointer to the object defined by e. A value expression is a string of the form val(e), where e is an expression. If e is an expression whose value is a pointer to an object x, then the value of val(e) is x; if not, then the value of val(e) is in violation.

An expression is either an identifier, constructor expression, selector expression, pointer expression, or value expression.

An assignment statement consists of a string of either the form 1) $i:=e_1$ or 2) $c_i(e)=e_i$, where i is an identifier, c_i a component identifier, and e, e_1 are expressions. Let v, v_1 denote the values of e, e_1. The execution of an assignment statement of form (1) results in assigning a "copy" of v as the value of i. The execution of an assignment command of form (2) results in assigning a copy of v_1 as the value of the c_i component of the value of c_1, provided v_1 is an instance of the class denoted by c_i. If not, then the command is in violation.

A predicate is a string of the form (is (s)(e)), where s is a structure identifier, and e is an expression. The value of a predicate is the truth value "true" if the value of e is an instance of the class of objects represented by s. Otherwise the value is

"false".

A repeat statement is a string of the form

if p then c_1 else repeat after c_2

where p is a predicate, and c_1 and c_2 are assignment commands. The execution of a repeat command is as follows: if the value of p is "true," then c_1 is executed; otherwise c_2 is executed, and the repeat statement is, itself, executed again.

Program Execution

A program consists of a sequence s of structure declarations followed by a sequence t of assignment and repeat commands such that: each declared structure identifier is different; each structure identifier in the program is declared; each component identifier in s is different; and each component identifier in t occurs in s.

EXAMPLE 1:

```
dec LIST=UNILIST[atom:prim] v PAIR[hd:prim|tℓ:LIST]
dec TREE=UNITREE[leaf:num| v BINTREE[node:num|ℓb:TREE|rb:TREE]
A:=(cons BINTREE)(1,2,3)
B:=(cons BINTREE)(4,5,6)
C:=(consPAIR)(1,(cons PAIR)(ptr(A),3))
P:=ptr(C)
if(is BINTREE)(val(B))then
hd(val(P)):=ptr(B)
      else repeat after
      P:=ptr(tℓ(val(P))
```

EXAMPLE 2:

```
dec PERSON=[birthday:num|dad:ptr youngestkid:ptr]
A              :=(cons PERSON)(1000,A,A)
I              :=(cons PERSON)
                 (1930,ptr(A),A)
youngestkid(A):=ptr(I)
J              :=(cons PERSON)
                 (1932,ptr(A),A)
youngestkid(A):=ptr(J)
X              :=(cons PERSON)
                 (1955,ptr(I),A)
youngestkid(I):=ptr(X)
```

After execution of the first three commands in Example 1, the objects of Figure 7 are formed. Execution of the remaining statements in the program results in the conversion of these objects to those shown in Figure 8.

In Example 2, we wish to develop a structure that gives the date of birth of the decendants (sic) of a person named A and, for each offspring, the names of his father and youngest child. The first command represents the birth of A. The next four commands represent the successive births of children I and J to A, and the subsequent two commands represent the birth of child X to I. The structure of Figure 9 is obtained after execution of these commands.

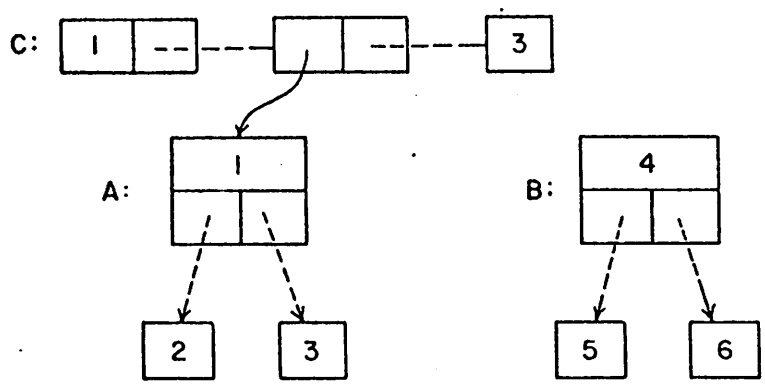


FIG. 7. Objects formed by the first 3 commands of Example 1.

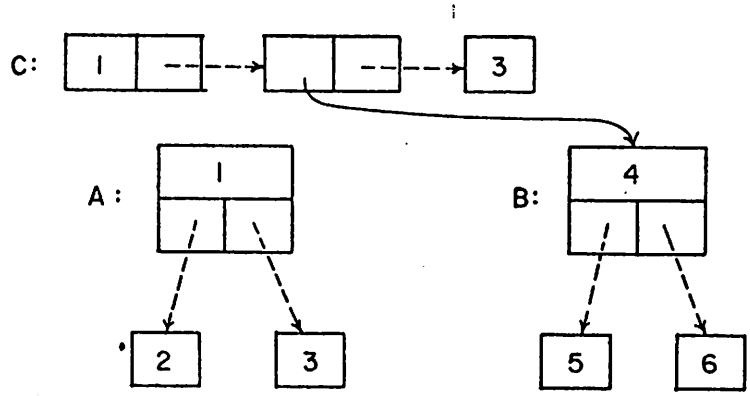


FIG. 8. Objects formed at the completion of Example 1.

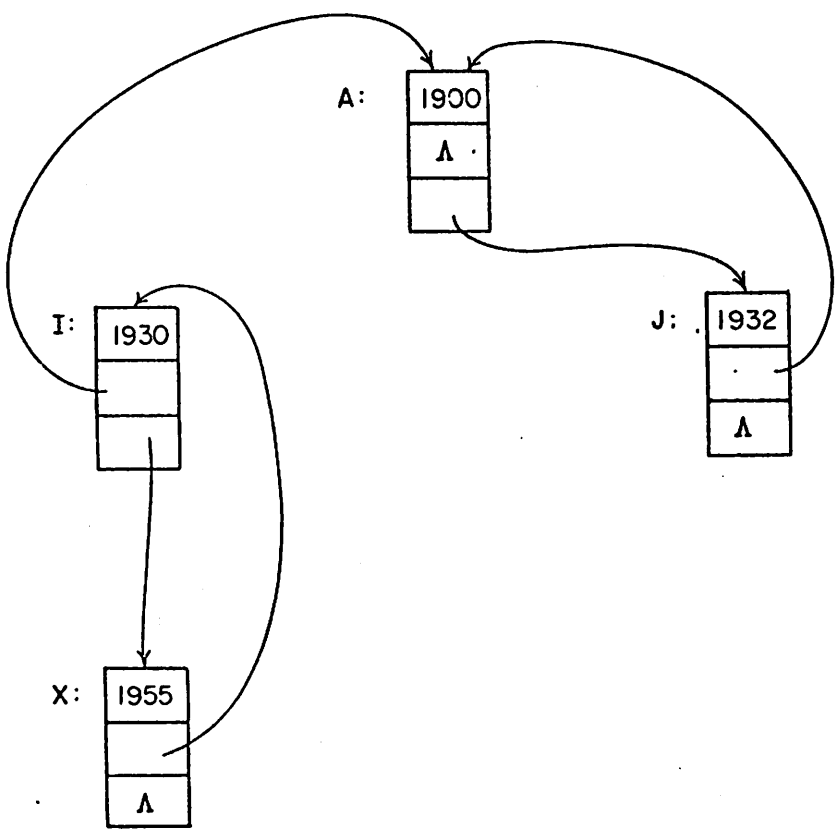


FIG. 9. Objects formed at the completion of Example 2(in mini-language 8).

4.8.1.a Notation

Only capital letters are used.

Angle brackets < > are used in place of square brackets [ ] in component descriptions.

Slashes (/) are used in place of vertical lines in component descriptions.

The null pointer is denoted by * rather than Λ.

A structure declaration is a string of either the form

DEC s=d

where s is a structure identifier and d, a component description or

DEC $s=s_1 \vee s_2 \vee \ldots \vee s_n$

where $s, s_1, \ldots, s_n$ are structure identifiers.

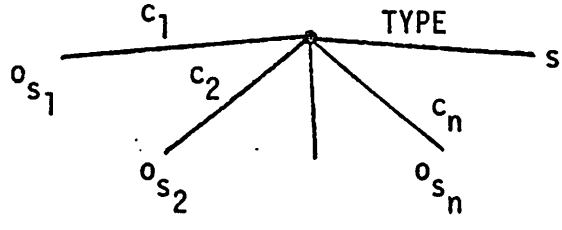
Declarations are required for simple variables.

4.8.2 Discussion

Clearly, the structured data objects of mini-language 8 can be faithfully represented by Vienna objects. For example, an object of the type defined by the structural declaration

DEC $s= c_1:s_1/c_2:s_2/\ldots/c_n:s_n$

will be represented by a VO of the form



where o_{s_i} is a representation of an object of type s_i. The TYPE

component is included in the representation so the type of any structured object can be determined easily.

The numerous operations in mini-language 8 can be translated into corresponding Vienna operators (or combinations of Vienna operators). Each expression will have an associated state vector function corresponding to evaluation of the expression on the left side of an equals sign (LHSFUN) and on the right side of an equals sign RHSFUN). Each of these will be discussed in detail in the next section. "Pointers" and "locations" are handled as in mini-language 2, and are discussed in Section 4.2.2.

For each structure identifier s, there is an entry in the IDLIST attribute of the root node of the program for s; if s is declared in a declaration of the form

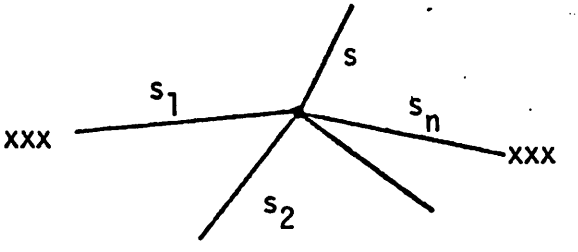
DEC s=d

where d is a component description, this entry has the form

If s is declared in a structure declaration of the form

DEC $s=s_1 \vee s_2 \vee \ldots \vee s_n$

then the entry for s has the form



This table is used in the evaluation of predicates; thus the predicate (IS s)(e) corresponds to the state vector predicate

$$ELEM(SEL(TYPE,f_e),SEL(s,IDLIST))$$

where f is the state vector function corresponding to the expression e, and ELEM is a basic predicate defined by

$$ELEM(s,o)=\underline{TRUE} \text{ if } SEL(s,o)\neq NULL$$

$$ELEM(s,o)=\underline{FALSE} \text{ if } SEL(s,o)=NULL$$

4.8.3 Formal Definition

The basic functions used in the definition of mini-language 8 are

1) the Vienna operators UO, U, U1, and SEL.

2) the predicate ELEM(α,β), where α is a string and β, a VO.

ELEM is defined by

$$ELEM(\alpha,\beta)=\begin{cases}\underline{TRUE} \text{ if } SEL(\alpha,\beta)\neq\eta\\ \underline{FALSE} \text{ otherwise}\end{cases}$$

Following the formal definition, the most significant attribute equations will be discussed.

MINI-LANGUAGE 8

SYNTACTIC AND SEMANTIC DESCRIPTION

```
<PROGRAM>A=<DECLARATION LIST>B ';' <COMMAND LIST>C ';END'
        <FLOWCHART>A F=U(B↑F,C↑F)
        <IDLIST>A I=B↑I
        C↑T=B↑T;C↑N='NUL'
        B↑N=C↑S

<DECLARATION LIST>A=<DECLARATION>B ';' <DECLARATION LIST>C;D=<DECLARATION>E
        <NEXT>N
        <STATENOO>A↑S=B↑S;D↑S=E↑S
        <IDLIST>A↑I=U(B↑I,C↑I);D↑I=E↑I
        <TYPETABLE>A↑T=U(B↑T,C↑T);D↑T=E↑T
        <FLOWCHART>A↑F=U(B↑F,C↑F);D↑F=E↑F
        B↑N=C↑S
        C↑N=A↑N
        E↑N=D↑N

<DECLARATION>A='DEC ' <VARLIST>B;
            .C='DEC ' <$IDENT>D '=' <COMPONENTS DESCRIPTOR>E;
            .F='DEC ' <$IDENT>G '=' <STRUCTURE LIST>H
        <NEXT>N
        <STATEND>A↑S=UNO();C↑S=C↑N;F↑S=F↑N
        <IDLIST>A↑I='NUL';C↑I=UO('D↑VAL',UO('D↑VAL','XXX'));F↑I=UO('G↑VAL',H↑L)
        <TYPETABLE>A↑T='NUL';C↑T=UO('D↑VAL',E↑S);F↑T='NUL'
        <FLOWCHART>A↑F=UO(A↑S,U(UO('FUN','B↑F'),UO('NEXT',A↑N)));C↑F='NUL';
        .F↑F='NUL'
        B↑I='1'

<COMPONENTS DESCRIPTOR>A='<' <COMPONENT LIST>B '>'
        <STRUCTURE>A↑S=B↑S
        B↑I='1'

<COMPONENT LIST>A=<COMPONENT DESC>B ' / ' <COMPONENT LIST>C;D=<COMPONENT DESC>E
        <INDEX>I
        <STRUCTURE>A↑S=U(B↑S,C↑S);D↑S=E↑S
        C↑I=A↑I + 1
        B↑I=A↑I
        E↑I=D↑I

<COMPONENT DESC>A=<$IDENT>B ':' <$IDENT>C
        <INDEX>I
        <STRUCTURE>A↑S=UO(A↑I,'B↑VAL')

<STRUCTURE LIST>A=<$IDENT>B ' & ' <STRUCTURE LIST>C;D=<$IDENT>E
        <LIST>A↑L=U(C↑L,UO('B↑VAL','XXX'));D L=UO('E↑VAL','XXX')

<VARLIST>A=<$IDENT>B ',' <VARLIST>C;D=<$IDENT>E
        <INDEX>I
        <FUNCTION>A↑F='U(C↑F,UO("B↑VAL",OMEGA))';D↑F='UO("E↑VAL",OMEGA)'
        C↑I=A I + 1

<COMMAND LIST>A=<COMMAND>B ';' <COMMAND LIST>C;D=<COMMAND>E
        <NEXT>N
        <TYPETABLE>T
        <STATENO>A↑S=B↑S;D↑S=E↑S
        <FLOWCHART>A↑F=U(B↑F,C↑F);D↑F=E↑F
```

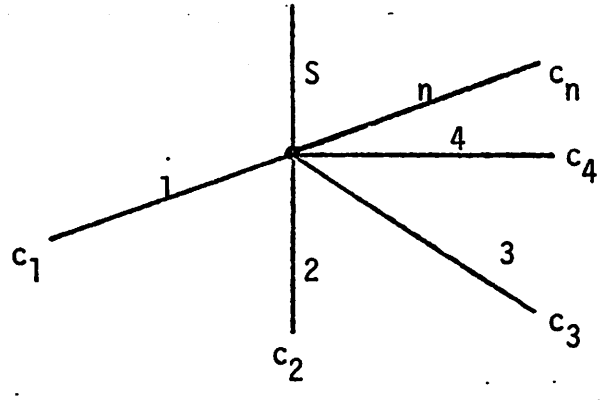
```
          B↑T=A↑T;B↑N=C↑S
          C↑T=A↑T;C↑N=A↑N
          E↑T=D↑T;E↑N=D↑N

<COMMAND>A=<ASSIGNMENT STATEMENT>B;C=<REPEAT STATEMENT>D
          <NEXT>N
          <TYPETABLE>T
          <STATENO>A↑S=B↑S;C↑S=D↑S
          <FLOWCHART>A↑F=B↑F;C↑F=D↑F
          B↑N=A↑N;B↑T=A↑T
          D↑N=C↑N;D↑T=C↑T

<ASSIGNMENT STATEMENT>A=<$IDENT>B '=' <EXPRESSION>C;D=<$IDENT>E
. '(' <EXPRESSION>F ')=' <EXPRESSION>G
          <NEXT>N
          <TYPETABEE>T
          <STATENO>A↑S=UNO();D↑S=UNO()
          <FLOWCHART>A↑F=UO(A S,U(UO('FUN','U(S,UO("B↑VAL",C↑R))'),UO('NEXT',A↑N)
          .));
          .D↑F=UO(D↑S,U(UO('FUN','U1(S,COMPOS("E↑VAL",F↑L1,G↑R)')
          .,UO('NEXT',D↑N)))
          C↑T=A↑T
          F↑T=D↑T
          G↑T=D↑T

<REPEAT STATEMENT>A='IF ' <PREDICATE>B ' THEN ' <ASSIGNMENT STATEMENT>C
. ';ELSE REPEAT AFTER ' <ASSIGNMENT STATEMENT>D
          <NEXT>N
          <TYPETABLE>T
          <STATENO>A↑S=UNO()
          <FLOWCHART>A↑F=U(UO(A↑S,U(UO('FUN','ID(S)'),UO('NEXT','COND(B↑P,C↑S,D↑S
          .)'))),U(C↑F,D↑F))
          C↑T=A↑T;C↑N=A↑N
          D↑T=A↑T;D↑N=A↑S
          B↑T=A↑T

<PREDICATE>A='(IS ' <$IDENT>B ')(' <EXPRESSION>C ')'
          <TYPETABLE>T
          <PREDICATE>A↑P='ELEM(SEL(TYPEL,C↑R),SEL("B↑VAL",IDLIST))'
          C↑T=A↑T

<EXPRESSION>A=<$IDENT>B;
.C=<$NUMBR>D;
.E=<CONSTRUCTOR EXPRESSION>F;
.I=<POINTER EXPRESSION>J;
.K=<VALUE EXPRESSION>L;
.G=<SELECTOR EXPRESSION>H
          <TYPETABLE>T
          <LHSFUN>A↑L='"B↑VAL"';C↑L='NUL';
          .E↑L=F↑L;G↑L=H↑L;I↑L=J↑L;K↑L=L↑L
          <RHSFUN>A↑R='SEL("B↑VAL",S)';
          .C↑R=D↑VAL;
          .E↑R=F↑R;G↑R=H↑R;I↑R=J↑R;K↑R=L↑R
          F↑T=E↑T;H↑T=G↑T;J↑T=I↑T;L↑T=K↑T

<CONSTRUCTOR EXPRESSION>A='(CONS ' <$IDENT>B ')(' <CONSTRUCTOR LIST>C ')'
          <TYPETABLE>T
          <RHSFUN>A↑R='U(UO(TYPEL,"B↑VAL"),C↑F)'
          <LHSFUN>A↑L='SLX(UO(UNO(),A↑R))'
          C C=SEL('B↑VAL',A↑T)
```

```
          C↑I='1'
          C↑T=A↑T

<CONSTRUCTOR LIST>A=<EXPRESSION>B ',' <CONSTRUCTOR LIST>C;D=<EXPRESSION>E
          <TYPETABLE>T
          <COMPONENTLIST>C
          <INDEX>I
          <FUNCTION>A↑F='U(UO("' SEL(A↑I,A↑C) '",B↑R),C↑F)';
          .D↑F='UO("' SEL(D↑I,D↑C) '",E↑R)'
          B↑T=A↑T
          C↑T=A↑T;C↑C=A↑C;C↑I=A↑I + 1
          E↑T=D↑T

<SELECTOR EXPRESSION>A=<$IDENT>B '(' <EXPRESSION>C ')'
          <TYPETABLE>T
          <LHSFUN>A↑L='COMPOS("B↑VAL",C↑L)'
          <RHSFUN>A↑R='SEL("B↑VAL",C↑R)'
          C↑T=A↑T

<POINTER EXPRESSION>A='PTR(' <EXPRESSION>B ')';C='*'
          <TYPETABLE>T
          <LHSFUN>A↑L='SLX(UO(UNO(),A↑R))';C↑L='NUL'
          <RHSFUN>A↑R=B↑L;C↑R='"*"'
          B↑T=A↑T

<VALUE EXPRESSION>A='VAL(' <EXPRESSION>B ')'
          <TYPETABLE>T
          <LHSFUN>A↑L=B↑R
          <RHSFUN>A↑R='SEL(B↑R,S)'
          B↑T=A↑T
END
```

Each structure declaration of the form

DEC s=d,

where d is a components descriptor of the form <c_1:t_1/c_2:t_2/.../c_n:t_n> creates an entry in the TYPETABLE of the form



The components of the form <n:c_n> originate in the rule

```
<COMPONENT DESC>A=<$IDENT>B ':' <$IDENT>C
⋮
        <STRUCTURE>A↑S=UO(A↑I,'B↑VAL')
```

- - - - - - - - -

In a constructor expression of the form (CONS s)(ℓ), the component selected by s in the typetable is the list of selectors for a structure of type s. This list is passed to the constructor list to create the state vector function corresponding to the constructor expression by the rule

```
<CONSTRUCTOR EXPRESSION>A='(CONS ' <$IDENT>B ')(' <CONSTRUCTOR LIST>C ')'
⋮
        C↑C=SEL('B↑VAL',A↑T)
```

The FUNCTION corresponding to a constructor list is defined by the rule

```
<CONSTRUCTOR LIST>A=<EXPRESSION>B ',' <CONSTRUCTOR LIST>C;D=<EXPRESSION>E
:
          <FUNCTION>A↑F='U(UO("' SEL(A↑C) '",B↑R),C↑F)';
          .D↑F='UO("' SEL(D↑I,D↑C) '",E↑R)'
```

Note that SEL(A↑I,A↑C) and SEL(D↑I,D↑C) refer to the apprporiate component names.

- - - - - - - -

The selector function in mini-language 8 is represented by the Vienna operator SEL by the rule

```
<SELECTOR EXPRESSION>A=<$IDENT>B '(' <EXPRESSION>C ')'
:
       RHSFUN A↑R='SEL("B↑VAL",C↑R)'
```

- - - - - - - -

The flowchart corresponding to a "repeat" statement of the form a=IF β THEN c ELSE REPEAT AFTER d is given by the rule

```
<REPEAT STATEMENT>A='IF ' <PREDICATE>B ' THEN ' <ASSIGNMENT STATEMENT>C
. ';ELSE REPEAT AFTER ' <ASSIGNMENT STATEMENT>D
:
          <FLOWCHART>A↑F=U(UO(A↑S,U(UO('FUN','ID(S)'),UO('NEXT','COND(B↑P,
          .C↑S,D↑S)'))),U(C↑F,D↑F))
```

The statement following d is identical to the statement following a; hence, we have

```
          ...C↑N=A↑N
```

and after d is executed, the predicate should be tried again; hence, we have

```
          ...D↑N=A↑S.
```

The recursive functional corresponding to the flowchart has the form

$$F_a = \text{if} \quad \text{then } F_e \circ F_c \text{ else } F_a \circ F_d$$

where e is the statement following a.

4.9 Mini-Language 9: String Manipulation

4.9.1 Ledgard's Description (Verbatim)

Description of Language Elements

The object alphabet comprises the capital English letters, the five characters "(", ")", "+", "*", and ",", as well as a null string (i.e., a string containing no characters at all). The null string is denoted by the special symbol "Λ".

The variables in mini-language 9 consist of the symbols a b ... z. Each variable is defined as representing a set of strings of object alphabet symbols (see variable definitions given below). Initially, each variable in mini-language 9 represents the empty set { }, i.e., the set containing no strings at all, not even the null string.

A pattern is a string of object alphabet symbols and variables. A pattern p represents the set of strings computed by concatenating, in order, from left to right, each of the object alphabet symbols in p with any string represented by a variable in p, with the requirement that multiple occurrences of the same variable in a pattern must denote the same object string. For example, if ℓ is a variable denoting the set of English letters, then the pattern "ℓMℓ" denotes the set AMA,BMB,...,ZMZ . Furthermore, if any variable in p represents the empty set, then the pattern p represents

the empty set. For example, if s is a variable denoting the empty set, then the pattern "AsA" denotes the empty set.

Variable definitions are sequences of rules of the form

$$v=p_1|p_2|\ldots|p_n$$

where v is a variable, and p_i, $1 \leq i \leq n$, are patterns. The variable definitions jointly define each variable on the left side of "=" as representing a set containing the union of sets denoted by each pattern given on the right side of "=". For example, the variable definitions:

1) $\ell=A|B|\ldots|Z$
   $x=\ell$
   $s=x|sx$

2) $\ell=A|B|\ldots|Z$
   $x=s$
   $s=x|sx$

3) $\ell=A|B|\ldots|Z$
   $x=\Lambda$
   $s=x|sx$

define the variables ℓ, x, and s as representing the following sets:

1) $\ell=\{A,B,\ldots,Z\}$
   $x=\{A,B,\ldots,Z\}$
   $s=\{A,AA,AB,AC,\ldots\}$

2) $\ell=\{A,B,\ldots,Z\}$
   $x=\{\ \}$
   $s=\{\ \}$

3) $\ell=\{A,B,\ldots,Z\}$

$x=\{\Lambda\}$

$s=\{\Lambda\}$

A transformation definition is a construct of the form

$$\text{let } x=\begin{bmatrix} p_1\rightarrow(\cdot)s_1 \\ p_2\rightarrow(\cdot)s_2 \\ \vdots \\ p_n\rightarrow(\cdot)s_n \end{bmatrix}$$

where $\underline{x}$ is an identifier, p_i, $1\leq i\leq n$, are patterns, and s_i, $1\leq i\leq n$, are strings of object alphabet symbols and variables such that each variable in s_i also occurs in p_i, and "$(\cdot)$" indicates the possible occurrence of a "$\cdot$" after "$\rightarrow$".

A pattern p_i is said to occur within an object string $\underline{Q}$ if one or more of the strings represented by p_i occurs within $\underline{Q}$. The "left-most and shortest" occurrence of p_i in $\underline{Q}$ is the string (of the occurrences of p_i in $\underline{Q}$) such that the occurrence begins with the leftmost object symbol and is as short as possible.

A string transformation $\underline{X}$ of the above form, when applied to an object string $\underline{Q}$, is taken to mean:

1) Look down among the rules of $\underline{X}$ for the first rule in which p_i occurs in $\underline{Q}$.
2) If such a rule is found, replace the left-most and shortest occurrence of p_i in $\underline{Q}$ with the string obtained from s_i by replacing each variable $\underline{v}$ in s_i with the string used for $\underline{v}$ in p_i. If a "$\cdot$" occurs after the "$\rightarrow$" in the substitution rule, terminate the

algorithm. Otherwise, repeat the application to the newly formed string.

3) If no such rule is found, terminate the algorithm.

If the above algorithm terminates, the string obtained upon termination of the algorithm is the result of applying X to Q.*

For example, consider the variable definitions:

ℓ=A|B|...|Z

x=Λ

s=ℓ|ℓs

and the transformation definitions:

let A=[ℓA→•A]

let B=[ℓA→A]

let C=[xA→A]

let D=[sAs→A]

let E=$\begin{bmatrix} \ell s \rightarrow s \\ \ell \rightarrow \bullet \ell\ell\ell \end{bmatrix}$

then: A transforms YZAZY into YAZY; B transforms YZAZY into AZY; C transforms YZAZY into YZAZY repeatedly, but the algorithm does not terminate; D transforms YZAZY into A; and E transforms YZAZY into YYY.

A result expression is a string of the form $t_n<\ldots t_2<t_1<s>>\ldots>$, where s is a string of object alphabet symbols, and t_i, $1 \leq i \leq n$, are

---

*The string transformation algorithm is computable only if the variables represent recursive sets. The variable definitions allowed in mini-language 9 define only context-free sets, which are encompassed by recursive sets.

identifiers denoting string transformations.

Program Execution

A program consists of a sequence s of variable definitions, followed by a sequence t of transformation definitions and a result expression r such that each identifier in r occurs as the name of one and only one transformation definition in t. The result of executing a program with a result expression $t_n<...t_2<t_1<s>>...>$ is computed by successively applying the transformations $t_1, t_2, ..., t_n$ to an object string whose initial value is s.

EXAMPLE 1:

```
ℓ=A|B|...|Z
m=ℓ
let P= [ ℓm*→m*ℓ
       | (m*→m(
       | ( )→•Λ
       | )→*) ]
P<(NOXIN)>
```

EXAMPLE 2:

```
ℓ=A|B|...|Z
w=ℓ|ℓw
v=w
let Q= [ v,w*→w*,v
       | (w,*→w,(
       | (w*)→w
       | ( )→•Λ
       | )→*) ]
Q<(HESSE,KAFKA,MANN)>
```

EXAMPLE 3:

```
ℓ=A|B|...|Z
p=ℓ|(a)
a=p|p*a|p+a
x=ℓ|(|)|*|+
y=x
let R=[p*a→a
       p+a→a
       (a)→a
       xy→*
       a→•YES
       *→•NO]
R<I*(J+K)+J>
```

In Example 1, the string transformation P defines a function mapping a parenthesized string of letters into the string with the letters reversed; hence, the value of the result expression is the string "NIXON." In Example 2, the string transformation Q defines a function mapping a parenthesized list of words into the list with the words in reverse order; hence, the value of the result expression is the string "MANN,KAFKA,HESSE." The string transformation R in Example 3 defines a function mapping arbitrary strings into one of the two strings "YES" or "NO" depending on whether or not the input string is a well-formed arithmetic expression. Hence, the value of the result expression is "YES".

4.9.1.a Notational Changes

Only capital letters are used.

Literal strings are enclosed in dashes; thus x=ABC is represented by x=-ABC-.

Slashes (/) separate alternatives in a variable definition instead of vertical lines.

A transformation definition has the form

LET Tk=<<a_1;

a_2;

⋮

a_n>>

where k is an integer and $a_1, a_2, \ldots, a_n$ are actions.

The null string is represented by ξ rather than Λ.

In an action, the right arrow → is replaced by #.

4.9.2 Discussion

In mini-language 9, both variables and patterns represent sets of strings. Since these sets may be infinite, it is clear that in our semantic model, the sets cannot be represented by listing their elements. In order to get a finite representation of these potentially infinite objects, both variables and patterns will be represented by state vector predicates. Let p be a predicate associated with a variable or pattern; then

p(x)=TRUE if x is an element of the set represented by the variable or pattern

p(x)=FALSE otherwise

The predicate associated with each variable is entered into the VARIABLETABLE attribute of the PROGRAM nonterminal.

Suppose we have a variable definition of the form

A=-α-

that is, where the right hand side is a literal string. Then the

set corresponding to A is {α}, and the predicate corresponding to A is given by

$$P_A = -\alpha-(S) = IDEN(S,\alpha)$$

where IDEN returns TRUE if its arguments are identical and FALSE otherwise.

The next case is more complicated. Let p be a pattern of the form

$$p_1\ p_2\ \cdots\ p_n$$

where p_i is either a variable or a literal, i=1,...,n. Let s_i be the set associated with p_i; then the set associated with p is

$$s_1 \times s_2 \times \ldots \times s_n.$$

To construct the predicate associated with p from the predicates associated with each of the p_i will require some work. Let SUBSTR(i_1,i_2,s) be a function taking two integer arguments, i_1 and i_2, and one string argument s, and returning the substring consisting of the i_1^{th} character of s to the i_2^{th} character of s. A string s is in the set represented by p if and only if

$$\exists\ i_1,\ldots,i_{n-1} \qquad 0 \leq i_1 \leq i_2 \leq \ldots \leq i_{n-1} \leq N+1$$

such that

$$p_1(SUBSTR(0,i_1,s)) = TRUE$$

and $$p_2(SUBSTR(i_1,i_2,s)) = TRUE$$

$$\vdots$$

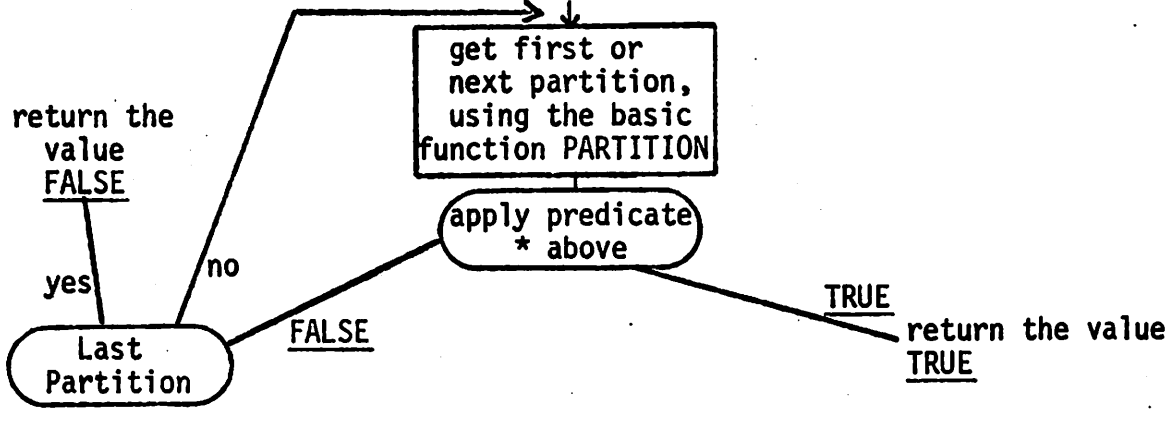
and $$p_n(SUBSTR(i_{N-2},i_{N-1},s)) = TRUE$$

this can be written

*AND(p_1(SUBSTR(0,i_1,s)),AND(p_2(SUBSTR(i_1,i_2,s)),...,AND(...,p_n(SUBSTR(i_{N-1},i_{N-2},s))...)

The problem is the "there exists" phrase preceding this expression. In defining state vector functions, only composition of known basic functions is allowed, and $\exists\ i_1,\ldots,i_n$ such that..., etc., does not represent the use of a basic function. This phrase can be eliminated from the definition of the predicate associated with p as follows:

Let PARTITION be a basic function which produces all the possible sets $i_1,\ldots,i_n$ such that $0\leq i_1\leq i_2\leq\ldots\leq i_{n-1}\leq N+1$. Using the function, the following flowchart can be defined (for a more detailed discussion, refer to the next section):



This flowchart defines the desired predicate for p. It is given a unique name and entered into the CONCATLIB attribute table; thus, every pattern in the program has a corresponding predicate defined in CONCATLIB.

Let A be defined by a variable definition of the form

$$A=\beta_1/\beta_2/\ldots/\beta_n$$

where $\beta_1,\ldots,\beta_n$ are patterns. Let s_i be the set of strings denoted by β_i for $i=1,\ldots,n$. Then the set of strings corresponding to A is

$$\bigcup_{i=1,n} s_i$$

and the natural predicate to associate with A is given by

$$**P_{A=p_1,\ldots,p_n}=OR(p_1,OR(p_2,\ldots,OR(p_{n-1},p_n)\ldots))$$

where OR is the usual "or" function defined by the truth table

	T	F
T	T	T
F	T	F

Unfortunately, this predicate does not work for the following reason: consider A=A'X'/'Y'. The associated predicate would be

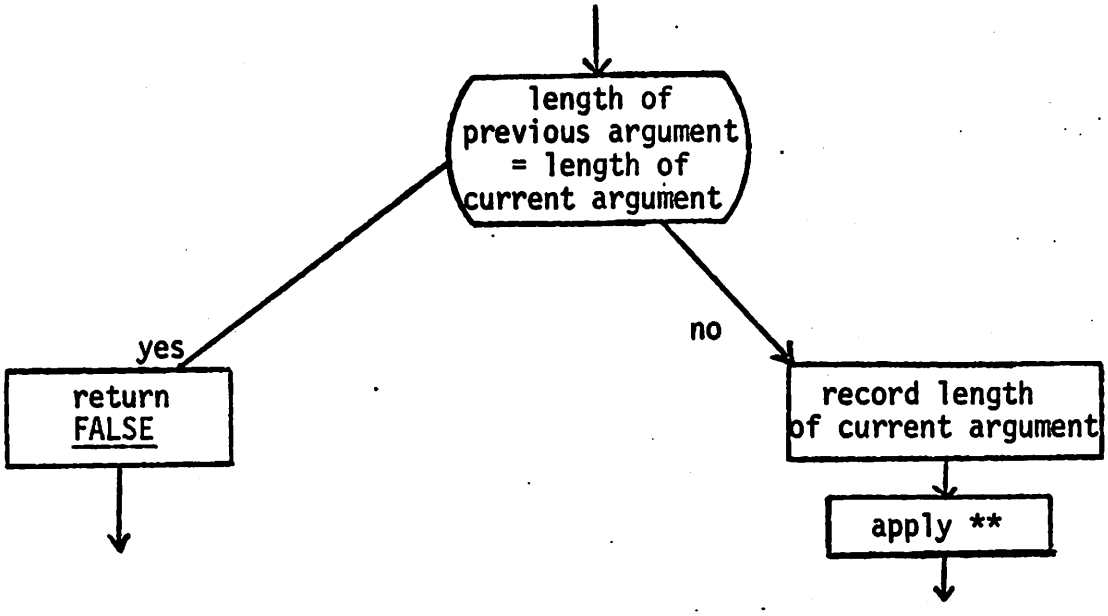
$$p_A=OR(p_{A'X'},p_{'Y'})$$

and the definition of $p_{A'X'}$ will create a recursive call of p_A. The problem is that the definition of p_A is applied to the null string. From the definition of $p_{A'X'}$, it can be seen that whenever this predicate is applied to any string, it involves applying p_A to the null string. Hence $p_A=\Omega$, i.e., the predicate is everywhere undefined. If $p_A(\eta)$ is known (η=null string) then P_A could be defined:

$$p_A(x)=\text{if } x=\text{null then } \_\_ \text{ else } OR(p_{A'X'},p_{'Y'}).$$

However, a priori, $p_A(\eta)$ is not known. The solution adopted is to modify p_A so that it updates the state vector with the length of its argument, it checks to see if it was previously called with an argument of the same length, and if it was, it returns FALSE; otherwise it applies the predicate ** to the argument. Thus, each var-

iable in the program has an entry in VARIABLETABLE of the form



In all of the semantic definitions, this is the only construction that is somewhat unnatural. It may be possible to avoid this construction as follows:

Let $A=\beta_1/.../\beta_n$

be a variable definition. Pass the variable name A to each of the patterns $\beta_1,\beta_2,...,\beta_n$. When defining the predicate associated with each β_i, if A occurs in the pattern, appropriately restrict the strings to which the predicate associated with A is applied in defining the predicate associated with the pattern β_i.

A transformation defined by

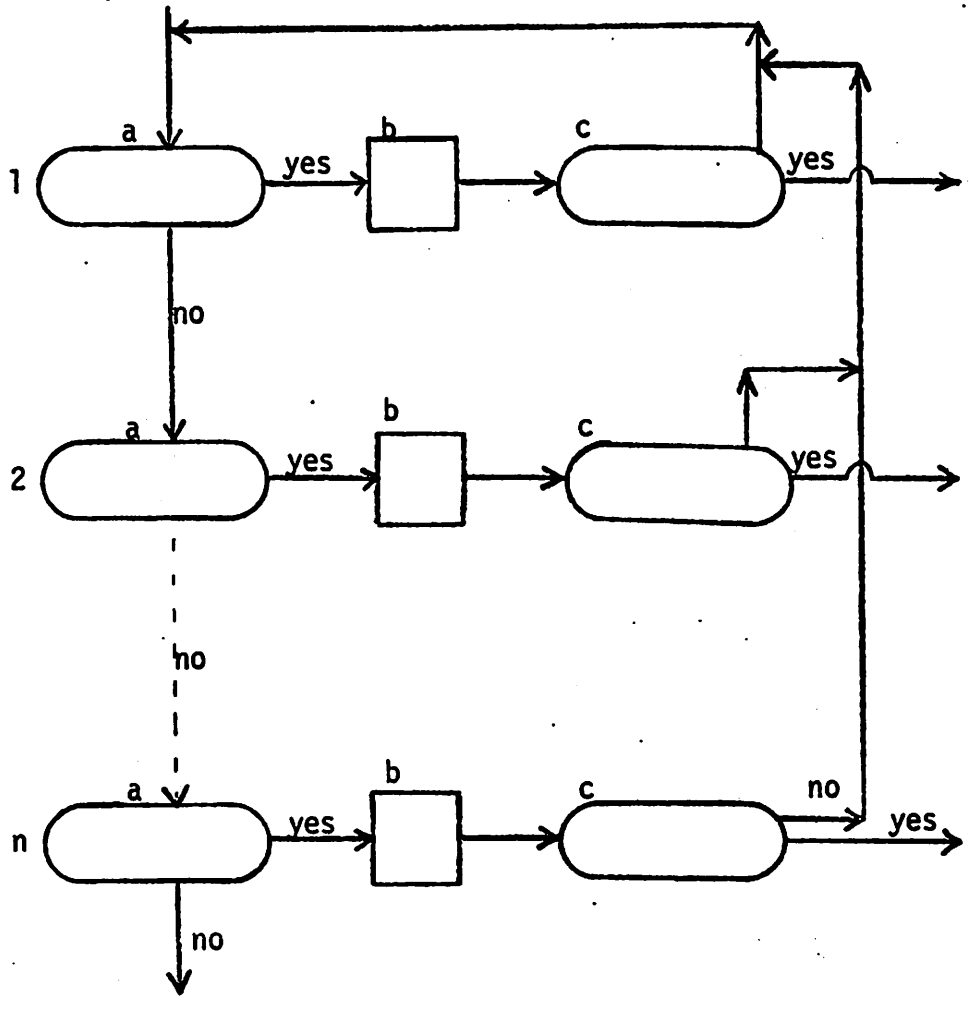
Let $T1=<<a_1;$

$a_2;$

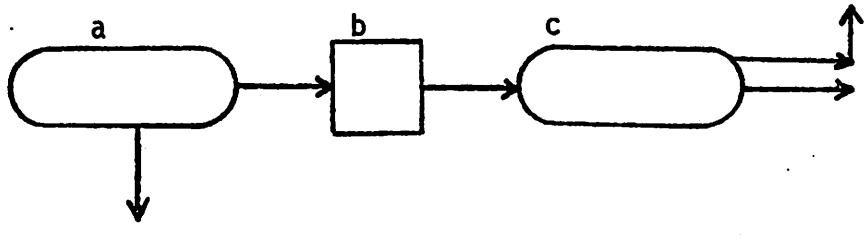
$\vdots$

$a_n>>$

corresponds to the flowchart of the form



where each sub-flowchart of the form



corresponds to an action. The node labelled a represents the predicate "does the pattern of this action match a substring of the argument to the transformation?". The node labelled b represents the indicated replacement in the argument string. The node label-

led c represents the predicate "is this action terminal?". Each transformation defined in a program has such a flowchart, and that flowchart is entered into TRANSTABLE. The complexity of the flowchart for a transformation represents the complexity of the definition for transformations.

4.9.3 Formal Definition

The basic functions used in the definition of mini-language 9 are

1) the Vienna operators U0, U, U1, and SEL.

2) SUBSTR(α,β,γ) where α and $\beta \varepsilon N$, and γ is a string.

   SUBSTR(α,β,γ)=the substring of γ starting with the α^{th} character and ending with the β^{th} character

3) LNX(α), where α is a string; the value of LNX(α) is the number of characters in α.

4) COND(α,β,γ), as defined for mini-language 4.

5) LEQ(α,β), where α and $\beta \varepsilon N$.

   $$\text{LEQ}(\alpha,\beta)=\begin{cases}\text{TRUE} & \text{if } \alpha \leq \beta \\ \text{FALSE} & \text{otherwise}\end{cases}$$

6) PARTITION(α,β,γ,ξ), where α, β, γ and ξ are ε N. A partition of a string of length ξ into substrings is a set of numbers

   $$0 \leq i_1 \leq i_2 \leq \ldots \leq i_{\alpha-1} \leq \xi+1$$

   These ordered sets $(i_1,\ldots,i_{\alpha-1})$ can be ordered as follows:

   $$(i_1,\ldots,i_{\alpha-1})<(j_1,\ldots,j_{\alpha-1})$$

   if $i_k=j_k$ for $k=1,\ldots,m$, $m \geq 0$

   and $i_{m+1}<j_{m+1}$.

PARTITION returns the β^{th} coordinate of the n^{th} partition in the above ordering.

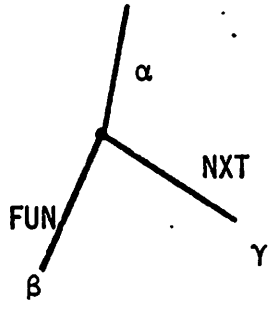
7) PLIM(α,β) is the total number of partitions of a string of length β into α substrings.

8) IDEN(α,β), defined for strings α and β

$$\text{IDEN}(\alpha,\beta) = \begin{cases} \underline{\text{TRUE}} \text{ if } \alpha \text{ is identical to } \beta \\ \underline{\text{FALSE}} \text{ otherwise} \end{cases}$$

Following the formal definition, the most significant attribute equations will be discussed.

Note: MKFLC(α,β,γ) creates the node



MINI-LANGUAGE 9

SYNTACTIC AND SEMANTIC DESCRIPTION

```
<PROGRAM>A=<VARIABLE DEF LIST>B ';' <TRANSFORMATION DEF LIST>C ';'
. <RESULT EXPRESSION>D ';END'
          <EXPRESSION>A E=UO(0,U(UO('FUN',D↑E'),UO('NXT',NUL)))
          <VARIABLETABLE>A↑V=B↑V
          <TRANSTABLE>A↑T=C↑T
          <CONCATLIB>A↑C=U(B↑C,C↑C)

<VARIABLE DEF LIST>A=<VARIABLE DEFINITION>B ';' <VARIABLE DEF LIST>C;
.D=<VARIABLE DEFINITION>E
          <VARIABLETABLE>A↑V=U(B↑V,C↑V);D↑V=E↑V
          <CONCALIB>A↑C=U(B↑C,C↑C);D↑C=E↑C

<VARIABLE DEFINITION>A=<$LETTR>B '=' <PATTERN>C
          <VARIABLETABLE>A↑V=UO('B↑VAL',U(
          .MKFLC(0,'S','COND(EQU(SEL("B↑VAL" "LNX",S), LNX(SEL("STR",S))),1,2)'),
          .MKFLC(1,'"F"',),
          .MKFLC(2,'U1(S,"B↑VAL" "LNX", LNX(SEL("STR",S)))',3),
          .MKFLC(3,'C↑P',)
          .))
          <CONCALIB>A↑C=C↑C

<PATTERN>A=<SIMPLE PATTERN>B  ' / ' <PATTERN>C;D=<SIMPLE PATTERN>E
          <PREDICATE>A↑P='OR(B↑P,C↑P)';D↑P=E↑P
          <CONCALIB>A↑C=U(B↑C,C↑C);D↑C=E↑C

<SIMPLE PATTERN>A=<CONCAT LIST>B;C='&'
            <PREDICATE>A↑P='SEL("TF",A↑F)';C↑P='SEL("TF",C↑F)'
            <FUNCTION>A↑F='APPLI(SEL(A↑I,CONCATLIB),S)';C↑F='UO("TF",EQU(0,
            .LEN(SEL("STR",S))))'
            <ID>A↑I=UNO();C↑I='NUL'
            <CONCATLIB>A↑C=UO(A↑I,U(MKFLC(0,'U1(S,"N",1)',1),
            .MKFLC(1,'S','COND(B↑P,2,3)'),
            .MKFLC(2,'U(S,B↑F,UO("TF","T"))',),
            .MKFLC(3,'U1(S,"N",SEL("N",S) + 1)',4),
            .MKFLC(4,'S','COND(LEQ(SEL("N",S),PLIM(B↑T,LNX(SEL("STR",S)))),1,5)'),
            .MKFLC(5,'UO("TF","F")',)))
           B↑T=B↑L;B↑N=1;B↑V='NUL'

<CONCAT LIST>A=<PATTERN ELEMENT>B <CONCAT LIST>C;D=<PATTERN ELEMENT>E
            <TOTALEN>T
            <NUMBER>N
            <VARTABLE>V
            <LENGTH>A↑L=1 + C↑L;D↑L=1
            <PREDICATE>A↑P='AND(B↑P,C↑P)';D↑P=E↑P
            <FUNCTION>A↑F='U(B↑F,C↑F)';D↑F=E↑F
            B↑T=A↑T;B↑N=A↑N;B↑V=A↑V
            C↑T=A↑T;C↑N=A↑N + 1;C↑V=U(A↑V,B↑X)
            E↑T=D↑T;E↑N=D↑N;E↑V=D↑V

<PATTERN ELEMENT>A='-' <$ALPHX>B '-';C=<$LETTR>D
            <TOTALEN>T
            <NUMBER>N
            <VARTABLE>V
            <XVARTABLEENTRY>A↑X='NUL';C↑X=UO('D↑VAL',C↑N)
            <SPECIAL>A↑P='PIK("B↑VAL",A↑T,A↑N)';
```

```
      .C↑P=COND(ELEM('D↑VAL',C↑V),    'P2K(' SEL('D↑VAL',C↑V) ',C↑T,C↑N)'
         .,'P3K("D↑VAL",C↑T,C↑N)')
          <PREDICATE>A↑Z='IDEN("B↑VAL",SUBSTR(PARTITION(A↑T,2 * A↑N - 1,
          .SEL("N",S),LNX(SEL("STR",S))),PARTITION(A T,2 * A↑N,SEL("N",S),
          .LNX(SEL("STR",S))),SEL("STR",S))';
          .C↑Z=COND(ELEM('D↑VAL',C↑V),
          .'IDEN(SUBSTR(PARTITION(C↑T,2 * C↑N - 1,SEL("N",S),LNX(SEL("STR",S))),
          .PARTITION(C↑T,2 * C↑N,SEL("N",S),LNX(SEL("STR",S))),SEL("STR",S)),
          .SUBSTR(PARTITION(C↑T,2 * ' SEL('D↑VAL',C↑V) ' - 1,SEL("N",S),
          .LNX(SEL("STR",S))),PARTITION(C↑T,2 * '  SEL('D↑VAL',C↑V) ',SEL("N",S),
          .LNX(SEL("STR",S))),SEL("STR",S)))',
          .'APPLI(SEL("D↑VAL",VARLIB),U1(S,"STR",SUBSTR(PARTITION(C↑T,2 * C↑N - 1
          .,
          .SEL("N",S),LNX(SEL("STR",S))),PARTITION(C↑T,2 * C↑N,SEL("N",S),
          .LNX(SEL("STR",S))),SEL("STR",S))))')
          <FUNCTION>A↑F='NUL';C↑F='UO("D↑VAL",SUBSTR(PARTITION(C↑T,2 * C↑N - 1,
          .SEL("N",S),LNX(SEL("STR",S))),PARTITION(C↑T,2 * C↑N,SEL("N",S),
          .LNX(SEL("STR",S))),SEL("STR",S)))'

<TRANSFORMATION DEF LIST>D=<TRANSFORMATION DEFINITION>E;
.<TRANSFORMATION DEF LIST>A=<TRANSFORMATION DEFINITION>B ';'
. <TRANSFORMATION DEF LIST>C
          <TRANSTABLE>A↑T=U(B↑T,C↑T);D↑T=E↑T
          <CONCATLIB>A↑C=U(B↑C,C↑C);D↑C=E↑C

<TRANSFORMATION DEFINITION>A='LET T' <$NUMBR>B '=<<' <ACTION LIST>C '>>'
          <CONCATLIB>A↑C=C↑C
          <TRANSTABLE>A↑T=UO(B↑VAL,U(MKFLC(0,'S',C↑L),C↑F))
          C↑N='NUL'

<ACTION LIST>A=<ACTION>B ';' <ACTION LIST>C;D=<ACTION>E
          <NEXT>N
          <FLOWCHART>A↑F=U(B↑F,C↑F);D↑F=E↑F
          <LINENO>A↑L=B↑L;D↑L=E↑L
          B↑N=C↑L
          C↑N=A↑N
          E↑N=D↑N
          <CONCATLIB>A↑C=U(B↑C,C↑C);D↑C=E↑C

<ACTION>A=<SIMPLE PATTERN>B ' #' <TERMINATION CHARACTER>C <REPLACEMENT STRING>D
          <NEXT>N
          <CONCATLIB>A↑C=B↑C
          <LINENO>A↑L=UNO()
          <UNIQUENOS>A↑U=U(UO(1,UNO()),UO(2,UNO()),UO(3,UNO()),UO(4,UNO()),
          .UO(5,UNO()))
          <S>A↑S=UNO()
          <KLUDGE>A↑F=U(A↑X,A↑Y)
          <FLOWCHART>A↑X=U(
          .MKFLC(A↑L,  'U(UO("P1",1),UO("P2",1),UO("STR",S))',SEL(1,A↑U)),
          .MKFLC(SEL(1,A↑U),'S','COND(APPLI( B↑P ,U1(S,"STR",SUBSTR(SEL("P1",S),
          .SEL("P2",S),SEL("STR",S)))),' SEL(2,A↑U) ',' SEL(3,A↑U) '))',
          .MKFLC(SEL(2,A↑U),'SUBSTR(1,SEL("P1",S)    ,SEL("STR",S))
         . APPLI( D↑F ,APPLI("B↑F",U1(S,"STR",SUBSTR(SEL("P1",S),SEL("P2",S),SEL("S
         .TR",
          .S))))))
          . SUBSTR(SEL("P2",S),LNX(SEL("STR",S)) + 1,SEL("STR",S))',
          .COND('C↑T',,0)),
          .)
          <Y>A↑Y=U(
          .MKFLC(SEL(3,A↑U),'U1(S,"P2",SEL("P2",S) + 1)','COND(LEQ(SEL("P2",S) -
```

```
          . 1,
          .LNX(SEL("STR",S))),' SEL(1,A↑U) ',' SEL(4,A↑U) ')'),
          .MKFLC(SEL(4,A↑U),'U(S,UO("P1",SEL("P1",S) + 1),UO("P2",SEL("P1",S) + 2
          .))',
          .'COND(LEQ(SEL("P1",S),LNX(SEL("STR",S))),' SEL(1,A↑U) ',' SEL(5,A↑U)
          . ')'),
          .MKFLC(SEL(5,A↑U),'SEL("STR",S)',A↑N))

<TERMINATION CHARACTER>A=' ';B=' .'
          <TERMINAL>A↑T='F';B↑T='T'

<REPLACEMENT STRING>A=<STRING REF LIST>B;C='&'
          <FUNCTION>A↑F=B↑F;C↑F='NUL'

<STRING REF LIST>A=<STRING REF>B <STRING REF LIST>C;D=<STRING REF>E
          <FUNCTION>A↑F='B↑F C↑F';D↑F=E↑F

<STRING REF>A='-' <$ALPHX>B '-';C=<$LETTR>D
          <FUNCTION>A↑F='"B↑VAL"';
          .C↑F='SEL("D↑VAL",S)'

<RESULT EXPRESSION>A='-' <$ALPHX>B '-';C='T' <$NUMBR>D '<' <RESULT EXPRESSION>E
. '>'
          <EXPRESSION>A↑E='"B↑VAL"';C↑E='APPLL(SEL(D↑VAL,TRANSLIB),E↑E)'
END
```

The flowchart corresponding to a variable definition is given by the rule

```
<VARIABLE DEFINITION>A=<$LETTR>B '=' <PATTERN>C
        <VARIABLETABLE>A↑V=UO('B↑VAL',U(
        .MKFLC(0,'S','COND(EQU(SEL("B↑VAL" "LNX",S), LNX(SEL("STR",S))),1,2
        .'),
        .MKFLC(1,'"F"',),
        .MKFLC(2,U1(S,"B VAL" "LNX", LNX(SEL("STR",S)))',3),
        .MKFLC(3,'C↑P',)
        .))
```

Letting α_1 represent the length of the previous argument for this function, and α_2 the length of the current argument, the functionals corresponding to the above flowchart are

F_0=if $\alpha_1=\alpha_2$ then F_1 else F_2

F_1="F" Note: "F" represents FALSE.

$F_2=F_3$(U1(S,"B↑VAL",LNX(α_2)))

F_3=C↑P

- - - - - - - -

Corresponding to a simple pattern there is an entry in CONCATLIB defined by

```
<SIMPLE PATTERN>A=<CONCAT LIST>B;C='ε'
        <CONCATLIB>A↑C=UO(A↑I,U(MKFLC(0,'U1(S,"N",1)',1),
        .MKFLC(1,'S','COND(B↑P,2,3)'),
        .MKFLC(2,'U(S,B↑F,UO("TF","T"))',),
        .MKFLC(3,'U1(S,"N",SEL("N",S) + 1)',4),
        .MKFLC(4,'S','COND(LEQ(SEL("N",S),PLIM(B↑T,LNX(SEL("STR",S)))),1,5)
        .'),
        .MKFLC(5,'UO("TF","F")',)))
```

Informally, the functions of the nodes in this flowchart are

0. Initialize n to 1.

1. Try the pattern predicate

2. If "1" succeeds, return "TRUE".

3. If "1" fails, increment n.

4. Any more partitions? If yes, go to "1".

5. Otherwise return "F", i.e., FALSE.

The predicate corresponding to a pattern element is given by

```
<PATTERN ELEMENT>A='-' <$ALPHX>B '-';C=<$LETTR>D
      <PREDICATE>A↑Z='IDEN("B↑VAL",SUBSTR(PARTITION(A↑T,2 * A↑N - 1,
       .SEL("N",S),LNX(SEL("STR",S))),PARTITION(A↑T,2 * A↑N,SEL("N",S),
       .LNX(SEL("STR",S))),SEL("STR",S)))';
       .C↑Z=COND(ELEM('D↑VAL',C↑V),
       .'IDEN(SUBSTR(PARTITION(C↑T,2 * C↑N - 1,SEL("N",S),LNX(SEL("STR",S
       .))),
       .PARTITION(C↑T,2 * C↑N,SEL("N",S),LNX(SEL("STR",S))),SEL("STR",S))
       .SUBSTR(PARTITION(C↑T,2 * ' SEL('D↑VAL',C↑V) ' - 1,SEL("N",S),
       .LNX(SEL("STR",S))),PARTITION(C↑T,2 * ' SEL('D↑VAL',C↑V) ',SEL("N"
       .,S),
       .LNX(SEL("STR",S))),SEL("STR",S)))',
       .'APPLI(SEL("D↑VAL",VARLIB),U1(S,"STR",SUBSTR(PARTITION(C↑T,2 * C↑
       . -1,
       .SEL("N",S),LNX(SEL("STR",S))),PARTITION(C↑T,2 * C↑N,SEL("N",S),
       .LNX(SEL("STR",S))),SEL("STR",S))))')
       <FUNCTION>A↑F='NUL';C↑F='UO("D↑VAL",SUBSTR(PARTITION(C↑T,2 * C↑N-1
       .SEL("N",S),LNX(SEL("STR",S))),PARTITION(C↑T,2 * C↑N,SEL("N",S),
       .LNX(SEL("STR",S))),SEL("STR",S)))'
```

Note: this expression was too complicated for SNOBOL, and as a result, the attribute SPECIAL was defined. The value of this attribute is identical to the value of PREDICATE, except complicated expressions are replaced by a subroutine call, making the total expression simple enough to be compiled without causing the compiler to conk out.

Let α_A=SUBSTR(PARTITION(A↑T,2 * A↑N-1,
SEL("N",S),LNX(SEL("STR",S))),
PARTITION(A↑T,2 * N,SEL("N",S),
LNX(SEL("STR",S))),SEL("STR",S))).

α_C=α_A with C substituted for A

$\alpha_{C'}$=α_C with SEL("D↑VAL",C↑V) substituted for C↑N

then the above attribute equation simplifies to the docile

A↑Z=IDEN("B↑VAL",α_1);

C↑Z=COND(ELEM("D↑VAL",C↑V),IDEN(α_C,α_C),

"APPLI(SEL("D↑VAL",VARLIB),U1(S,"STR",α_C)))

- - - - - - - -

The flowchart corresponding to an action is given by (here KLUDGE was introduced because the expression for the flowchart was too complicated to be compiled):

```
<ACTION>A=<SIMPLE PATTERN>B' #' <TERMINATION CHARACTER>C <REPLACEMENT STRING>[
          <KLUDGE>A↑F=U(A↑X,A↑Y)
          <FLOWCHART>A↑X=U(
          .MKFLC(A↑L, 'U(UO("P1",1),UO("P2",1),UO("STR",S))',SEL(1,A↑U)),
          .MKFLC(SEL(1,A↑U),'S','COND(APPLI( B↑P ,U1(S,"STR",SUBSTR(SEL(,
          ."P1",S),
          .SEL("P2",S),SEL("STR",S)))),' SEL(2,A↑U" ',' SEL(3,A↑U) ')'),
          .MKFLC(SEL(2,A↑U),'SUBSTR(1,SEL("P1",S)    ,SEL("STR",S))
        . APPLI( D↑F ,APPLI("B↑F",U1(S,"STR",SUBSTR(SEL("P1",S),SEL("P2",S),
        .SEL("STR",
          .S)))))
          . SUBSTR(SEL("P2",S),LNX(SEL("STR",S)) + 1,SEL("STR",S)),
          .COND('C↑T',,0)),
          .)
          <Y>A↑Y=U(
          .MKFLC(SEL(3,A↑U),'U1(S,"P2",SEL("P2",S) + 1)','COND(LEQ(SEL("P2,,
          .S) - 1,
          .LNX(SEL("STR",S))),' SEL(1,A↑U) ',' SEL(4,A↑U) ')'),
          .MKFLC(SEL(4,A↑U),'U(S,UO("P1",SEL("P1",S) + 1),UO("P2",SEL("P1,S,
          .) + 2))',
          .'COND(LEQ(SEL("P1",S),LNX(SEL("STR",S))),' SEL(1,A↑U) ',' SEL(,
          .5,A↑U)
          .')'),
          .MKFLC(SEL(5,A↑U),'SEL("STR",S)',A↑N))
```

Informally, the functions of the nodes in the flowchart are:

0. Initialize p_1 and p_2.

1. Does the pattern of the action match SUBSTR(p_1,p_2,α), where

α is the string argument?

2. Yes, make the appropriate substitution and go to the first node of the action list, or return, if this action is terminal.
3. No, increment and test p_2.
4. Increment p_1 if necessary; go to next action if no more substrings to test against pattern, otherwise go to "1".

4.10 Mini-Language 10: Input/Output

4.10.1 Ledgard's Description (Verbatim)

Description of Language Elements

The notion of a stream of items is akin to the notion of a "list" of items, except that the items in a stream are determined dynamically. In a sense, a stream is a (perhaps infinite or initially undetermined) list whose items flow by, one by one. A stream S is specified by a list of expressions $e_1,e_2,\ldots,e_n$. The first item in the stream is obtained by evaluating the first expression, e_1. If e_1 denotes only one item, we take it as the first item in the stream and use the list $e_2,\ldots,e_n$ to obtain subsequent items. If e_1 denotes a series of two or more items, the first item in the series becomes the first item in the stream, e_1 is changed to expression e_1', indicating that its first item has been removed, and the list $e_1',e_2,\ldots,e_n$ is used to obtain subsequent items. A stream is "exhausted" when the list used to obtain an item is null. A stream is "recycled" when the original list of expressions $e_1,e_2,$

..., e_n is used again to obtain subsequent items.

Two types of streams are used in mini-language 10: "layout" streams, which specify the way in which characters are arranged on the input or output medium, and "data" streams, which specify data objects whose character representations appear on the input or output medium.

Let us assume that we have two devices, the input medium and the output medium, each of which can be viewed as a teletype containing an infinitely long piece of paper allowing 100 characters per line. On the input medium we shall look at the characters printed thereon in the conventional order, from left to right and line by line. On the output medium we shall print characters in the conventional order.

The array identifiers comprise the symbols A1 A2.... Each array identifier represents a one-dimensional array with an unspecified number of elements. Data stream identifiers comprise the symbols D1 D2.... Layout stream identifiers consist of the symbols L1 L2....

An expression is either a numeral, a named expression, or an arithmetic expression. A named expression is either an identifier, or an array identifier followed by an expression enclosed in square brackets. An arithmetic expression is a string of either the form (e_1+e_2) or (e_1-e_2), where e_1 and e_2 are expressions.

A layout expression is either a unit layout expression or a replicated layout expression. A replicator is an expression. A

unit layout expression consists of a replicator followed by one of the symbols d, b, or ℓ. A replicated layout expression consists of a replicator followed by a unit layout expression enclosed in square brackets.

A layout stream declaration is a string of the form

layout stream p=ℓ

where p is a layout stream identifier, and ℓ is a list of layout expressions. The items of a layout stream are obtained as follows:

1) If the first expression ℓ_i of the current list $\ell_i,\ell_j,\ldots,\ell_n$ is a unit layout item, then ℓ_i is taken as the next item, and the list $\ell_j,\ldots,\ell_n$ is used to obtain successive items.
2) If the first expression ℓ_i of the current list $\ell_i,\ell_j,\ldots,\ell_n$ is a replicated layout expression of the form r[u], where r is a replicator and u is a unit layout expression, then if r is an expression whose current value is ≥ 1, u is taken as the next item, and the list (r-1)[u], $\ell_j,\ldots,\ell_n$ is used to obtain successive items; otherwise, the list $\ell_j,\ldots,\ell_n$ is used to obtain the next and successive items.

A data expression is either a unit data expression or an iterative data expression. A unit data expression is a named expression. An iterative data expression is a string of the form

b for i:=e_1 to e_2

where b is a named expression, i an identifier, and e_1 and e_2 are expressions.

A data stream declaration is a string of the form

data stream p=ℓ

where p is a data stream identifier, and ℓ is a list of data expressions. The items of a data stream are obtained as follows:

1) If the first expression ℓ_i on the current list $\ell_i,\ell_j,\ldots,\ell_n$ is a unit data expression, then ℓ_i is taken as the next item of the stream, and the list $\ell_j,\ldots,\ell_n$ is used to obtain successive items.

2) If the first expression ℓ_i on the current list $\ell_i,\ell_j,\ldots,\ell_n$ is an iterative data expression of the form:

   b for i:=e_1 to e_2

   then let n_1 and n_2 be the current values of e_1 and e_2:

   a) if $n_1 \neq n_2$, then i is assigned the value n_1, b is taken as the next item of the stream, and the list

      b for i:=($e_1 \pm 1$) to $e_2,\ell_j,\ldots,\ell_n$

      is used to obtain subsequent items (here (e_2+1) is used when $n_1<n_2$, and (e_1-1) is used when $n_1>n_2$.); or

   b) if $n_1=n_2$, i is assigned the value n_1,b is taken as the next item of the stream, and the list $\ell_j,\ldots,\ell_n$ is used to obtain successive items.

A command is either an input command, an output command, an assignment command, or an iteration command.

Input and output commands are either of the strings input(ℓ,d) or output(ℓ,d), where ℓ is a layout stream identifier, and d is a data stream identifier. An input or output command is executed as follows:

1) The next item on the layout stream ℓ is obtained; a) if the

layout item is of the form *rb* or *rℓ*, where *r* is a replicator, a spacing input or output action takes place (see Table II) and step (1) is repeated; or b) if the layout item is of the form *rd*, the next item on the data stream *d* is obtained, a data input or output action takes place (see Table II), and step (1) is repeated. If the data stream happens to be exhausted, it is recycled to obtain the next item.

TABLE II. INPUT/OUTPUT ACTIONS

Let ℓ_i be the next item on the layout stream, *n* be the value of the replicator *r* for ℓ_i, and let *m* be the named expression for the next item on the data stream.

Form of Layout item	Action on input	Action on output
Spacing:		
rb	The next *n* characters on the input medium are skipped.	The next *n* characters on the output medium are printed as blanks.
rℓ	The remaining characters on the current line and the next n-1 lines on the input medium are skipped.	The remaining characters on the current line and the next n-1 lines on the output medium are printed as blanks.
Data:		
rd	The next *n* characters on the input medium will be treated as a natural number and will be assigned as the value of *m*. If the next *n* characters are not a well-formed natural number, the input action is in violation.	The leftmost *n* digits of the value of *m* will be placed on the output medium. If the value of *m* can be specified by fewer than *n* digits, the number will be right-justified with leading zeros suppressed.

2) If the layout stream is exhausted, then, if the data stream is not exhausted, the layout stream is recycled and step (1) is

repeated, or, if the data stream is also exhausted, the input or output command is terminated.

Note that an input or output command terminates only when both the layout and data streams are exhausted simultaneously.

An assignment command is a string of the form p:=e, where p is a named expression, and e is an expression.

An iteration command is a string of the form

for i:=e_1 to e_2 do[c]

where i is an identifier, e_1 and e_2 are expressions, and c is a sequence of commands. The execution of an iteration command is as follows: let n_1 and n_2 be the numerical values of e_1 and e_2, and let m be the absolute value of (n_1-n_2). The sequence of commands c is executed (m+1) times, and on each iteration the value of the identifier i is successively set to one of the integer values n_1 up to n_2 (if $n_1 \leq n_2$), or n_1 down to n_2 (if $n_1 \geq n_2$).

Program Execution

A program consists of a sequence d of layout and data stream declarations followed by a sequence c of commands such that each layout and data stream identifier in c is declared once and only once in d. The commands in a program are executed sequentially. We take the conventional meaning for assignment commands and the operations "+" (addition) and "-" (subtraction) for natural numbers. Any attempt to input or output more than 100 characters per line is in violation.

EXAMPLE 1:

```
layout stream L1=(4d,4d,4d)
data stream   D1=A,B
A:=11
B:=22
output(L1,D1)
```

EXAMPLE 2:

```
layout stream L1=3d,1ℓ
data stream   D1=N
layout stream L2=1b,3d,1b,3d,1ℓ
data stream   D2=A,B
data stream   D3=B,A
input(L1,D1)
for X:=1 to N do[input(L2,D2)
                 output(L2,D3)]
```

EXAMPLE 3:

```
layout stream L1=(48-(J+J))b,
                 J[4d],1ℓ
data stream   D1=A1[I] for I:=1 to J
for J:=1 to 2 do[A1[j]:=1
                 output(L1,D1)]
for J:=3 to 5 do[A1[J]:=1
                 for K:=(J-1) to 2
                   do[A1[K]:=(A1[K]
                   +A1[(K-1)])]
                 output(L1,D1)]
```

EXAMPLE 4:

```
layout stream L1=(48-(J+J))b,J[4d],
                 1ℓ
data stream D1=A;[I] for I:=K
                     to (K-(J-1))
data stream D2=A1[I] for K:=K
                     to(K-(J-1))
K:=1
for J:=1 to 4 do[input(L1,D1)
                 K:=(K+J)]
K:=(K-1)
for J:=4 to 1 do[output(L1,D2)
                 K:=(K-J)]
```

In Example 1, both the layout and data streams are recycled, and the string printed on the output medium is:

```
 11   22   11   22   11   22
```

In Example 2, if the first line of the input device contains a natural number n as its first three characters, and the next n lines contain two (appropriately spaced) columns of numbers, then the two columns are printed in reverse order on the output medium. For example, if

```
3
    1     2
   11    22
  111   222
```

appears on the input medium, then

```
  2     1
 22    11
222   111
```

will be printed on the output medium.

In Example 3, the first five rows of Pascal's triangle

```
          1
        1   1
      1   2   1
    1   3   3   1
  1   4   6   4   1
```

will be printed on the input medium.

In Example 4, if the first four lines on the input medium contain an (appropriately spaced) triangular configuration of numbers, the configuration is inverted row-wise and column-wise on the output medium. For example, if

```
          1
        2   3
      4   5   6
    7   8   9   10
```

appears on the input medium, then

```
10    9    8    7
   6    5    4
      3    2
           1
```

will appear on the output medium.

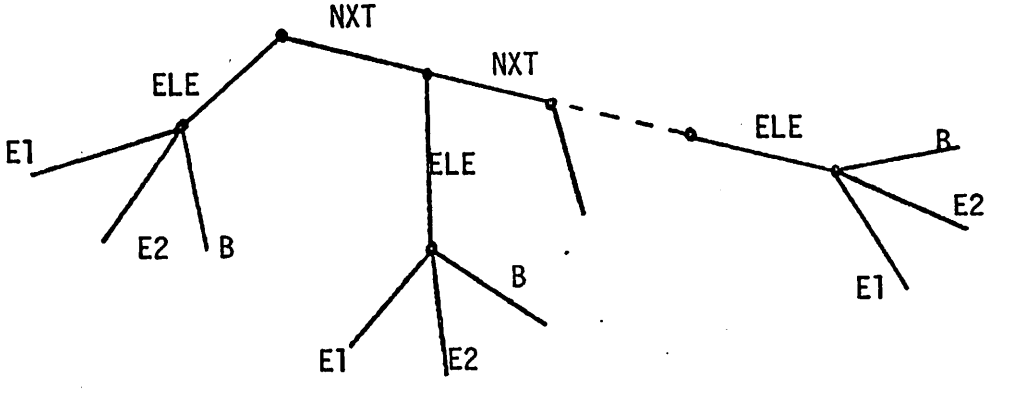
4.10.1.a Notational Changes

Square brackets [ ] are replaced by angle brackets < >.

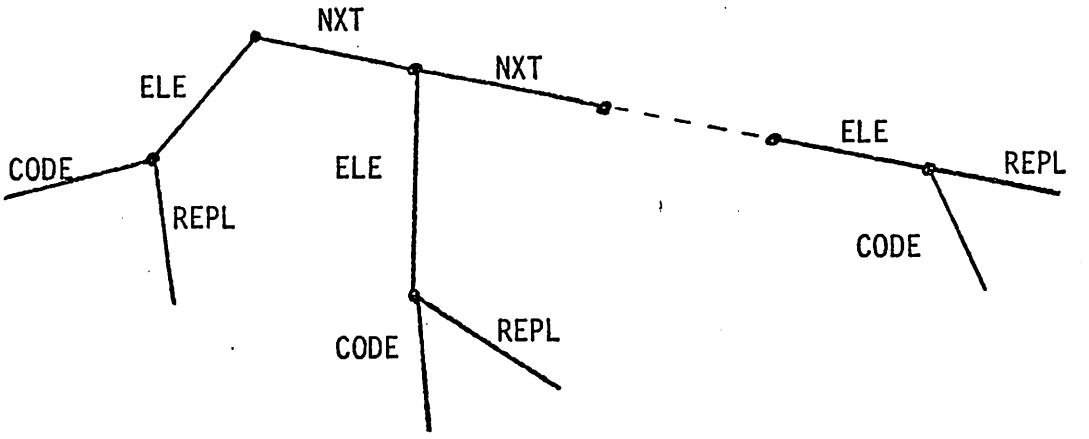
4.10.2 Discussion

The data stream and the layout stream of mini-language 10 correspond roughly to the variable list and format in FORTRAN I/O statements. The primary difference is that the data stream and layout stream are determined dynamically, and in a complicated manner. As an aside, it seems ridiculous that Ledgard could offer mini-language 10 to satisfy the need for "standardized, straightforward techniques" for specifying I/O.

In our treatment of mini-language 10, data streams and layout streams will be represented by list-valued Vienna objects. Data streams are represented by VO's of the form



Layout streams are represented by VO's of the form



These data structures are constructed by the attribute equations for data stream declarations and layout stream declarations.

Two functions are defined in PROCLIB named GNDI and GNLI, which are acronyms for "Gen Next Data Item" and "Get Next Layout Item", respectively. The argument for GNDI is a data stream, and GNDI returns a data reference and a new value of the data stream. GNLI takes a layout stream as an argument and returns a layout specification and a new value for the layout stream. The flowcharts for these two functions correspond closely to the actions of getting a data item and getting a layout item as discussed on pages 195 and 196.

Input and output statements are very complicated in mini-language 10, as they involve stepping simultaneously through the data stream and layout stream (using GNDI and GNLI), and if one stream becomes exhausted while the other is not, "recycling" the exhausted stream and finally terminating when both streams are exhausted simul-

taneously. The action of the I/O statement is on pages 196 and 197. In the flowchart for a program in mini-language 10, each I/O statement is represented by a "sub" flowchart which corresponds to the action of the I/O statement as given by Ledgard. These "sub" flowcharts will be discussed in detail in the next section, but a word is in order here about their complexity. In both mini-language 9 and 10, very complicated actions can be specified very concisely; as a result, the flowchart corresponding to a "simple" program in one of these languages can be surprisingly complex, since all of the implied actions are made explicit.

Let f_p be the function corresponding to a program P. The I/O actions of P will be represented in f_p by the functions READX(α,β), where α is an integer representing the field size and β is r, d, or ℓ, and the function PRINTX(α,β,γ) where α and β are as above, and γ is an integer (the data item). The state of the input device is not incorporated into the initial state vector, and when the computation rule APPLI(f_p,S) requires evaluation of the function READX(α,β), the value is obtained by interpreting the actual input stream to the semantics-based interpreter appropriately; when PRINTX(α,β,γ) is evaluated, it returns the null result, and it is also appropriately interpreted and the output appears on the actual output medium. Thus, a program interpreted by the semantics-based interpreter will be followed by the cards representing its input, and its output appears on the line printer (see pages 373-402, Appendix I).

4.10.3 Formal Definition

The basic functions used in the definition of mini-language 10 are

1) the Vienna operators UO, U, U1 and SEL.

2) READX and PRINTX, discussed in the previous section.

3) LTN(α,β), where α, $\beta \in N$

$$LTN(\alpha,\beta) = \begin{cases} \text{TRUE} & \text{if } \alpha < \beta \\ \text{FALSE} & \text{otherwise} \end{cases}$$

4) integer addition and subtraction

Following the formal definition, the most significant attribute equations will be discussed.

MINI-LANGUAGE 10

SYNTACTIC AND SEMANTIC DESCRIPTION

```
<PROGRAM>A=<DEC LIST>B ';' <COMMAND LIST>C ';END'
        <FLOWCHART>A↑F=C↑F
        <PROCLIB>A↑P=U(
        .U0('GNLI',U(
        .MKFLC(0,'S','COND(EQU(SEL("LIST",S),NUL),4,1)'),
        .MKFLC(1,'S','COND(LTN(APPLIF(SEL("REPL",SEL("ELE",SEL("LIST",S))),S),1
        .),
        .2,3)'),
        .MKFLC(2,'U1(S,"LIST",SEL("NXT",SEL("LIST",S)))',0),
        .MKFLC(3,'U(JO("ITEM",U(UO("REPL",APPLIF(SEL("REPL",SEL("UNIT",SEL("ELE
        .",
        .SEL("LIST",S)))),S)),UO("CODE",SEL("CODE",SEL("UNIT",SEL("ELE",SEL("LI
        .ST",S)
       .))))),UO("LIST",U1(SEL("LIST",S),"REPL?ELE",MKCON(APPLIF(SEL("REPL",
        .SEL("ELE",SEL("LIST",S))),S) - 1)))',NUL),
        .MKFLC(4,'NUL',NUL)
        .)),
        .U0('GNDI',U(
        .MKFLC(0,'S','COND(EQU(SEL("LIST",S),NUL),4,1)'),
        .MKFLC(1,'S','COND(EQU(APPLIF(SEL("E2",SEL("ELE",SEL("LIST",S))),S) -
        . APPLIF(SEL("E1",SEL("ELE",SEL("LIST",S))),S),0),2,3)'),
        .MKFLC(2,'U(UO("ITEM",APPLIF(SEL("B",SEL("ELE",SEL("LIST",S))),
        .U1(S,SEL("I",SEL("ELE",SEL("LIST",S))),APPLIF(SEL("E1",SEL("ELE",
        .SEL("LIST",S))),S)))),UO("LIST",SEL("NXT",SEL("LIST",S))))',NUL),
        .MKFLC(3,'U(UO("ITEM",APPLIF(SEL("B",SEL("ELE",SEL("LIST",S))),
        .U1(S,SEL("I",SEL("ELE",SEL("LIST",S))),APPLIF(SEL("E1",SEL("ELE",
        .SEL("LIST",S))),S)))),UO("LIST",U1(SEL("LIST",S),"E1?ELE",
        .MKCON(APPLIF(SEL("E1",SEL("ELE",SEL("LIST",S))),S) +
        . INCR(APPLIF(SEL("E2",SEL("ELE",SEL("LIST",S))),S) -
        . APPLIF(SEL("E1",SEL("ELE",SEL("LIST",S))),S))     ))))',NUL),
        .MKFLC(4,'NUL',NUL)
        .)))
        <LAYOUTLIB>A↑L=B↑L
        <DATALIB>A↑D=B↑D
        C↑N='NUL'

<DEC LIST>A=<DECLARATION>B ';' <DEC LIST>C;D=<DECLARATION>E
        <LAYOUTLIB>A↑L=U(B↑L,C↑L);D↑L=E↑L
        <DATALIB>A↑D=U(B↑D,C↑D);D↑D=E↑D

<DECLARATION>A=<LAYOUT STREAM DECLARATION>B;C=<DATA STREAM DECLARATION>D
        <LAYOUTLIB>A↑L=B↑L;C↑L='NUL'
        <DATALIB>A↑D='NUL';C↑D=D↑D

<LAYOUT STREAM DECLARATION>A='LAYOUT STREAM L' <$NUMBR>B '=' <LAYOUT LIST>C
        <LAYOUTLIB>A↑L=UO('L' B↑VAL,C↑L)

<LAYOUT LIST>A=<LAYOUT EXP>B ',' <LAYOUT LIST>C;D=<LAYOUT EXP>E
        <LIST>A↑L=U(UO('ELE',B↑L),UO('NXT',C↑L));
        .D↑L=U(UO('ELE',E↑L),UO('NXT',NUL))

<LAYOUT EXP>A=<EXPRESSION>B <CODE>C;D=<EXPRESSION>E '<' <EXPRESSION>F <CODE>G
. '>'
        <LISTEL>A↑L=U(UO('REPL',1),UO('UNIT',U(UO('REPL','B↑F'),UO('CODE','C↑C'
        .))));
```

```
        .D↑L= U(JO('REPL','E↑F'),UO('UNIT',U(UO('REPL','F↑F'),UO('CODE','G↑C'))
        .))

<CODE>A='B';B='L';C='D'
        <CODE>A↑C='B';B↑C='L';C↑C='D'

<DATA STREAM DECLARATION>A='DATA STREAM D' <$NUMBR>B '=' <DATA LIST>C
        <DATALIB>A↑D=UO('D' B↑VAL,C↑L)

<DATA LIST>A=<DATA EXP>B ',' <DATA LIST>C;D=<DATA EXP>E
        <LIST>A L=U(UO('ELE',B↑L),UO('NXT',C↑L));
        .D↑L=U(JO('ELE',E↑L),UO('NXT',NUL))

<DATA EXP>A=<NAMED EXPRESSION>B ;
.C=<NAMED EXPRESSION>D ' FOR ' <$LETTR>E '=' <EXPRESSION>F ' TO ' <EXPRESSION>G
        <LISTEL>A↑L=J(UO('B','B↑L'),U(UO('E1',0),UO('E2',0)));
        .C↑L=U(U(UO('B','D↑L'),UO('I','E↑VAL')),U(UO('E1','F↑F'),UO('E2','G↑F'))
        .))

<EXPRESSION>A=<$NUMBR>B;
.C=<NAMED EXPRESSION>D;
.E='(' <EXPRESSION>F <ARITH OP>G <EXPRESSION>H ')'
        <FUNCTION>A↑F=B↑VAL;
        .C↑F=D↑R;
        .E↑F='(F↑F G↑V H↑F)'

<NAMED EXPRESSION>A=<$LETTR>B;
.C='A' <$NUMBR>D '<' <EXPRESSION>E '>'
        <RHSFUN>A↑R='SEL("B↑VAL",S)';
        .C R='SEL(E↑F,SEL("A" D↑VAL,S))'
        <LHSFUN>A↑L='"B↑VAL"';
        .C↑L='COMPOS(E↑F,"A" D↑VAL)'

<ARITH OP>A='+';B='-'
        <VAL>A↑V='+';B↑V='-'

<COMMAND LIST>A=<COMMAND>B ';' <COMMAND LIST>C;D=<COMMAND>E
        <NEXT>N
        <LINENO>A↑L=B↑L;D↑L=E↑L
        <FLOWCHART>A↑F=U(B↑F,C↑F);D↑F=E↑F
        B↑N=C↑L
        C↑N=A↑N
        E↑N=D↑N

<COMMAND>A=<ASSIGNMENT COMMAND>B;C=<ITERATION COMMAND>D;E=<IO COMMAND>F
        <NEXT>N
        <LINENO>A↑L=B↑L;C↑L=D↑L;E↑L=F↑L
        <FLOWCHART>A↑F=B↑F;C↑F=D↑F;E↑F=F↑F
        B↑N=A↑N
        D↑N=C↑N
        F↑N=E↑N

<ASSIGNMENT COMMAND>A=<NAMED EXPRESSION>B '=' <EXPRESSION>C
        <NEXT>N
        <LINENO>A↑L=UNO()
        <FLOWCHART>A↑F=MKFLC(A↑L,'U1(S,B↑L,C↑F)',A↑N)

<ITERATION COMMAND>A='FOR ' <$LETTR>B '=' <EXPRESSION>C ' TO '
. <EXPRESSION>D ' DO <' <COMMAND LIST>E '>'
        <NEXT>N
```

```
            <LINENO>A↑L=UNO()
            <U>A↑U=U(UO(1,UNO()),UO(2,UNO()))
            <FLOWCHART>A↑F=U(
            .MKFLC(A↑L,'U1(S,"B↑VAL",C↑F)',E↑L),
            .MKFLC(SEL(1,A↑U),'S','COND(EQQ(SEL("B↑VAL",S),D↑F),A↑N,SEL(2,A↑U))'),
            .MKFLC(SEL(2,A↑U),'U1(S,"B↑VAL",SEL("B↑VAL",S) + INCR(D↑F -
            . SEL("B↑VAL",S)))',E↑L),
            .E↑F)
            E↑N=SEL(1,A↑U)

<IO COMMAND>A=<IO INST>B '(L' <$NUMBR>C ',D' <$NUMBR>D ')'
            <NEXT>N
            <LINENO>A↑L=UNO()
            <U>A↑U=U(JO(1,UNO()),UO(2,UNO()),UO(3,UNO()),UO(4,UNO()),UO(5,UNO()),
            .UO(6,UNO()),UO(7,UNO()),UO(8,UNO()),UO(9,UNO()),UO(10,UNO()))
            <FLOWCHART>A↑F=U(A↑X,A↑Y)
            <XFLOWCHART>A↑X=U(
            .MKFLC(A↑L,'U(S,U(UO("L-",APPLI(SEL("GNLI",PROCLIB),    U1(S,"LIST",
            .SEL("L"  C↑VAL,LAYOUTLIB)))) ,UO("D-",APPLI(SEL("GNDI",PROCLIB),
            .U1(S,"LIST",SEL("D" D VAL,DATALIB)))))',SEL(1,A U)),
            .MKFLC(SEL(1,A↑U),'S','COND(EQU(SEL("ITEM",SEL("L-",S)),NUL),'
            . SEL(2,A↑U) ',' SEL(4,A↑U) ')'),
            .MKFLC(SEL(2,A↑U),'S','COND(EQU(SEL("ITEM",SEL("D-",S)),NUL),A↑N,'
            . SEL(3,A↑U) ')'),
            .MKFLC(SEL(3,A↑U),'U1(S,"L-"  ,APPLI(SEL("GNLI",PROCLIB),U1(S,"LIST",
            .SEL("L" C↑VAL,LAYOUTLIB)))',SEL(1,A↑U)),
            .MKFLC(SEL(4,A↑U),'S',   'COND(EQU(SEL("CODE",SEL("ITEM",SEL("L-",S))),
             ."D"),' SEL(6,A↑U) ',' SEL(5,A↑U) ')'),
            .MKFLC(SEL(5,A↑U),'B A',SEL(10,A↑U)),
            .MKFLC(SEL(6,A↑U),'S','COND(EQU(SEL("ITEM",SEL("D-",S)),NUL),'
            . SEL(7,A↑U) ',' SEL(8,A↑U) ')'))
            <YYFLOWCHART>A↑Y=U(
            .MKFLC(SEL(7,A↑U),'U1(S,"D-",APPLI(SEL("GNDI",PROCLIB),U1(S,"LIST",
            .SEL("D" D↑VAL,DATALIB)))',SEL(8,A↑U)),
            .MKFLC(SEL(8,A↑U),'B↑B',SEL(9,A↑U)),
           .U(MKFLC(SEL(9,A↑U),'U1(S,"D-",APPLI(SEL("GNDI",PROCLIB),U1(S,"LIST",
            .SEL("LIST",SEL("D-",S)))))',SEL(10,A↑U)),
            .MKFLC(SEL(10,A↑U),'U1(S,"L-",APPLI(SEL("GNLI",PROCLIB),U1(S,"LIST",
            .SEL("LIST",SEL("L-",S)))))',SEL(1,A↑U))
            .))

 <IO INST>A='INPUT';B='OUTPUT'
            <A>A↑A='PROJECT(S,READX(SEL("REPL",SEL("ITEM",SEL("L-",S))),SEL("CODE",
            .SEL("ITEM",SEL("L-",S)))))';
            .B↑A='PROJECT(S,PRINTX(SEL("REPL",SEL("ITEM",SEL("L-",S))),SEL("CODE",
             .SEL("ITEM",SEL("L-",S)))))'
            <B>A↑B='U1(S,SEL("ITEM",SEL("D-",S)),READX(SEL("REPL",SEL("ITEM",
            .SEL("L-",S)))),SEL("CODE",SEL("ITEM",SEL("L-",S)))))';
            .B↑B='PROJECT(S,PRINTX(SEL("REPL",SEL("ITEM",SEL("L-",S))),SEL("CODE",
            .SEL("ITEM",SEL("L-",S))),SEL(SEL("ITEM",SEL("D-",S)),S)))'

 END
```

The functions GNLI and GNDI are defined by the rule

```
<PROGRAM>A=<DEC LIST>B ';' <COMMAND LIST>C ';END'
        <FLOWCHART>A↑F=C↑F
        <PROCLIB>A↑P=U(
        .UO('GNLI',U(
        .MKFLC(0,'S','COND(EQU(SEL("LIST",S),NUL),4,1)'),
        .MKFLC(1,'S','COND(LIN(APPLIF(SEL("REPL",SEL("ELE",SEL("LIST",S))),S),1
        .),
        .2,3)'),
        .MKFLC(2,'U1(S,"LIST",SEL("NXT",SEL("LIST",S)))',0),
        .MKFLC(3,'U(JO("ITEM",U(UO("REPL",APPLIF(SEL("REPL",SEL("UNIT",SEL("ELE
        .",
        .SEL("LIST",S)))),S)),UO("CODE",SEL("CODE",SEL("UNIT",SEL("ELE",SEL("LI
        .ST",S)
       .)))))),UO("LIST",U1(SEL("LIST",S),"REPL?ELE",MKCON(APPLIF(SEL("REPL",
        .SEL("ELE",SEL("LIST",S))),S) - 1))))',NUL),
        .MKFLC(4,'NUL',NUL)
        .)),
        .UO('GNDI',U(
        .MKFLC(0,'S','COND(EQU(SEL("LIST",S),NUL),4,1)'),
        .MKFLC(1,'S','COND(EQU(APPLIF(SEL("E2",SEL("ELE",SEL("LIST",S))),S) -
        . APPLIF(SEL("E1",SEL("ELE",SEL("LIST",S))),S),0),2,3)'),
        .MKFLC(2,'U(JO("ITEM",APPLIF(SEL("B",SEL("ELE",SEL("LIST",S))),
        .U1(S,SEL("I",SEL("ELE",SEL("LIST",S))),APPLIF(SEL("E1",SEL("ELE",
        .SEL("LIST",S))),S))),UO("LIST",SEL("NXT",SEL("LIST",S))))',NUL),
        .MKFLC(3,'U(UO("ITEM",APPLIF(SEL("B",SEL("ELE",SEL("LIST",S))),
        .U1(S,SEL("I",SEL("ELE",SEL("LIST",S))),APPLIF(SEL("E1",SEL("ELE",
        .SEL("LIST",S))),S))),UO("LIST",U1(SEL("LIST",S),"E1?ELE",
        .MKCON(APPLIF(SEL("E1",SEL("ELE",SEL("LIST",S))),S) +
        . INCR(APPLIF(SEL("E2",SEL("ELE",SEL("LIST",S))),S) -
        . APPLIF(SEL("E1",SEL("ELE",SEL("LIST",S))),S)     ))))',NUL),
        .MKFLC(4,'NUL',NUL)
        .)))
```

Informally, the function of each of the nodes in the flowchart defining GNLI is as follows:

0. If the list is null, go to "4".
1. Is the replication factor for the first element of the list less than 1?
2. Yes, get next element, go to "3".
3. Return a VO of the form

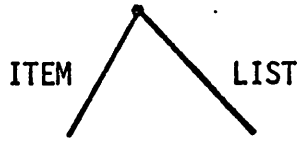
where the component selected by ITEM is the next layout

item, and the component selected by LIST is the "updated" list.

4. Return the null object.

The function of each of the nodes in the flowchart defining GNDI are:

0. If the list is null, go to "4".
1. For the first item on the list, is the index value higher than the upper bound?
2. Yes, return a VO of the form



where the component selected by ITEM is the next data reference, and the component selected by LIST is the "updated" list.

3. Similar to "2", except that the components are calculated differently.
4. Return the null value.

The flowchart corresponding to an Input/Output command is defined by the rule

```
<IO COMMAND>A=<IO INST>B '(L' <$NUMBR>C ',D' <$NUMBR>D ')'
        <NEXT>N
        <LINENO>A↑L=UNO()
        <U>A↑U=U(JO(1,UNO()),UO(2,UNO()),UO(3,UNO()),UO(4,UNO()),UO(5,UNO()),
        .UO(6,UNO()),UO(7,UNO()),UO(8,UNO()),UO(9,UNO()),UO(10,UNO()))
        <FLOWCHART>A↑F=U(A↑X,A↑Y)
        <XFLOWCHART>A↑X=U(
        .MKFLC(A↑L,'U(S,U(UO("L-",APPLI(SEL("GNLI",PROCLIB),    UI(S,"LIST",
        .SEL("L"  C↑VAL,LAYOUTLIB)))) ,UO("D-",APPLI(SEL("GNDI",PROCLIB),
        .SEL("L"  C↑VAL,LAYOUTLIB))))))',SEL(1,A U)),
        .UI(S,"LIST",SEL("D" D VAL,DATALIB))))))',SEL(1,A U)),
        .MKFLC(SEL(1,A↑U),'S','COND(EQU(SEL("ITEM",SEL("L-",S)),NUL),'
        . SEL(2,A↑U) ',' SEL(4,A↑U) ')'),
        .MKFLC(SEL(2,A↑U),'S','COND(EQU(SEL("ITEM",SEL("D-",S)),NUL),A↑N,'
        . SEL(3,A↑U) ')'),
        .MKFLC(SEL(3,A↑U),'UI(S,"L-"  ,APPLI(SEL("GNLI",PROCLIB),UI(S,"LIST",
        .SEL("L"  C↑VAL,LAYOUTLIB))))',SEL(1,A↑U)),
        .MKFLC(SEL(4,A↑U),'S',   'COND(EQU(SEL("CODE",SEL("ITEM",SEL("L-",S))),
        ."D"),' SEL(6,A↑U) ',' SEL(5,A↑U) ')'),
        .MKFLC(SEL(5,A↑U),'B A',SEL(10,A↑U)),
        .MKFLC(SEL(6,A↑U),'S','COND(EQU(SEL("ITEM",SEL("D-",S)),NUL),'
        . SEL(7,A↑U) ',' SEL(8,A↑U) ')'))
        <YYFLOWCHART>A↑Y=U(
        .MKFLC(SEL(7,A↑U),'UI(S,"D-",APPLI(SEL("GNDI",PROCLIB),UI(S,"LIST",
        .SEL("D"  D↑VAL,DATALIB))))',SEL(8,A↑U)),
        .MKFLC(SEL(8,A↑U),'B↑B',SEL(9,A↑U)),
      .U(MKFLC(SEL(9,A↑U),'UI(S,"D-",APPLI(SEL("GNDI",PROCLIB),UI(S,"LIST",
        .SEL("LIST",SEL("D-",S)))))',SEL(10,A↑U)),
        .MKFLC(SEL(10,A↑U),'UI(S,"L-",APPLI(SEL("GNLI",PROCLIB),UI(S,"LIST",
        .SEL("LIST",SEL("L-",S)))))',SEL(1,A↑U))
        .))
```

The functions of each of the nodes in such a flowchart are as follows:

0. Initialize the "variables" L- and D- to the first item on the layout stream and data stream, respectively.
1. Is the layout item null? If not, go to "4".
2. Yes, is the data item null? If so, go to the next statement following the I/O command.
3. Recycle the layout list.
4. Is the layout code "D"?
5. No, make appropriate call to READX or PRINTX (no data is

read or printed here). Go to "10".

6. Yes, is the data item null?
7. Yes, recycle the data list.
8. Make appropriate call to READX or PRINTX.
9. Get next data item.
10. Get next layout item, go to "1".

4.11 Correctness of the Formal Definitions

We would like to have the best evidence obtainable that our formal definitions of the mini-languages are correct. However, to be "correct", the formal definitions must capture the meaning intended by Ledgard, which has only been specified informally; clearly, there is no way we can prove the formal definitions are correct. Evidence for the correctness of the formal definitions is provided as follows: let *L* be a particular mini-language, and let P be a program written in *L*. Then the result of "simulating" P based on the informal description of *L* can be compared with the result of translating P to a set of recursive functional equations and taking the fixed point, i.e., the function determined by the formal definition of *L*. The above comparison was made for each of the sample programs presented in Ledgard's paper (Le 71), and in all cases the results agree*. These results are in Appendix I. What has been shown is that $Sem_L(P)$ agrees with the intuitive notion of the func-

*This process led to the discovery of several errors in the sample programs; these errors were confirmed by Ledgard via private communication and corrected.

tion computed by P for each of the sample programs, and while this does not prove that $Sem_L(P)$ will agree with the intuitive interpretation of P for every program P in *L*, it is the best evidence obtainable that Sem_L provides a reasonable definition of the semantics of *L*.

Chapter 5. Methods of Semantic Definition Compared, and Some Directions for Future Research

Abstract

The use of recursion equations in the definition of programming languages is discussed. The advantages of the method of semantic definition presented in this thesis are pointed out in comparisons with other current methods. Three directions for further research in semantics are discussed.

5.0 Introduction

In our semantic method, Sem_L is defined by (see Section 2.6)

$$Sem_L = Sem_R \circ Trans_{L-R}$$

where $Trans_{L-R}$ translates programs in L to sets of recursive functional equations and Sem_R maps sets of recursive functional equations to their minimal fixed point (see Section 2.4); $Trans_{L-R}$ is defined using synthesized and inherited attributes as in Knuth (Kn 68) and Maurer (Mau 73). The idea of explaining programming language semantics in terms of recursive functional equations is far from new; the following is a partial list of authors that have considered this approach to one degree or another: McCarthy (McC 63a and McC 63b); Landin (La 65); Strachey (St 66); Park (Pa 69); deBakker and Scott (deB 69); Scott (Sc 70); Scott and Strachey (Sc 71); Manna, Ness and Vuillemin (Ma 72); and Cadion and Manna (Ca 72). This list raises two questions: first, why is the use of recursive

functional equations in explaining programming language semantics so popular; and second, what does the present thesis add to the work that has already been done using this approach?

The outstanding quality of sets of recursive functional equations is that they define a "computational process" without reference to a computer, or a computational process of any sort. It is thus possible to define programming languages independent of any machine or abstract machine. The definition of a computational process using a computer or an abstract machine involves keeping track of the state of the computer or computation, which is a forever changing entity. The fixed point characterization of semantics defines the computation by an invariant quantity, namely the fixed point. Invariant entities are easier to handle mathematically than constantly changing ones, which makes the latter type of definition appealing from a mathematical point of view. Manna, Ness, and Vuillemin have exploited the static quality of the semantics of recursive functional equations to prove a number of properties about them in (Ma 71). In fact, McCarthy first suggested this type of proof in (McC 63a).

Another advantage of our method of using recursive functional equations is that all implementation details may be left out of the definition of a language. For example, mini-language 4 permits recursive functions, yet its definition was accomplished without ever describing a stack or a "call and return" mechanism. Arithmetic expressions are evaluated in mini-languages 4, 6, and 7 with no mention of storage needed to store intermediate results. None of

the definitions involve "location counters". These considerations are all properly a part of a computer implementation of a language, but clearly are not essential to the definition of a language, and hence tend to obscure what which is essential if they are present in a definition.

What the previously mentioned papers have failed to contain is any application of this theory to anything resembling a reasonably complete programming language, or any programming language having any complex features other than simple recursion, which, of course, is a very simple feature when explained in terms of recursion equations. For example, McCarthy (McC 63b) shows that a very simple flowchart language can be translated into recursive functionals, but later, when he gives a definition of a small subset of ALGOL in (McC 64), he resorts to an interpreter. In (Sc 71) Scott and Strachey give a complete description of a language so simple it does not have variables! A description of a slightly more powerful (!) language is begun in the same paper, and broken off with the comment, "the reason for passing over this topic is the difficulty of keeping track of the state transformations involved in such a definition". In Manna's papers, he does not attempt to define a "real" programming language, but proves his theorems for a language based directly on recursive functional equations.

In fact, in Ledgard's paper (Le 71) containing the ten mini-languages defined in Chapter 4, Ledgard suggests that the ten mini-languages would be good test cases for a definitional method, and indicates that he does not believe any method current at that time

capable of giving acceptible definitions of all ten languages.

There are two primary factors that make it possible to give complete, clear, and concise definitions of the ten mini-languages, which contain a number of very complex features. First is the assumption of the state vector thesis of McCarthy, that is, the effect of a programming language construct is defined by its effect on the variables it references; and the choice of the Vienna Definition Language subset described in Section 2.2 for defining the necessary state vector operations of variable assignment and referencing. Second was the choice of the "attribute method" for defining $Trans_{L-R}$. This is attributable to Maurer (Mau 73) whose work is in turn based on Knuth's (Kn 68). Actually, the list of alternative formal methods for specifying the translation $Trans_{L-R}$ is very short. A summary of such formal methods is contained in (Ah 72) and it can be seen that most of the research on syntax-directed translation has been done on methods in which the translation at each node depends only on the translation at the nodes immediately below it in the parse tree (its immediate descendents). Expressing this in terms of attributes, only synthesized attributes are used. As can be seen by examining, for example, the attribute equation for the FUNCTION attribute in the rule

```
<VALUE>A=<$NUMBER>B;C=<$LETTER>D
        <VARTABLE>
        <FUNCTION>A↑F=B↑VAL;C↑F='SEL(' SEL('D↑VAL',C↑V) ',S)'
```

in the definition of mini-language 1 in Section 4.1.3. The

function corresponding to a particular nonterminal may be dependent on attribute values defined at nodes above it in the parse tree (in this case the VARTABLE attribute). In fact, this situation occurs in all of the mini-languages. Thus, for all practical purposes, the methods of formally specifying a translation of the restricted type mentioned above would be all but useless for defining the ten mini-languages.

There are also a number of positive advantages in our choice of method for specifying $Trans_{L-R}$, as follows:

1) The translation is defined mathematically without reference to any algorithm for implementing the translation. This eliminates extraneous information in the same way as defining a language by a grammar G rather than by a recognition algorithm.
2) The translation is defined locally; this breaks the translation down into parts that can be easily understood, and modified if necessary.
3) It associates with each node in a parse tree a set of attributes representing the information available at that node. Thus, parts of a program are given a semantic interpretation relative to the entire program.
4) The translation is defined so that it corresponds to the intuitive understanding of the translation process. This is very important as a semantic difinition is only useful if it can be understood with a reasonable amount of effort, hopefully no more than would be required to understand a "language manual" style of definition.

5.1 Comparison with Other Method of Semantic Definition

5.1.1 Compiler-Oriented Methods

Representatives of this method include Wirth and Weber's definition of EULER (Wi 66), along with the compiler-compiler approaches; see, for example, (Fe 66) or (Fe 68). In these methods, Sem_L is defined by

$$Sem_L = Sem_M \circ Trans_{L-M}$$

where $Trans_{L-M}$ translates L to an elementary language M, which may be the machine language for some real machine, or an independently defined "machine-like" language. Sem_M represents the semantics of the elementary language M. Sem_M is either taken to be self-evident, or embodied in some real hardware for a machine that executes M. There are a number of obvious drawbacks to this method. $Trans_{L-M}$ is typically specified procedurally, that is, a procedure is given which implements $Trans_{L-M}$. As has been pointed out before, this introduces a great number of detailed considerations that are extraneous to the definition of L; further, it ties the definition down to a particular parsing algorithm. Also, there is the problem of circularity, as $Trans_{L-R}$ will be in fact a program written in some special language. Note that none of this is true for our specification of $Trans_{L-R}$, as is indicated in the previous Section.

Also, because the "language" of recursive functional equations is much more powerful than a typical machine language, the translation $Trans_{L-R}$ is much simpler than the translation $Trans_{L-M}$. In fact, $Trans_{L-R}(P)$ will probable look very much like $\underline{P}$ itself, whereas

$Trans_{L-M}(P)$ will typically be much larger than (P). Also, $Trans_{L-M}(P)$ will contain all of the implementation details necessary to implement P on a machine (real or abstract).

Sem_R is defined mathematically, while Sem_M is either taken to be self-evident or defined by hardware. Certainly the definition of Sem_R is aesthetically more pleasing. Also, all the methods of proving program correctness, termination, and equivalence, are applicable using Sem_R, whereas their relation to Sem_M is obscure at best.

5.1.2 Interpreter-Oriented Methods

The foremost representative of this method is the Vienna definition of PL/1 (PL 66). In this definition, $Sem_L = Sem_M \circ Trans_{L-M}$ but $Trans_{L-M}$ is the identity, hence $Sem_L = Sem_M$ and the semantics of *L* are embodied in an abstract machine M. Since M "executes" programs in *L*, all the details of a particular implementation, including stack manipulations, call and return mechanisms, parameter passing mechanisms, etc., are present in M. In the compiler oriented definition, the implementaion details are present in the translation process; in the interpreter-oriented method, they are present in the abstract machine. In either case, the user of such a definition must contend with either a translation or an abstract machine containing all of the details of one particular implementation. And this is necessary to understand even the simplest construct in the language. This makes the definition more complex than it needs to be, and makes it impossible to determine which parts of the

definition are describing intrinsic features of the language, and which parts of the definition are present effect a particular implementation. In the method presented in this thesis, it is shown that any one of a particular class of computation rules computes the value calculated by the program, but no particular computation rule or class of computation rules enters into the definition itself.

Using the Vienna method, or any interpretive method, no intrinsic semantics is attached to individual statements in a program. Thus, if c is a statement in a program P, the only way of determining the significance of c is by executing P and, essentially, monitoring the execution of c. That is, the Vienna method defines the semantics of the whole program but does not define the semantics of individual statements. Thus, verification conditions for statements are going to be very difficult to derive for a language defined by the Vienna method. This contrasts with the method of this thesis, which associates with each construct in a program a set of attribute values which constitute the semantics of that construct.

5.1.3 Floyd's Method

Floyd's original techniques have been extended to handle certain programming language features more complex than those considered by Floyd by C.A.R. Hoare (Ho 69 and Ho 71). These techniques have been used by Hoare and N. Wirth to give a formal definition of (a subset of?) PASCAL (Wi 71). Still, whether or not these techniques are sufficiently flexible and powerful to define programming language features as diverse and complex as those in the ten mini-languages remains an open question. Also, if Sem_L is

5.1.2. Floyd's Method

defined using Floyd's method, then $Sem_L(P)$ is not the function computed by P but is the verification condition for P, and this does not seem adequate for definitional purposes. However, this method is very important because it forms the link between programming languages and correctness proofs. Thus, for any semantic method, it is important that verification conditions can be derived for the statements in a program. In Section 5.2.1 it will be shown how verification conditions can be derived using the semantic method of this thesis.

5.1.4 Scott's Method

Scott's method has not been applied to any real or even quasi-real programming languages, although Scott himself has claimed for the last three years that such a definition is right around the corner. Therefore, it is difficult to judge this method. Scott's theory is especially applicable to the definition of programming languages that manipulate functions as data. This requires a little explanation. Mini-languages 4, 5, 6, and 7 ostensibly manipulate functions; functions can be associated with identifiers, functions can be passed as parameters, and functions can be returned as the value of a function call. However, in all of these languages (as is also the case in ALGOL and PL/1) quite a bit is known about these functions before the program is "executed". Thus it is possible to pass information representing the function, such as its name, rather than the function itself. Function names can be treated as ordinary data and do not introduce any difficulties into the semantic

definition using the technique of this thesis. However, if the program can manipulate functions that cannot be represented by their names (this occurs whenever a language allows strings to be compiled and treated as functions during execution of a program, e.g. LISP and SNOBOL4), then the domain of our recursive functional equations must be expanded to include functions, and the mathematical theory breaks down because the domain has an unbounded cardinality and the fixed point theorem is no longer applicable. Scott (Sc 70b) has shown the existence of a domain that does contain all the functions that are computable by computer programs, and for which the fixed point theorem does apply. This domain, however, can also be used as the domain for the recursive functional equations defined by our semantic method, with the same result; namely, that semantic definitions can be given for these languages.

The difference between our method and Scott's is that he defines Sem_L itself as the fixed point of a set of recursive functional equations defined on the syntactic classes of L (see Section 1.8), and our method defines Sem_L by the equation $Sem_L = Sem_R \circ Trans_{L-R}$. It seems likely that Scott's method for defining Sem_L will be completely intractible for any real programming language, because many features of programming languages are primarily for the user's convenience, and Scott's method has only been applied thus far to languages designed for its own convenience. That is, somehow all of the information embodied in the equations defining $Trans_{L-R}$ must be expressed in the semantic equations defining Sem_L in Scott's method; and at the present, its not clear to the author of this thesis how it can be done.

5.2 Directions for Future Research

5.2.1 Program Correctness

In order to prove that a program P is correct, verification conditions $V_c(Q,R)$ must be derived for each command c in P (see Section 1.4 for a discussion of Floyd's initial paper on this subject. See also (Ho 69) and (Ho 70)). In this section, we will show that Floyd's verification condition for a command of the form

$$x \leftarrow f(x,y_1,y_2, \ldots y_n)$$

namely

$$V_{x \leftarrow f(x,y_1, \ldots y_n)}(Q,R) = \lfloor x_0(x=f(x_0,y_1, \ldots y_n) \wedge Q(x_0,y_1 \cdot \ldots y_n)) \Rightarrow R(x_0,y_1, \ldots y_n)$$

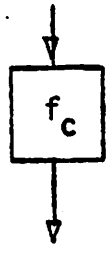
can be derived from the semantic definition of the command $x \leftarrow f(x,y_1, \ldots y_n)$ that is given by the semantic method of this thesis. The state vector function that would be associated with $x \leftarrow f(x,y_1, \ldots y_n)$ by a formal definition of a language containing this type of statement would have the form

$$f_c = U1(S,x,f(SEL(x,S),SEL(y_1,S), \ldots SEL(y_n,S)))$$

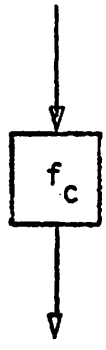
and the functional equation associated with c will have the form (see Section 2.4)

$$F_c = F_d f_c$$

where d is the statement following c, this equation has the following representation as a node in a flowchart



Let $Q(x,y_1, \ldots y_n)$ and $R(x,y_1, \ldots y_n)$ be assertions about the variables $x, y_1, \ldots y_n)$ before execution of $\underline{c}$ and after execution of $\underline{c}$, respectively; then verifying the path consisting of the single statement $\underline{c}$ corresponds to verifying the following path in the flowchart representation of $\underline{c}$



$\leftarrow Q(SEL(x,S),SEL(y_1,S), \ldots SEL(y_n,S))$

$=\overline{Q}$

$\leftarrow R(SEL(x,S),SEL(y_1,S), \ldots SEL(y_n,S))$

$=\overline{R}$

$\overline{Q}$ partitions the set of all state vectors S into two disjoint sets,

$$\overline{Q}_T = \{S \varepsilon S \mid \overline{Q}(S) = \text{TRUE}\}$$

$$\overline{Q}_F = \{S \varepsilon S \mid \overline{Q}(S) = \text{FALSE}\}$$

and similarly, $\overline{R}$ partitions S into two disjoint sets

$$\overline{R}_T = \{S \varepsilon S \mid \overline{R}(S) = \text{TRUE}\}$$

$$\overline{R}_F = \{S \varepsilon S \mid \overline{R}(S) = \text{FALSE}\}$$

Verifying the above path consists of showing whenever f_c is applied to a state vector $\underline{S}$ for which $\overline{Q}$ is true, then $\overline{R}(f_c(S))$ is true, i.e.

$$* \quad f_c(\overline{Q}_T) \subseteq \overline{R}_T$$

This verification condition follows immediately from our semantic definition of the statement $x \leftarrow f(x,y_1, \ldots y_n)$; we will show that it is equivalent to the verification condition derived by Floyd,

$$** \quad \exists x_0(x=f(x_0,y_1, \ldots ,y_n) \wedge Q(x_0,y_1, \ldots ,y_n) \Rightarrow R(x,y_1, \ldots y_n))$$

** => *

Suppose $\overline{Q}(S)$ = TRUE,

let α_0=SEL(x,S), γ_1=SEL(y_1,S), ... ,γ_n=SEL(y_n,S),

and let α=f(α_0,γ_1, ... ,γ_n).

The hypotheses of ** are satisfied, therefore

$R(\alpha,\gamma_1, \ldots ,\gamma_n)$ = TRUE,

so $R(f(\alpha_0,\gamma_1, \ldots ,\gamma_n),\gamma_1, \ldots ,\gamma_n)$ = TRUE,

and $\overline{R}(f_c(S))$ = TRUE

* => **

Suppose $\exists\alpha_0$ such that $\alpha=f(\alpha_0,\gamma_1, \ldots ,\gamma_h)$ and $Q(\alpha_0,\gamma_1, \ldots ,\gamma_n)$.

Let S be a state vector such that

α_0=SEL(x,S), γ_1=SEL(y_1,S), ... ,γ_n=SEL(y_n,S)

then $\overline{Q}(S)$ = TRUE and the hypothesis of * is satisfied,

therefore

$\overline{R}(S)$ = TRUE,

so $R(f(\alpha_0,\gamma_1, \ldots ,\gamma_n),\gamma_1, \ldots ,\gamma_n)$ = TRUE,

and $R(\alpha,\gamma_1, \ldots ,\gamma_n)$ = TRUE

The reason this proof is trivial is that our definition of the semantics of $x \leftarrow f(x,y_1, \ldots ,y_n)$ corresponds to the intuitive notion from which Floyd originally "derived" his verification condition.

5.2.2 Proving Assertions About Translations

The above discussion was informal, and no formal study of the equivalence of the verification condition semantics of Floyd and the functional semantics of our method has been made. One approach to a formal proof of equivalence would be to associate verification

conditions with constructs in a language as attributes defined by attribute equations. Thus, associated with a program $P \in L$ we would have not only

$$\text{Trans}_{L\text{-}R}(P) \quad \text{but also } V_p(Q,R)$$

where $V_p(Q,R)$ is the verification for the entire program. To show the equivalence of the functional definition and the definition by verification would amount to showing

$$*** \quad \text{Sem}_R \circ \text{Trans}_{L\text{-}R}(Q_T) \subseteq R_T \iff V_p(Q,R)$$

where $\underline{Q}$ and $\underline{R}$ are assertions about the input values and output values of $\underline{P}$ respectively, and

$$Q_T = \{\text{input value } \alpha \mid Q(\alpha) = \text{TRUE}\}$$

$$R_T = \{\text{output value } \alpha \mid R(\alpha) = \text{TRUE}\}$$

The method of proof that appears to be most reasonable for proving *** consists of making assertions about the relations that hold for the various attributes defined for the nonterminals in the grammar generating L. In particular, for any nonterminal having a functional attribute representing the action of the construct, and an attribute representing the verification condition for the construct, their equivalence could be formulated. Once the necessary assertions relating the attributes have been formulated, it must be shown that the attribute equations defining the values of the attributes preserve the truth of the assertions. Thus, if we have an attribute equation of the form

$$f_\alpha : V_{\alpha_1} \times \ldots \times V_{\alpha_t} \rightarrow V_\alpha$$

associated with the production

$$X_{p_0} \rightarrow X_{p_1} X_{p_2} \ldots X_{p_n}$$

then any assertions about the attribute value α must be derivable from the assertions for the attributes $\alpha_1, \alpha_2, \ldots, \alpha_t$ and the function f_α which defines α.

If the assertions are true for constant-valued attributes, then each attribute value defined by the attribute equations will satisfy its assertions, and hence the assertions relating the attributes of the root node can be proved.

5.2.3 Applications of Recursive Function Theory

In the definition $Sem_L = Sem_R \circ Trans_{L-R}$, the translation $Trans_{L-R}$ can be regarded as making "notational" changes to the programs in L; the semantics of "execution" is given primarily be Sem_R. A number of techniques for proving correctness, termination and equivalence, for sets of recursive functional equations are developed in a paper by Manna, Ness, and Vuillemin (Ma 72). These techniques are applicable to any programming language L for which $Trans_{L-R}$ is defined. Since recursive function theory has been a (major ?) branch of mathematics for at least twenty years, it seems reasonable to expect that there are a number of results that are relevant to the theory of programming languages. This appears to be an important area for future research.

BIBLIOGRAPHY

Ah 72 Aho, A. V., and Ullman, J. D. Theory of Parsing, Translating, and Compiling. Vol. II. Inglewood, New Jersey: Prentice-Hall, 1972.

Ca 72 Cadiou, J. M., and Manna, Z. "Recursive Definitions of Partial Functions and Their Computations," in Proceedings of an ACM Conference on Proving Assertions About Programs. New York: The Association for Computing Machinery (ACM), 1972, pp. 58-65.

Ch 57 Chomsky, N. Syntactic Structures. The Hague, Netherlands: Moulton and Co., 1957.

deB 69 deBakker, J. W., and Scott, D. "A Theory of Programs." Unpublished Memorandum, August 1969.

Fa 72 Fang, I. "FOLDS: A Declarative Formal Language Definition System". Ph.D. dissertation, Stanford University, 1972.

Fe 66 Feldman, J. "A Formal Semantics for Computer Languages and Its Application to a Compiler-Compiler," Communications of the ACM, IX (January, 1966).

Fe 68 Feldman, J., and Gries, D. "Translator Writing Systems," Communications of the ACM (February, 1968)., 77-113.

Fl 67 Floyd, R. W. "Assigning Meaning to Programs," in Proceedings of a Symposium in Applied Mathematics. Vol. XIX: Mathematical Aspects of Computer Science (J. T. Schwartz, ed.). American Mathematical Society, 1967, pp. 19-32.

Gr 71 Gries, D. Compiler Construction for Digital Computers. New York: John Wiley and Sons, 1971.

Ho 69 Hoare, C. A. R. "An Axiomatic Approach to Computer Programming," Communications of the ACM, X (October, 1969), 576-580.

Ho 71 Hoare, C. A. R. "Procedures and Parameters: an Axiomatic Approach," in Symposium on Semantics of Algorithmic Languages (E. Engeler, ed.). Springer-Verlag, 1971, pp. 102-116.

Ho 73 Hoare, C.A.R. and Wirth, N. An Axiomatic Definition of PASCAL, Acta Informatica, Springer-Verlag. (to appear).

Kl 52 Kleene, S. C. Introduction to Meta-Mathematics. Princeton, N. J.: Van Nostrand, 1952.

Kn 68 Knuth, D. E. "Semantics of Context-Free Languages," Mathematical Systems Theory, II, No. 2 (1968), 127-145.

La 65 Landin, P. J. "A Correspondence Between ALGOL 60 and Church's Lambda-Notation," Communications of the ACM, VIII (February and March, 1965), 89-101 and 158-165.

Le 71 Ledgard, H. F. "Ten Mini-Languages: A Study of Topical Issues in Programming Languages," Computer Surveys, III, No. 3 (September, 1971), 117-146.

Lu 67 Lucas, P. Introduction to the Method Used for the Formal Definition of PL/1. (IBM Vienna Laboratory Technical Report TR 25.081) IBM, October 1967.

Ma 72 Manna, Z., Ness, S., and Vuillemin, J. "Inductive Methods for Proving Properties of Programs," in Proceedings of an ACM Conference on Proving Assertions About Programs. New York: The Association for Computing Machinery (ACM), 1972.

Mau 73 Maurer, W. D. "A Semantic Extension of BNF," International Journal of Computer Mathematics, 1972.

McC 60 McCarthy, J. The LISP Programmer's Manual. Boston: M.I.T. Computation Center, 1960.

McC 63a McCarthy, J. "A Basis for a Mathematical Theory of Computation," in Computer Programming and Formal Systems (P. Braffort and D. Hirschberg, eds.). New York: Humanities Press, 1963, pp. 33-70.

McC 63b McCarthy, J. "Towards a Mathematical Science of Computation," in Proceedings of the IFIP Congress. Amsterdam: North Holland Publishing Co., 1963, pp. 21-28.

McC 64 McCarthy, J. "A Formal Description of a Subset of ALGOL," in Formal Language Description Languages for Computer Programming (Proceedings of the IFIP Working Conference). Amsterdam: North Holland Publishing Co., 1964, pp. 1-12.

Na 60 Naur, P. (ed.). "Report on the Algorithmic Language ALGOL 60," Communications of the ACM, VI (May, 1960), 299-314.

Pa 69 Park, D. "Fixpoint Induction and Proofs of Program Properties," in Machine Intelligence 5 (B. Meltzer and D. Michie, eds.). Edinburgh: Edinburgh University Press, 1969, pp. 59-78.

PL 66 PL/1 Definition Group of the Vienna Laboratory. Formal Definition of PL/1 (Universal Language Document No. 3, Technical Report TR 25.071) Vienna: IBM Laboratory, December 1966.

Re 73 Reynolds, J. "Definitional Interpreters for Higher-Order Programming Languages," Communications of the ACM (to appear).

Sc 70a Scott, D. "Outline of a Mathematical Theory of Computation," in Proceedings of the Fourth Annual Princeton Conference of Information Science and Systems. Princeton, N. J.: Princeton University Press, 1970, pp. 164-176.

Sc 70b Scott, D. "Lattice-Theoretic Models for the λ-Calculus." Unpublished Monograph, Princeton University, 1970.

Sc 71 Scott, D., and Strachey, C. Towards a Mathematical Semantics for Computer Languages. (Technical Monograph PRG-6) Oxford, England: Oxford University Press, August 1971.

St 66 Strachey, C. "Towards a Formal Semantics," in Proceedings of the IFIP Working Conference. Amsterdam: North Holland Publishing Co., 1966, pp. 198-220.

vanW 66 van Wijngaarden, A. "Recursive Definition of Syntax and Semantics," in Formal Language Description Languages for Computer Programming (T. B. Steel, ed.). Amsterdam: North Holland Publishing Co., 1966.

Vu 72 Vuillemin, J. "Proof Techniques for Recursive Programs". Ph.D. dissertation, Stanford University, 1972.

We 72 Wegner, P. "The Vienna Definition Language," Computing Surveys, IV, No. 1 (March, 1972), 5-63.

Wil 70 Wilner, W. T. "A Formal Definition of SIMULA 67". Ph.D. dissertation, Stanford University, 1970.

Wi 66 Wirth, N., and Weber, H. "EULER, A Generalization of ALGOL, and its Formal Definition," Communications of the ACM, IX (January and February, 1966), 12-23 and 89-99.

Wi 71 Wirth, N. "The Programming Language PASCAL," Acta-Information 1. Vol. III. Springer-Verlag, 1971.

APPENDIX I

PROGRAMS RUN ON THE SEMANTICS-BASED INTERPRETER

MINI-LANGUAGE 1

SYNTACTIC AND SEMANTIC DESCRIPTION

```
<PROGRAM>A=<BLOCK>B
        <FLOWCHART>A↑F=B↑F
        <SYMBOLTABLE>A↑S=B↑V
        B↑N='NUL';B↑G='NUL'

<BLOCK>A='BEGIN ' <DECLARATION>B ';' <COMMAND LIST>C ';END'
        <NEXT>N
        <GLOBALS>G
        <VARTABLE>A↑V=U(A↑G,B↑V)
        <FLOWCHART>A↑F=C↑F
        <STATENO>A↑S=C↑S
        C↑N=A↑N;C↑V=A↑V;C↑L=C↑X

<DECLARATION>A='IDEN ' <VARLIST>B
        <VARTABLE>A↑V=B↑V

<VARLIST>A=<$LETTR>B ',' <VARLIST>C;D=<$LETTR>E
        <VARTABLE>A V=U(UO('B VAL',UNO()),C V);D V=UO('E VAL',UNO())

<COMMAND LIST>A=<COMMAND>B ';' <COMMAND LIST>C;D=<COMMAND>E
        <NEXT>N
        <VARTABLE>V
        <LABELTABLE>L
        <XLABELS>A↑X=U(B↑X,C↑X);D↑X=E↑X
        <FLOWCHART>A↑F=U(B↑F,C↑F);D↑F=E↑F
        <STATENO>A↑S=B↑S;D↑S=E↑S
        B↑V=A↑V;B↑N=C↑S;B↑L=A↑L
        C↑V=A↑V;C↑N=A↑N;C↑L=A↑L
        E↑V=D↑V;E↑N=D↑N;E↑L=D↑L

<COMMAND>A=<LABEL>B ' ' <UNLABELLED COMMAND>C;D=<UNLABELLED COMMAND>E
        <NEXT>N
        <VARTABLE>V
        <LABELTABLE>L
        <XLABELS>A↑X=UO('B↑V',C↑S);D↑X='NUL'
        <FLOWCHART>A↑F=C↑F;D↑F=E↑F
        <STATENO>A↑S=C↑S;D↑S=E↑S
        C↑N=A↑N;C↑V=A↑V;C↑L=A↑L
        E↑N=D↑N;E↑V=D↑V;E↑L=D↑L

<LABEL>A='L' <$NUMBR>B
        <VALUE>A↑V='L' B↑VAL

<UNLABELLED COMMAND>A=<$LETTR>B '=' <VALUE>C;
.D='HOPTO ' <LABEL>E;
.F=<BLOCK>G
        <NEXT>N
        <VARTABLE>V
        <LABELTABLE>L
        <STATENO>A↑S=UNO();D S=UNO();F↑S=G↑S
        <FLOWCHART>A↑F=UO(A↑S,U(UO('FUN','U(S,UO(' SEL('B↑VAL',A↑V) ',C↑F))'),
         .UO('NXT',A↑N)));
         .D F=JO(D↑S,U(UO('FUN','S'),UO('NXT',SEL('E↑V'  ,D↑L))));
         .F F=G↑F
        C↑V=A↑V
```

```
APPLY FLOWCHART TO NULL STATE VECTOR S


          FINAL STATE VECTOR
PLIST-    ----------
1     57
2     58

X     3
I     6
AR    ----------
3     6
6     3


TIME TO PROCESS SEMANTIC DESCRIPTION 1839 MILLISEC
TIME TO PARSE 2730 MILLISEC
TIME TO EVALUATE SEMANTIC ATTRIBUTES 31292 MILLISEC
'EXECUTION' TIME 180 MILLISEC
'STATEMENTS' EXECUTED 10
```

```
        G↑N=F↑N;G↑G=F↑V

<VALUE>A=<$NUMBR>B;C=<$LETTR>D
        <VARTABLE>V
        <FUNCTION>A↑F=B↑VAL;C↑F='SEL(' SEL:'D↑VAL',C↑V) ',S)'
END
```

SOURCE PROGRAM

```
BEGIN IDEN A,B;
  B=2;
  A=1;
  BEGIN IDEN B,C;
    B=A;
    C=7;
  END;
  A=3;
  B=3;
END
```

```
LISTING OF ATTRIBUTES OF PROGRAM NODE

          DISPLAY OF FLOWCHART-F
5     ----------
FUN     U(S,U0(1,2))
NXT     6

6     ----------
FUN     U(S,U0(3,1))
NXT     11

11    ----------
FUN     U(S,U0(9,SEL(3,S)))
NXT     12

12    ----------
FUN     U(S,U0(7,7))
NXT     13

13    ----------
FUN     U(S,U0(3,3))
NXT     14

14    ----------
FUN     U(S,U0(1,3))
NXT


          DISPLAY OF SYMBOLTABLE-S
A     3
B     1
```

```
APPLY FLOWCHART TO NULL STATE VECTOR S


          FINAL STATE VECTOR
9     1
7     7
3     3
1     3

TIME TO PROCESS SEMANTIC DESCRIPTION 971 MILLISEC
TIME TO PARSE 824 MILLISEC
TIME TO EVALUATE SEMANTIC ATTRIBUTES 12058 MILLISEC
'EXECUTION' TIME 62 MILLISEC
'STATEMENTS' EXECUTED 6
```

SOURCE PROGRAM

```
BEGIN IDEN A,B;
  A=1;
  B=2;
  BEGIN IDEN B,C;
    B=A;
    C=7;
  END;
  HOPTO L1;
  A=3;
L1 B=3;
END
```

```
LISTING OF ATTRIBUTES OF PROGRAM NODE

          DISPLAY OF FLOWCHART-F
32      ----------
FUN      U(S,U0(30,1))
NXT      33

33      ----------
FUN      U(S,U0(28,2))
NXT      38

38      ----------
FUN      U(S,U0(36,SEL(30,S)))
NXT      39

39      ----------
FUN      U(S,U0(34,7))
NXT      40

40      ----------
FUN      S
NXT      42

41      ----------
FUN      U(S,U0(30,3))
NXT      42

42      ----------
FUN      U(S,U0(28,3))
NXT


          DISPLAY OF SYMBOLTABLE-S
A      30
B      28
```

```
APPLY FLOWCHART TO NULL STATE VECTOR S


FINAL STATE VECTOR
30      1
36      1
34      7
28      3

TIME TO PROCESS SEMANTIC DESCRIPTION 971 MILLISEC
TIME TO PARSE 1059 MILLISEC
TIME TO EVALUATE SEMANTIC ATTRIBUTES 13833 MILLISEC
'EXECUTION' TIME 62 MILLISEC
'STATEMENTS' EXECUTED 6
```

MINI-LANGUAGE 2

SYNTACTIC AND SEMANTIC DESCRIPTION

```
<PROGRAM>A=<ASSIGNMENT LIST>B ';END'
         <FLOWCHART>A↑F=B↑F
         B↑N='NUL'

<ASSIGNMENT LIST>A=<ASSIGNMENT COMMAND>B ';' <ASSIGNMENT LIST>C;
.D=<ASSIGNMENT COMMAND>E
         <NEXT>N
         <FLOWCHART>A↑F=U(B↑F,C↑F);D↑F=E↑F
         <LINEND>A↑L=B↑L;D↑L=E↑L
         B↑N=C↑L
         C↑N=A↑N
         E↑N=D↑N

<ASSIGNMENT COMMAND>A=<LEFT HAND EXPRESSION>B '=' <RIGHT HAND EXPRESSION>C
         <NEXT>N
         <LINEND>A↑L=UNO()
         <FLOWCHART>A↑F=UO(A↑L,U(UO('FUN','U1(S,B↑L,C↑R)'),UO('NXT',A↑N)))

<LEFT HAND EXPRESSION>A=<$LETTR>B;
.C='?' <LEFT HAND EXPRESSION>D
         <LHSFUN>A↑L='"B↑VAL"';
         .C↑L='SEL(D↑L,S)'

<RIGHT HAND EXPRESSION>A=<LITERAL>B;
.C=<$LETTR>D;
.E='?' <RIGHT HAND EXPRESSION>F
         <RHSFUN>A↑R=B↑V;
         .C↑R='SEL("D↑VAL",S)';
         .E↑R='SEL(F↑R,S)'

<LITERAL>A='(' <$LETTR>B ')'
         <VALUE>A↑V='"B↑VAL"'
END
```

SOURCE PROGRAM

```
X=(A);
Y=(X);
Z=Y;
?Z=(B);
END
```

LISTING OF ATTRIBUTES OF PROGRAM NODE

```
          DISPLAY OF FLOWCHART-F
1       ----------
FUN       U1(S,"X","A")
NXT       2

2       ----------
FUN       U1(S,"Y","X")
NXT       3

3       ----------
FUN       U1(S,"Z",SEL("Y",S))
NXT       4

4       ----------
FUN       U1(S,SEL("Z",S),"B")
NXT
```

APPLY FLOWCHART TO NULL STATE VECTOR S

FINAL STATE VECTOR

```
Y     X
Z     X
X     B
```

```
TIME TO PROCESS SEMANTIC DESCRIPTION 502 MILLISEC
TIME TO PARSE 342 MILLISEC
TIME TO EVALUATE SEMANTIC ATTRIBUTES 2177 MILLISEC
'EXECUTION' TIME 48 MILLISEC
'STATEMENTS' EXECUTED 4
```

SOURCE PROGRAM

```
X=(A);
Y=?X;
END
```

LISTING OF ATTRIBUTES OF PROGRAM NODE

DISPLAY OF FLOWCHART-F

```
12
FUN   U1(S,"X","A")
NXT   13

13
FUN   U1(S,"Y",SEL(SEL("X",S),S))
NXT
```

```
APPLY FLOWCHART TO NULL STATE VECTOR S


FINAL STATE VECTOR
X     A
Y

TIME TO PROCESS SEMANTIC DESCRIPTION 502 MILLISEC
TIME TO PARSE 339 MILLISEC
TIME TO EVALUATE SEMANTIC ATTRIBUTES 955 MILLISEC
'EXECUTION' TIME 113 MILLISEC
'STATEMENTS' EXECUTED 2
```

MINI-LANGUAGE 3

SYNTACTIC AND SEMANTIC DESCRIPTION

```
<PROGRAM>A=<BLOCK>B
        <FLOWCHART>A↑F=B↑F
        <LABELTABLE>A↑L=B↑X
        <SYMBOLTABLE>A↑S=B↑V
        B↑N='NUL';B↑G='NUL';B↑L=B↑X

<BLOCK>A='BEGIN ' <DECLARATION>B ';' <COMMAND LIST>C ';END'
        <NEXT>N
        <GLOBALS>G
        <LABELTABLE>L
        <VARTABLE>A↑V=U(A↑G,B↑V)
        <XLABELS>A↑X=C↑X
        <FLOWCHART>A↑F=C↑F
        <STATENO>A↑S=C↑S
        C↑N=A↑N;C↑V=A↑V;C↑L=A↑L

<DECLARATION>A='IDEN ' <VARLIST>B
        <VARTABLE>A↑V=B↑V

<VARLIST>A=<$LETTR>B ',' <VARLIST>C;D=<$LETTR>E
        <VARTABLE>A↑V=U(UO('B↑VAL',UNO()),C↑V);D↑V=UO('E↑VAL',UNO())

<COMMAND LIST>A=<COMMAND>B ';' <COMMAND LIST>C;D=<COMMAND>E
        <VARTABLE>V
        <NEXT>N
        <LABELTABLE>L
        <XLABELS>A↑X=U(B↑X,C↑X);D↑X=E↑X
        <FLOWCHART>A↑F=U(B↑F,C↑F);D↑F=E↑F
        <STATENO>A↑S=B↑S;D↑S=E↑S
        B↑V=A↑V;B↑N=C↑S;B↑L=A↑L
        C↑V=A↑V;C↑N=A↑N;C↑L=A↑L
        E↑V=D↑V;E↑N=D↑N;E↑L=D↑L

<COMMAND>A=<LABEL>B ' ' <UNLABELLED COMMAND>C;D=<UNLABELLED COMMAND>E
        <VARTABLE>V
        <NEXT>N
        <LABELTABLE>L
        <XLABELS>A↑X=UO('B↑V',C↑S);D↑X=E↑X
        <FLOWCHART>A↑F=C↑F;D↑F=E↑F
        <STATENO>A↑S=C↑S;D↑S=E↑S
        C↑N=A↑N;C↑V=A↑V;C↑L=A↑L
        E↑N=D↑N;E↑V=D↑V;E↑L=D↑L

<LABEL>A='L' <$NUMBR>B
        <VALUE>A↑V='L' B↑VAL

<UNLABELLED COMMAND>A=<$LETTR>B '=' <VALUE>C;
.D='GOTO ' <LABEL REF>E;
.F=<BLOCK>G
        <NEXT>N
        <VARTABLE>V
        <LABELTABLE>L
        <STATENO>A↑S=UNO();D↑S=UNO();F↑S=G↑S
        <XLABELS>A↑X='NUL';D↑X='NUL';F↑X=G↑X
        <FLOWCHART>A↑F=UO(A↑S,U(UO('FUN','U(S,UO(' SEL('B↑VAL',A↑V) ',C↑F))'),
```

```
            .UO('NXT',A↑N)));
        .D F=UO(D↑S,U(UO('FUN','S'),UO('NXT','E F'))));
            .F↑F=G↑F
            C↑V=A↑V;C↑L=A↑L
            E↑V=D↑V;E↑L=D↑L
            G↑N=F↑N;G↑G=F↑V;G↑L=F↑L

<LABEL REF>A=<LABEL>B;C=<$LETTR>D
            <VARTABLE>V
            <LABELTABLE>L
            <FUNCTION>A F=SEL('B↑V',A↑L);C↑F='SEL(SEL(' SEL('D↑VAL',C↑V)
            . ',S),LABELTABLE)'

<VALUE>A=<$NUMBR>B;C=<$LETTR>D;E=<LABEL>F
            <VARTABLE>V
            <LABELTABLE>L
            <FUNCTION>A↑F=B↑VAL;C↑F='SEL(' SEL('D↑VAL',C↑V) ',S)';
            .E↑F='"F↑V"'
END
```

SOURCE PROGRAM

```
BEGIN IDEN A,B;
      A=L1;
      B=2;
      BEGIN IDEN B,C;
            B=3;
            C=4;
            GOTO A;
      END;
      A=6;
   L1 A=1;
END
```

```
LISTING OF ATTRIBUTES OF PROGRAM NODE

          DISPLAY OF FLOWCHART-F
5       ----------
FUN       U(S,UO(3,"L1"))
NXT       6

6       ----------
FUN       U(S,UO(1,2))
NXT       11

11      ----------
FUN       U(S,UO(9,3))
NXT       12

12      ----------
FUN       U(S,UO(7,4))
NXT       13

13      ----------
FUN       S
NXT       SEL(SEL(3,S),LABELTABLE)

14      ----------
FUN       U(S,UO(3,6))
NXT       15

15      ----------
FUN       U(S,UO(3,1))
NXT


          DISPLAY OF LABELTABLE-L
L1        15

          DISPLAY OF SYMBOLTABLE-S
A         3
B         1
```

```
APPLY FLOWCHART TO NULL STATE VECTOR S


FINAL STATE VECTOR
1     2
9     3
7     4
3     1

TIME TO PROCESS SEMANTIC DESCRIPTION 1165 MILLISEC
TIME TO PARSE 953 MILLISEC
TIME TO EVALUATE SEMANTIC ATTRIBUTES 15837 MILLISEC
'EXECUTION' TIME 65 MILLISEC
'STATEMENTS' EXECUTED 6
```

SOURCE PROGRAM

```
BEGIN IDEN A,B;
      B=2;
      BEGIN IDEN B,C;
            B=3;
            A=L3;
            GOTO L1;
         L3 C=4;
            GOTO L2;
      END;
      A=6;
   L1 GOTO A;
   L2 A=1;
END
```

```
LISTING OF ATTRIBUTES OF PROGRAM NODE

          DISPLAY OF FLOWCHART-F
40      ----------
FUN      U(S,UO(36,2))
NXT      45

45      ----------
FUN      U(S,UO(43,3))
NXT      46

46      ----------
FUN      U(S,UO(38,"L3"))
NXT      47

47      ----------
FUN      S
NXT      56

48      ----------
FUN      U(S,UO(41,4))
NXT      50

50      ----------
FUN      S
NXT      58

55      ----------
FUN      U(S,UO(38,6))
NXT      56

56      ----------
FUN      S
NXT      SEL(SEL(38,S),LABELTABLE)

58      ----------
FUN      U(S,UO(38,1))
NXT


          DISPLAY OF LABELTABLE-L
L3      48
L1      56
L2      58


          DISPLAY OF SYMBOLTABLE-S
A      38
B      36
```

```
APPLY FLOWCHART TO NULL STATE VECTOR S


FINAL STATE VECTOR
36      2
43      3
41      4
38      1

TIME TO PROCESS SEMANTIC DESCRIPTION 1165 MILLISEC
TIME TO PARSE 1257 MILLISEC
TIME TO EVALUATE SEMANTIC ATTRIBUTES 20018 MILLISEC
*EXECUTION* TIME 81 MILLISEC
*STATEMENTS* EXECUTED 8
```

MINI-LANGUAGE 4

SYNTACTIC AND SEMANTIC DESCRIPTION

```
4

<PROGRAM>A=<EXPRESSION>B ';END'
         <FLOWCHART>A↑F=UO(O,U(UO('FUN','B↑F'),UO('NXT',)))
         <MFUNLIB>A↑M=B↑M
         <EXPLIB>A↑E=B↑E
         B↑D='NUL'

<EXPRESSION>A=<$NUMBR>B;
.C=<LET EXPRESSION>D;
.E=<COMBINATION>F;
.G=<IDENT>H
          <DECLNAMES>D
          <MFUNLIB>A↑M='NUL';C↑M=D↑M;E↑M=F↑M;G↑M='NUL'
          <EXPLIB>A↑E='NUL';C↑E=D↑E;E↑E=F↑E;G↑E='NUL'
          <FUNCTION>A↑F=B↑VAL;C↑F=D↑F;E↑F=F↑F;G↑F=H↑F
          D↑D=C↑D
          F↑D=E↑D
         H↑D=G↑D

<IDENT>G=<$ALPHA>H
         <DECLNAMES>D
         <FUNCTION>G↑F=COND(EQU(SEL('TYPE',SEL('H↑VAL',G↑D)),'FUN'),
         .'U(UO("NAME","H↑VAL"),UO("ENV",SEL("ENV",S)))',
          .COND(EQU(SEL('TYPE',SEL('H↑VAL',G↑D)),'EXP'),
         .'APPLI(SEL("H↑VAL",MFUNLIB),S)',
         .'SEL(' SEL('NO',SEL('H↑VAL',G↑D)) ',SEL(SEL("' SEL('FNAME',
         .SEL('H↑VAL',G↑D)) '",SEL("ENV",S)),S))'))

<LET EXPRESSION>A='LET ' <$ALPHA>B '=' <EXPRESSION>C 'IN ' <EXPRESSION>D;
.E='LET ' <$ALPHA>F '(' <PARAM LIST>G ')=' <EXPRESSION>H 'IN ' <EXPRESSION>I
         <DECLNAMES>D
         <FUNCTION>A↑F=D↑F;E↑F=I↑F
        <MFUNLIB>A↑M=U(UO('B↑VAL',UO(O,U(UO('FUN','C↑F'),UO('NXT',NUL)))))
         .,D↑M,C↑M);
         .E↑M=U(UO('F↑VAL',UO(O,U(UO('FUN','H↑F'),UO('NXT',NUL)))))
         .,U(H↑M,I↑M))
         <EXPLIB>A↑E=U(C↑E,D↑E);E↑E=U(H↑E,I↑E)
         C↑D=U(A↑D,UO('B↑VAL',UO('TYPE','EXP')));
         .D↑D=C↑D;
         .H↑D=U(E↑D,U(G↑D,UO('F↑VAL',UO('TYPE','FUN'))));
         .I↑D=H↑D
         G↑I=1;G↑N='F↑VAL'

<PARAM LIST>A=<$ALPHA>B ',' <PARAM LIST>C;D=<$ALPHA>E
         <NAME>N
         <INDEX>I
         <DECLNAMES>A↑D=U(C↑D,UO('B↑VAL',U(U(UO('TYPE','PARAM'),UO('NO',A↑I)),
         .UO('FNAME',A↑N))));
         .D↑D=UO('E↑VAL',U(U(UO('TYPE','PARAM'),UO('NO',D↑I)),UO('FNAME',
         .'D↑N')))
         C↑N=A↑N;C↑I=A↑I + 1

<COMBINATION>A='(' <EXPRESSION>B '+' <EXPRESSION>C ')';
.D='(IF ' <EXPRESSION>E '>' <EXPRESSION>F ' THEN ' <EXPRESSION>G ' ELSE '
```

```
.<EXPRESSION>M ')';
.L=<IDENT>M '(' <EXPRESSION LIST>N ')'
          <DECLNAMES>D
           <UNO>A↑U='NUL';D↑U=UNO();L↑U='NUL'
          <MFUNLIB>A↑M=U(B↑M,C↑M);
          .D↑M=U(U(U(E↑M,F↑M),G↑M),H↑M);
          .L↑M=N↑M
          <EXPLIB>A↑E=U(B↑E,C↑E);
          .D E=U(U(U(E↑E,F↑E),G↑E),H↑E,
           .UO('SELECT' D↑U,U(MKFLC(0,'S','COND(GTN(E↑F ,F↑F),1,2)'),
          .MKFLC(1,'G↑F',),MKFLC(2,'H↑F',))));
          .L↑E=N↑E
          <FUNCTION>A↑F='B↑F + C↑F';
           .D↑F='APPLI(SEL("SELECT" D U,EXPLIB),S)';
          .L↑F='APPLI(SEL(SEL("NAME",M↑F),MFUNLIB),
          .U(S,UO("ENV",U1(SEL("ENV",M↑F),SEL("NAME",M↑F),UNO())),
          .UO(UNIQUENUMBER,N↑F)))'
          B↑D=A↑D
          C↑D=A↑D
          E↑D=D↑D
          F↑D=D↑D
          G↑D=D↑D
          H↑D=D↑D
          M↑D=L↑D
          N↑D=L↑D;N↑I='1'

<EXPRESSION LIST>A=<EXPRESSION>B ',' <EXPRESSION LIST>C;D=<EXPRESSION>E
          <DECLNAMES>D
          <INDEX>I
          <MFUNLIB>A↑M=U(B↑M,C↑M);D↑M=E↑M
          <EXPLIB>A↑E=U(B↑E,C↑E);D↑E=E↑E
          <FUNCTION>A↑F='U(C↑F,UO(A↑I,B↑F))';D↑F='UO(D↑I,E↑F)'
          B↑D=A↑D
          C↑D=A↑D;C↑I=A↑I + 1
          E↑D=D↑D
END
```

SOURCE PROGRAM

```
LET F(X)=(IF X>3 THEN X ELSE (X+F((X+1))))
    IN F(2);
END
```

```
LISTING OF ATTRIBUTES OF PROGRAM NODE


          DISPLAY OF FLOWCHART-F
0      ----------
FUN       APPLI(SEL(SEL("NAME",U(UO("NAME","F"),UO("ENV",SEL("ENV",S)))),MFUNLIB),
U(S,UO("ENV",U1(SEL("ENV",U(UO("NAME","F"),UO("ENV",SEL("ENV",S)))),SEL("NAME",U
(UO("NAME","F"),UO("ENV",SEL("ENV",S)))),UNO())),UO(UNIQUENUMBER,UO(1,2))))
NXT


          DISPLAY OF MFUNLIB-M
F      ----------
     0     ----------
FUN       APPLI(SEL("SELECT" 2,EXPLIB),S)
NXT


          DISPLAY OF EXPLIB-E
SELECT2     ----------
     0      ----------
FUN       S
NXT       COND(GTN(SEL(1,SEL(SEL("F",SEL("ENV",S)),S)) ,3),1,2)

     1      ----------
FUN       SEL(1,SEL(SEL("F",SEL("ENV",S)),S))
NXT

     2      ----------
FUN       SEL(1,SEL(SEL("F",SEL("ENV",S)),S)) + APPLI(SEL(SEL("NAME",U(UO("NAME","
F"),UO("ENV",SEL("ENV",S)))),MFUNLIB),U(S,UO("ENV",U1(SEL("ENV",U(UO("NAME","F")
,UO("ENV",SEL("ENV",S)))),SEL("NAME",U(UO("NAME","F"),UO("ENV",SEL("ENV",S)))),U
NO())),UO(UNIQUENUMBER,UO(1,SEL(1,SEL(SEL("F",SEL("ENV",S)),S)) + 1))))
NXT
```

```
APPLY FLOWCHART TO NULL STATE VECTOR S


FINAL STATE VECTOR
9

TIME TO PROCESS SEMANTIC DESCRIPTION 1484 MLLLISEC
TIME TO PARSE 1117 MILLISEC
TIME TO EVALUATE SEMANTIC ATTRIBUTES 10182 MILLISEC
'EXECUTION' TIME 100 MILLISEC
'STATEMENTS' EXECUTED 10
```

SOURCE PROGRAM

```
LET F(X)=(X+X)
    IN LET G(P,X)=(P(X)+P(1))
        IN G(F,2);
END
```

```
LISTING OF ATTRIBUTES OF PROGRAM NODE


          DISPLAY OF FLOWCHART-F
0      ----------
FUN       APPLI(SEL(SEL("NAME",U(UO("NAME","G"),UO("ENV",SEL("ENV",S)))),MFUNLIB),
U(S,UO("ENV",U1(SEL("ENV",U(UO("NAME","G"),UO("ENV",SEL("ENV",S)))),SEL("NAME",U
(UO("NAME","G"),UO("ENV",SEL("ENV",S)))),UNO())),UO(UNIQUENUMBER,U(UO(2,2),UO(1,
U(UO("NAME","F"),UO("ENV",SEL("ENV",S)))))))
NXT


          DISPLAY OF MFUNLIB-M
F      ----------
     0      ----------
FUN       SEL(1,SEL(SEL("F",SEL("ENV",S)),S)) + SEL(1,SEL(SEL("F",SEL("ENV",S)),S)
)
NXT


G      ----------
     0      ----------
FUN       APPLI(SEL(SEL("NAME",SEL(1,SEL(SEL("G",SEL("ENV",S)),S))),MFUNLIB),U(S,U
O("ENV",U1(SEL("ENV",SEL(1,SEL(SEL("G",SEL("ENV",S)),S))),SEL("NAME",SEL(1,SEL(S
EL("G",SEL("ENV",S)),S))),UNO())),UO(UNIQUENUMBER,UO(1,SEL(2,SEL(SEL("G",SEL("EN
V",S)),S)))))) + APPLI(SEL(SEL("NAME",SEL(1,SEL(SEL("G",SEL("ENV",S)),S))),MFUNL
IB),U(S,UO("ENV",U1(SEL("ENV",SEL(1,SEL(SEL("G",SEL("ENV",S)),S))),SEL("NAME",SE
L(1,SEL(SEL("G",SEL("ENV",S)),S))),UNO())),UO(UNIQUENUMBER,UO(1,1))))
NXT


          DISPLAY OF EXPLIB-E
NUL
```

```
          APPLY FLOWCHART TO NULL STATE VECTOR S


          FINAL STATE VECTOR
6

TIME TO PROCESS SEMANTIC DESCRIPTION 1484 MILLISEC
TIME TO PARSE 1016 MILLISEC
TIME TO EVALUATE SEMANTIC ATTRIBUTES 12083 MILLISEC
'EXECUTION' TIME 83 MILLISEC
'STATEMENTS' EXECUTED 4
```

SOURCE PROGRAM

```
LET F(Y)=(Y+3)
    IN (F(1)+F(2));
END
```

LISTING OF ATTRIBUTES OF PROGRAM NODE

DISPLAY OF FLOWCHART-F

```
0      ----------
FUN       APPLI(SEL(SEL("NAME",U(UO("NAME","F"),UO("ENV",SEL("ENV",S)))),MFUNLIB),
U(S,UO("ENV",U1(SEL("ENV",U(UO("NAME","F"),UO("ENV",SEL("ENV",S)))),SEL("NAME",U
(UO("NAME","F"),UO("ENV",SEL("ENV",S)))),UNO())),UO(UNIQUENUMBER,UO(1,1)))) + AP
PLI(SEL(SEL("NAME",U(UO("NAME","F"),UO("ENV",SEL("ENV",S)))),MFUNLIB),U(S,UO("EN
V",U1(SEL("ENV",U(UO("NAME","F"),UO("ENV",SEL("ENV",S)))),SEL("NAME",U(UO("NAME"
,"F"),UO("ENV",SEL("ENV",S)))),UNO())),UO(UNIQUENUMBER,UO(1,2))))
NXT
```

DISPLAY OF MFUNLIB-M

```
F      ----------
     0      ----------
FUN       SEL(1,SEL(SEL("F",SEL("ENV",S)),S)) + 3
NXT
```

DISPLAY OF EXPLIB-E

```
NUL
```

```
          APPLY FLOWCHART TO NULL STATE VECTOR S


          FINAL STATE VECTOR
9

TIME TO PROCESS SEMANTIC DESCRIPTION 1484 MILLISEC
TIME TO PARSE 550 MILLISEC
TIME TO EVALUATE SEMANTIC ATTRIBUTES 6966 MILLISEC
'EXECUTION' TIME 33 MILLISEC
'STATEMENTS' EXECUTED 3
```

SOURCE PROGRAM

```
LET Y=2
    IN LET F(X)=(X+Y)
       IN F;
END
```

```
LISTING OF ATTRIBUTES OF PROGRAM NODE

          DISPLAY OF FLOWCHART-F
0      ----------
FUN      U(UO("NAME","F"),UO("ENV",SEL("ENV",S)))
NXT


          DISPLAY OF MFUNLIB-M
Y      ----------
     0      ----------
FUN      2
NXT

F      ----------
     0      ----------
FUN      SEL(1,SEL(SEL("F",SEL("ENV",S)),S)) • APPLI(SEL("Y",MFUNLIB),S)
NXT


          DISPLAY OF EXPLIB-E
NUL
```

```
APPLY FLOWCHART TO NULL STATE VECTOR S


FINAL STATE VECTOR
NAME     F
ENV

TIME TO PROCESS SEMANTIC DESCRIPTION 1484 MILLISEC
TIME TO PARSE 333 MILLISEC
TIME TO EVALUATE SEMANTIC ATTRIBUTES 4833 MILLISEC
'EXECUTION' TIME 0 MILLISEC
'STATEMENTS' EXECUTED 1
```

MINI-LANGUAGE 5

SYNTACTIC AND SEMANTIC DESCRIPTION

```
<PROGRAM>A=<DEC LIST>B ';' <COMMAND LIST>C. ';END'
        <FLOWCHART>A↑F=C↑F
        <PROCLIB>A↑P=B↑P
        <ARGLIB>A↑A=C↑A
        C↑N='NUL';C↑P='NUL';C↑S=B↑S

<DECLARATION>A='PROC P' <$NUMBR>B '(' <VARLIST>C ');' <SPEC LIST>D ';'
. <COMMAND LIST>F ';END'
        <PROCLIB>A↑P=UO('B↑VAL',F↑F)
        <SPECTABLE>A↑S=UO('B↑VAL',D↑S)
        C↑I=1
        D↑P=C↑V
        F↑N='NUL';F↑P=C↑V;F↑S=D↑S

<DEC LIST>A=<DECLARATION>B ';' <DEC LIST>C;D=<DECLARATION>E
        <PROCLIB>A↑P=U(B↑P,C↑P);D↑P=E↑P
        <SPECTABLE>A↑S=U(B↑S,C↑S);D↑S=E↑S

<VARLIST>A=<$ALPHA>B ',' <VARLIST>C;D=<$ALPHA>E
        <INDEX>I
        <VARLIST>A↑V=U(UO('B↑VAL',A↑I),C↑V);D↑V=UO('E↑VAL',D↑I)
        C↑I=A↑I + 1

<SPEC LIST>A=<SPEC>B ',' <SPEC LIST>C;D=<SPEC>E
        <PARAMTABLE>P
        <SPECTABLE>A↑S=U(B↑S,C↑S);D↑S=E↑S
        B↑P=A↑P
        C↑P=A↑P
        E↑P=D↑P

<SPEC>A='EXP ' <SVARLIST>B;
.C='COPY VAL ' <SVARLIST>D;
.E='LOC ' <SVARLIST>F
        <PARAMTABLE>P
        <SPECTABLE>A↑S=B↑S;C↑S=D↑S;E↑S=F↑S
        B↑P=A↑P;B↑T='EXP'
        D↑P=C↑P;D↑T='COPY VAL'
        F↑P=E↑P;F↑T='LOC'

<SVARLIST>A=<$ALPHA>B ',' <SVARLIST>C;D=<$ALPHA>E
        <PARAMTABLE>P
        <TYPE>T
        <SPECTABLE>A↑S=U(UO(SEL('B↑VAL',A↑P),'A↑T'),C↑S);
        .D S=UO(SEL('E↑VAL',D↑P),'D↑T')
        C↑P=A↑P;C↑T=A↑T

<COMMAND LIST>A=<COMMAND>B ';' <COMMAND LIST>C;D=<COMMAND>E
        <NEXT>N
        <PARAMTABLE>P
        <SPECTABLE>S
        <LINENO>A↑L=B↑L;D↑L=E↑L
        <FLOWCHART>A↑F=U(B↑F,C↑F);D↑F=E↑F
        <ARGLIB>A↑A=U(B↑A,C↑A);D↑A=E↑A
        B↑N=C↑L;B↑P=A↑P;B↑S=A↑S
        C↑N=A↑N;C↑P=A↑P;C↑S=A↑S
```

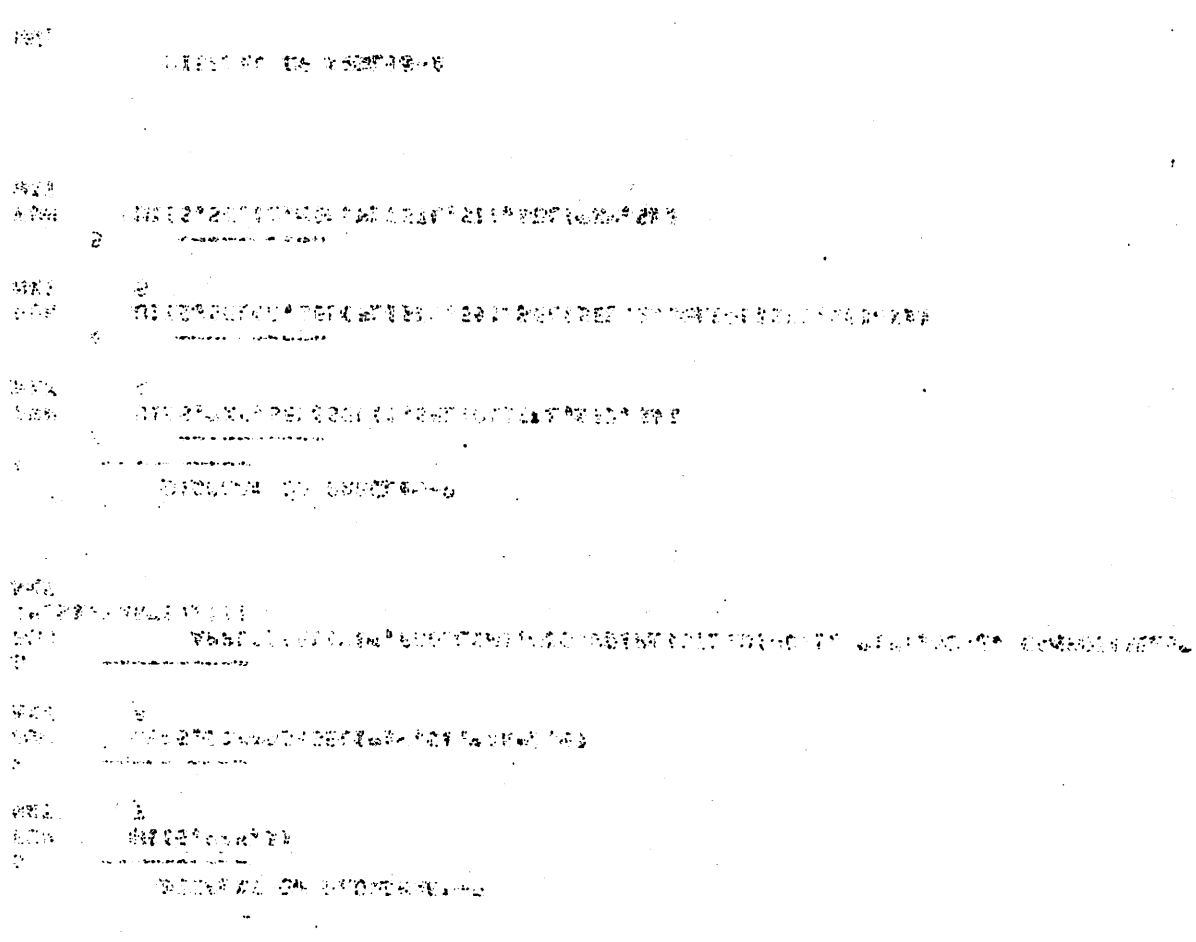
```
          E↑N=D↑N;E↑P=D↑P;E↑S=D↑S

<COMMAND>A=<ASSIGNMENT COMMAND>B;C=<PROCEDURE COMMAND>D
          <NEXT>N.
          <PARAMTABLE>P
          <SPECTABLE>S
          <LINENO>A↑L=B↑L;C↑L=D↑L
          <FLOWCHART>A↑F=B↑F;C↑F=D↑F
          <ARGLIB>A↑A='NUL';C↑A=D↑A
          B↑N=A↑N;B↑P=A↑P;B↑S=A↑S
          D↑N=C↑N;D↑S=C↑S

<ASSIGNMENT COMMAND>A=<EXPRESSION>B '=' <EXPRESSION>C
          <NEXT>N
          <PARAMTABLE>P
          <SPECTABLE>S
          <LINENO>A↑L=UNO()
          <FLOWCHART>A↑F=UO(A↑L,U(UO('FUN','U1(S,B↑L,C↑R)'),UO('NXT',A↑N)))
          B↑P=A↑P;B↑S=A↑S
          C↑P=A↑P;C↑S=A↑S

<PROCEDURE COMMAND>A='P' <$NUMBR>B '(' <ARGLIST>C ')'
          <NEXT>N
          <SPECTABLE>S
           <LINENO>A↑L=UNO()
           <FLOWCHART>A↑F=UO(A↑L,U(UO('FUN','     APPLI(SEL("B↑VAL",PROCLIB),
           .U(S,UO(PLISTL,C↑F))) '),UO('NXT',A↑N)))
           <ARGLIB>A↑A=C↑A
           C↑N=B↑VAL;C↑I=1;C↑S=A↑S

<ARGLIST>A=<ARG>B ',' <ARGLIST>C;D=<ARG>E
          <NAME>N
          <INDEX>I
          <SPECTABLE>S
          <FUNCTION>A↑F='U(B↑F,C↑F)';D↑F=E↑F
          <ARGLIB>A↑A=U(B↑A,C↑A);D↑A=E↑A
          B↑N=A↑N;B↑I=A↑I;B↑S=A↑S
          C↑N=A↑N;C↑I=A↑I + 1;C↑S=A↑S
          E↑N=D↑N;E↑I=D↑I;E↑S=D↑S

<EXPRESSION>A=<$NUMBR>B;
.C=<$ALPHA>D;
.E=<$ALPHA>F '=' <EXPRESSION>G '-'
          <PARAMTABLE>P
          <SPECTABLE>S
          <LHSFUN>A↑L='NUL';
          .C↑L=COND(EQU(SEL(SEL('D↑VAL',C↑P),C↑S),'COPY VAL'),'NUL',
           .COND(EQU(SEL(SEL('D↑VAL',C↑P),C↑S),'LOC'),
           .'SEL(' SEL('D↑VAL',C↑P) ',SEL(PLISTL,S))',
           .COND(EQU(SEL(SEL('D↑VAL',C↑P),C↑S),'EXP'),
           .'SEL("LHSVAL",APPLI(SEL(SEL(' SEL('D↑VAL',C↑P) ',SEL(PLISTL,S))
           .,ARGLIB),S))',
           .'"D↑VAL"')));
           .E↑L='COMPOS(G↑R,"F↑VAL")'
           <RHSFUN>A↑R=B↑VAL;
          .C↑R=COND(EQU(SEL(SEL('D↑VAL',C↑P),C↑S),'COPY VAL'),
           .'SEL(' SEL('D↑VAL',C↑P) ',SEL(PLISTL,S))',
           .COND(EQU(SEL(SEL('D↑VAL',C↑P),C↑S),'LOC'),'SEL(C↑L,S)',
           .COND(EQU(SEL(SEL('D↑VAL',C↑P),C↑S),'EXP'),
           .'SEL("RHSVAL",APPLI(SEL(SEL(' SEL('D↑VAL',C↑P) ',SEL(PLISTL,S))
```

```
                ..ARGLIB),S))',
                .'SEL("D↑VAL",S)')));
                .E↑R='SEL(E↑L,S)'
                G↑P=E↑P;G↑S=E↑S

<ARG>A=<EXPRESSION>B
                <NAME>N
                <INDEX>I
                <SPECTABLE>S
                <UNO>A↑U=UNO()
                <FUNCTION>A↑F=COND(EQU(SEL(A↑I,SEL(A↑N,A↑S)),'COPY VAL'),'UO(A↑I, B↑R)'
               ..COND(EQU(SEL(A↑I,SEL(A↑N,A↑S)),'LOC'),'UO(A↑L, B↑L)',
                .'UO(A↑I,A↑U)'))
                <ARGLIB>A↑A=COND(EQU(SEL(A↑I,SEL(A↑N,A↑S)),'EXP'),
                .UO(A U,UO(O,U(UO('FUN','U(UO("LHSVAL",B↑L),UO("RHSVAL",B↑R))'),
                .UO('NXT',NUL)))),
                .'NUL')
                B↑P='NUL';B↑S='NUL'
END
```

SOURCE PROGRAM

```
PROC P1(A,B);
          LOC A,B;
          X=A;
          A=B;
          B=X;
          END;
I=3;
AR"I-=6;
P1(I,AR"I-);
END
```

```
LISTING OF ATTRIBUTES OF PROGRAM NODE

          DISPLAY OF FLOWCHART-F
6     ----------
FUN     U1(S,"I",3)
NXT     7

7     ----------
FUN     U1(S,COMPOS(SEL("I",S),"AR"),6)
NXT     8

8     ----------
FUN          APPLI(SEL("1",PROCLIB),U(S,UO(PLISTL,U(UO(1, "I"),UO(2, COMPOS(SEL("
I",S),"AR"))))))
NXT


          DISPLAY OF PROCLIB-P
1     ----------
      3     ----------
FUN     U1(S,"X",SEL(SEL(1,SEL(PLISTL,S)),S))
NXT     4

      4     ----------
FUN     U1(S,SEL(1,SEL(PLISTL,S)),SEL(SEL(2,SEL(PLISTL,S)),S))
NXT     5

      5     ----------
FUN     U1(S,SEL(2,SEL(PLISTL,S)),SEL("X",S))
NXT


          DISPLAY OF ARGLIB-A
NUL
```



```
                APPLY FLOWCHART TO NULL STATE VECTOR S


                FINAL STATE VECTOR
PLIST-          ----------
1          I
2          37AR

X          3
I          6
AR         ----------
3          3


TIME TO PROCESS SEMANTIC DESCRIPTION 1839 MILLISEC
TIME TO PARSE 2468 MILLISEC
TIME TO EVALUATE SEMANTIC ATTRIBUTES 30439 MILLISEC
'EXECUTION' TIME 118 MILLISEC
'STATEMENTS' EXECUTED 6
```

SOURCE PROGRAM

```
PROC P1(A,B);
          COPY VAL A,B;
          X=A;
          A=B;
          B=X;
          END;
I=3;
AR"I-=6;
P1(I,AR"I-1);
END
```

```
LISTING OF ATTRIBUTES OF PROGRAM NODE


          DISPLAY OF FLOWCHART-F
30      ----------
FUN       U1(S,"I",3)
NXT       31

31      ----------
FUN       U1(S,COMPOS(SEL("I",S),"AR"),6)
NXT       32

32      ----------
FUN           APPLI(SEL("1",PROCLIB),U(S,UO(PLISTL,U(UO(1, SEL("I",S)),UO(2, SEL(C
OMPOS(SEL("I",S),"AR"),S)))))
NXT


          DISPLAY OF PROCLIB-P
1       ----------
      27      ----------
FUN       U1(S,"X",SEL(1,SEL(PLISTL,S)))
NXT       28

      28      ----------
FUN       U1(S,NUL,SEL(2,SEL(PLISTL,S)))
NXT       29

      29      ----------
FUN       U1(S,NUL,SEL("X",S))
NXT


          DISPLAY OF ARGLIB-A
NUL
```

APPLY FLOWCHART TO NULL STATE VECTOR S

FINAL STATE VECTOR

```
I     3
AR    ----------
3     6

PLIST-    ----------
1     3
2     6

X     3
```

TIME TO PROCESS SEMANTIC DESCRIPTION 1839 MILLISEC
TIME TO PARSE 2667 MILLISEC
TIME TO EVALUATE SEMANTIC ATTRIBUTES 30910 MILLISEC
'EXECUTION' TIME 102 MILLISEC
'STATEMENTS' EXECUTED 6

SOURCE PROGRAM

```
PROC P1(A,B);
          EXP A,B;
          X=A;
          A=B;
          B=X;
          END;
I=3;
AR"I-=6;
P1(I,AR"I-);
END
```

```
LISTING OF ATTRIBUTES OF PROGRAM NODE


          DISPLAY OF FLOWCHART-F
54      ----------
FUN       U1(S,"I",3)
NXT       55

55      ----------
FUN       U1(S,COMPOS(SEL("I",S),"AR"),6)
NXT       56

56      ----------
FUN           APPLI(SEL("1",PROCLIB),U(S,UO(PLISTL,U(UO(1,57),UO(2,58)))))
NXT



          DISPLAY OF PROCLIB-P
1       ----------
      51      ----------
FUN       U1(S,"X",SEL("RHSVAL",APPLI(SEL(SEL(1,SEL(PLISTL,S)),ARGLIB),S)))
NXT       52

      52      ----------
FUN       U1(S,SEL("LHSVAL",APPLI(SEL(SEL(1,SEL(PLISTL,S)),ARGLIB),S)),SEL("RHSVAL
",APPLI(SEL(SEL(2,SEL(PLISTL,S)),ARGLIB),S)))
NXT       53

      53      ----------
FUN       U1(S,SEL("LHSVAL",APPLI(SEL(SEL(2,SEL(PLISTL,S)),ARGLIB),S)),SEL("X",S))
NXT



          DISPLAY OF ARGLIB-A
57      ----------
      0       ----------
FUN       U(UO("LHSVAL","I"),UO("RHSVAL",SEL("I",S)))
NXT


58      ----------
      0       ----------
FUN       U(UO("LHSVAL",COMPOS(SEL("I",S),"AR")),UO("RHSVAL",SEL(COMPOS(SEL("I",S)
,"AR"),S)))
NXT
```

```
APPLY FLOWCHART TO NULL STATE VECTOR S

          FINAL STATE VECTOR
PLIST-    ----------
1      57
2      58

X      3
I      6
AR     ----------
3      6
6      3


TIME TO PROCESS SEMANTIC DESCRIPTION 1839 MILLISEC
TIME TO PARSE 2730 MILLISEC
TIME TO EVALUATE SEMANTIC ATTRIBUTES 31292 MILLISEC
'EXECUTION' TIME 180 MILLISEC
'STATEMENTS' EXECUTED 10
```

MINI-LANGUAGE 6

SYNTACTIC AND SEMANTIC DESCRIPTION

```
<PROGRAM>A=<DEC LIST>B ';' <COMMAND LIST>C ';END'
        <FLOWCHART>A↑F=CONU(EQU('A↑C','LEGAL'),C↑F,'NUL')
        <PROCLIB>A↑P=U(B↑P,UO('SQ',MKFLC(0,'SQ(SEL(1,S))',)))
        <CHECKTYPE>A↑C=COND(EQU('B↑C','LEGAL'),'C↑C','ILLEGAL')
        C↑N='NUL';C↑S=B↑S

<TYPE DESIGNATION>A='N';B='(' <TYPE LIST>C '—' <TYPE DESIGNATION>D ')'
        <CODE>A↑C='N';B↑C='(C↑C-D↑C)'
        <TYPE>A↑T=U(UO('RNG','N'),UO('CD','N'));
        .B↑T=U(UO('DMN',C↑L),UO('RNG','D↑C'),UO('CD','B↑C'))
        C↑I=1

<TYPE LIST>A=<TYPE DESIGNATION>B ',' <TYPE LIST>C;D=<TYPE DESIGNATION>E
        <INDEX>I
        <CODE>A↑C=SEL('CD',B↑T) ',' SEL('CD',C↑L);D C=SEL('CD',E↑T)
        <LIST>A↑L=U(UO('CD','A↑C'),UO(A↑L,B↑T),C↑L);
        .D L=U(UO('CD',SEL('CD',E↑T)),UO(D↑I,E↑T))
        C↑I=A↑I + 1

<DECLARATIONA>A='DEC ' <VARLIST>B ' TYPE ' <TYPE DESIGNATION>C
        <SYMBOLTABLE>A↑S=B↑S
        B↑T=C↑T

<VARLIST>A=<$ALPHA>B ',' <VARLIST>C;D=<$ALPHA>E
        <TYPE>T
        <SYMBOLTABLE>A↑S=U(UO('B↑VAL',U(UO('TYPE', A↑T ),UO('KIND','VAR'))),
        .C↑S);
        .D↑S=UO('E↑VAL',U(UO('TYPE', D↑T ),UO('KIND','VAR')))
        C↑T=A↑T

<DECLARATIONB>A='DEC ' <$ALPHA>B '(' <FPLIST>C ')=' <EXPRESSION>D ' WHERE '
. <TYPE LIST>E
      <SYMBOLTABLE>A↑S=UO('B↑VAL',U(UO('TYPE',U(UO('DMN',E↑L),UO('RNG','D↑T'),
        .UO('CD','(E↑C-D↑T)'))),
        .UO('KIND','NAME')))
        <PROCLIB>A↑P=UO('B↑VAL',UO(0,U(UO('FUN','D↑F'),UO('NXT',NUL))))
        <CHECKTYPE>A↑C=D↑C
        C↑I=1;C↑T=E↑L
        D↑S=C↑S
        E↑I='1'

<FPLIST>A=<$ALPHA>B ',' <FPLIST>C;D=<$ALPHA>E
        <INDEX>I
        <TYPELIST>T
        <SYMBOLTABLE>A↑S=U(UO('B VAL',U(U(UO('NO',A↑I),UO('KIND','PARAM')),
        .UO('TYPE',SEL(A↑I,A↑T)))),C↑S);
        .D↑S=UO('E↑VAL',U(U(UO('NO',D↑I),UO('KIND','PARAM')),UO('TYPE',SEL(D↑I,
        .D↑T))))
        C↑I=A↑I + 1;C↑T=A↑T

<DECLARATION>A=<DECLARATIONA>B;C=<DECLARATIONB>D
        <SYMBOLTABLE>A↑S=B↑S;C↑S=D↑S
        <PROCLIB>A↑P='NUL';C↑P=D↑P
        <CHECKTYPE>A↑C='LEGAL';C↑C=D↑C
```

```
<DEC LIST>A=<DECLARATION>B ';' <DEC LIST>C;D=<DECLARATION>E
          <SYMBOLTABLE>A↑S=U(B↑S,C↑S);D↑S=E↑S
          <PROCLIB>A↑P=U(B↑P,C↑P);D↑P=E↑P
          <CHECKTYPE>A↑C=COND(EQU('B↑C','LEGAL'),'C↑C','ILLEGAL');
          .D↑C=E↑C

<PRIMITIVE OBJECT>A=<$NUMBR>B;C='SQ';D='+'
          <VALUE>A↑V=B↑VAL;C↑V='"SQ"';D↑V='"ADD"'
          <TYPE>A↑T='N';C↑T='(N-N)';D↑T='(N,N-N)'

<IDENTIFIER>A=<$ALPHA>B
          <SYMBOLTABLE>S
          <FUNC>A↑F=COND(EQU(SEL('KIND',SEL('B↑VAL',A↑S)),'VAR'),'SEL("B↑VAL",S)'
          ,
          .COND(EQU(SEL('KIND',SEL('B↑VAL',A↑S)),'NAME'),'"B↑VAL"',
          .'SEL(' SEL('NO',SEL('B↑VAL',A↑S)) ',S)'))
          <TYPE>A↑T=SEL('TYPE',SEL('B↑VAL',A↑S))
          <VALUE>A↑V='B↑VAL'

<EXPRESSION>A=<PRIMITIVE OBJECT>B;
.C=<IDENTIFIER>D;
.E='SQ(' <EXPRESSION>F ')';
.G='(' <EXPRESSION>H '+' <EXPRESSION>I ')';
 .J=<IDENTIFIER>K '(' <ARGLIST>L ')';
.M='(IF ' <EXPRESSION>N '>' <EXPRESSION>O ' THEN ' <EXPRESSION>P ' ELSE '
. <EXPRESSION>Q ')'
           <SYMBOLTABLE>S
           <TYPE>A↑T=B↑T;
          .C↑T=SEL('CD',D↑T);
           .E↑T='N';
           .G↑T='N';
          .J↑T=SEL('RNG',K↑T);
           .M↑T=P↑T
          <FUNCTION>A↑F=B↑V;
          .C↑F=D↑F;
          .E↑F='SQ(F↑F)';
          .G↑F='(H↑F + I↑F)';
          .J↑F='APPLI(SEL(K↑F,PROCLIB),U(S,L↑F))';
          .M↑F='SELECT(N↑F,O↑F,P↑F,Q↑F)'
          <CHECKTYPE>A↑C='LEGAL';
          .C↑C='LEGAL';
          .E↑C=COND(EQU('F↑T F↑C','N LEGAL'),'LEGAL','ILLEGAL');
          .G↑C=COND(EQU('H↑T H↑C I↑T I↑C','N LEGAL N LEGAL'),'LEGAL','ILLEGAL');
          .J↑C=L↑C;
          .M↑C=COND(EQU('N↑T N↑C O↑T O↑C P↑C Q↑C','N LEGAL N LEGAL LEGAL LEGAL'),
          .COND(EQU('P↑T','Q↑T'),'LEGAL','ILLEGAL'),'ILLEGAL')
          D↑S=C↑S
          F↑S=E↑S
          H↑S=G↑S
          I↑S=G↑S
          K↑S=J↑S
          L↑S=J↑S;L↑I=1;L↑A=SEL('DMN',SEL('TYPE',SEL('K↑V',J↑S)))
          N↑S=M↑S
          O↑S=M↑S
          P↑S=M↑S
          Q↑S=M↑S

<ARGLIST>A=<EXPRESSION>B ',' <ARGLIST>C;D=<EXPRESSION>E
          <SYMBOLTABLE>S
          <INDEX>I
```

```
            <ARGTYPES>A
            <FUNCTION>A↑F='U(UO(A↑I,B↑F),C↑F)';D↑F='UO(D↑I,E↑F)'
            <CHECKTYPE>A↑C=COND(EQU('B↑C C↑C','LEGAL LEGAL'),
            .COND(EQU('B↑T',SEL('CD',SEL(A↑I,A↑A))),'LEGAL','ILLEGAL'),'ILLEGAL');
            .D C=COND(EQU('E↑T',SEL('CD',SEL(D↑I,D↑A))),'E↑C','ILLEGAL')
            B↑S=A↑S
            C↑S=A↑S;C↑I=A↑I + 1;C↑A=A↑A
            E↑S=D↑S

<ASSIGNMENT COMMAND>A=<$ALPHA>B '=' <EXPRESSION>C
            <NEXT>N
            <SYMBOLTABLE>S
            <LINENO>A↑L=UNO()
            <FLOWCHART>A↑F=UO(A↑L,U(UO('FUN','U1(S,"B↑VAL",C↑F)'),UO('NXT',A↑N)))
            <CHECKTYPE>A↑C=COND(EQU(SEL('CD',SEL('TYPE',SEL('B↑VAL',A↑S))),
            .'C↑T'),'C↑C'  ,'ILLEGAL')
            C↑S=A↑S

<COMMAND LIST>A=<ASSIGNMENT COMMAND>B ';' <COMMAND LIST>C;
.D=<ASSIGNMENT COMMAND>E
            <NEXT>N
            <SYMBOLTABLE>S
            <LINENO>A↑L=B↑L;D↑L=E↑L
            <FLOWCHART>A↑F=U(B↑F,C↑F);D↑F=E↑F
            <CHECKTYPE>A↑C=COND(EQU('B↑C','LEGAL'),'C↑C','ILLEGAL');D↑C=E↑C
            B↑N=C↑L;B↑S=A↑S
            C↑N=A↑N;C↑S=A↑S
            E↑N=D↑N;E↑S=D↑S
END
```

SOURCE PROGRAM

```
DEC A TYPE N;
DEC B TYPE (N-N);
DEC F(X)=X WHERE N;
DEC G(X)=X WHERE (N-N);
DEC H(X,Y)=X(Y) WHERE ((N-N)-(N-N)),(N-N);
B=H(G,SQ);
A=B(2);
B=H(F,2);
END
```

```
LISTING OF ATTRIBUTES OF PROGRAM NODE

          DISPLAY OF FLOWCHART-F
NUL


          DISPLAY OF PROCLIB-P
F      ----------
     0     ----------
FUN      SEL(1,S)
NXT


G      ----------
     0     ----------
FUN      SEL(1,S)
NXT


H      ----------
     0     ----------
FUN      APPLI(SEL(SEL(1,S),PROCLIB),U(S,UO(1,SEL(2,S))))
NXT


SQ     ----------
     0     ----------
FUN      SQ(SEL(1,S))
NXT


          DISPLAY OF CHECKTYPE-C
ILLEGAL
```

```
APPLY FLOWCHART TO NULL STATE VECTOR S

FINAL STATE VECTOR

TIME TO PROCESS SEMANTIC DESCRIPTION 2158 MILLISEC
TIME TO PARSE 3973 MILLISEC
TIME TO EVALUATE SEMANTIC ATTRIBUTES 42802 MILLISEC
'EXECUTION' TIME 8 MILLISEC
'STATEMENTS' EXECUTED 0
```

SOURCE PROGRAM

```
DEC A,B TYPE N;
DEC F(X)=SQ(SQ(X)) WHERE N;
A=2;
B=F(A);
END
```

```
LISTING OF ATTRIBUTES OF PROGRAM NODE


          DISPLAY OF FLOWCHART-F
5     ----------
FUN     U1(S,"A",2)
NXT     6

6     ----------
FUN     U1(S,"B",APPLI(SEL("F",PROCLIB),U(S,U0(1,SEL("A",S)))))
NXT


          DISPLAY OF PROCLIB-P
F     ----------
    0     ----------
FUN     SQ(SQ(SEL(1,S)))
NXT


SQ    ----------
    0     ----------
FUN     SQ(SEL(1,S))
NXT



          DISPLAY OF CHECKTYPE-C
LEGAL
```

```
APPLY FLOWCHART TO NULL STATE VECTOR S


FINAL STATE VECTOR
A     2
B     16

TIME TO PROCESS SEMANTIC DESCRIPTION 2158 MLLLISEC
TIME TO PARSE 2263 MILLISEC
TIME TO EVALUATE SEMANTIC ATTRIBUTES 12160 MILLISEC
'EXECUTION' TIME 35 MILLISEC
'STATEMENTS' EXECUTED 3
```

SOURCE PROGRAM

```
DEC X TYPE N;
DEC A,B TYPE (N-N);
DEC F(X)=SQ(SQ(X)) WHERE N;
DEC G(X,Y)=X(Y) WHERE (N-N),N;
A=SQ;
B=F;
X=(G(A,1)+G(B,2));
END
```

```
LISTING OF ATTRIBUTES OF PROGRAM NODE .


          DISPLAY OF FLOWCHART-F
37      ----------
FUN      U1(S,"A","SQ")
NXT      38

38      ----------
FUN      U1(S,"B","F")
NXT      39

39      ----------
FUN      U1(S,"X",(APPLI(SEL("G",PROCLIB),U(S,U(UO(1,SEL("A",S)),UO(2,1)))) + APP
LI(SEL("G",PROCLIB),U(S,U(UO(1,SEL("B",S)),UO(2,2))))))
NXT


          DISPLAY OF PROCLIB-P
F       ----------
     0      ----------
FUN      SQ(SQ(SEL(1,S)))
NXT

G      ----------
     0      ----------
FUN      APPLI(SEL(SEL(1,S),PROCLIB),U(S,UO(1,SEL(2,S))))
NXT


SQ      ----------
     0      ----------
FUN      SQ(SEL(1,S))
NXT



          DISPLAY OF CHECKTYPE-C
LEGAL
```

```
APPLY FLOWCHART TO NULL STATE VECTOR S


          FINAL STATE VECTOR
A      SQ
B      F
X      17

TIME TO PROCESS SEMANTIC DESCRIPTION 2158 MILLISEC
TIME TO PARSE 3078 MILLISEC
TIME TO EVALUATE SEMANTIC ATTRIBUTES 34774 MILLISEC
'EXECUTION' TIME 86 MILLISEC
'STATEMENTS' EXECUTED 7
```

SOURCE PROGRAM

```
DEC A,B TYPE N;
DEC F(X)=(X+X) WHERE N;
DEC G(X,Y)=(X+Y) WHERE N,N;
DEC H(X,Y)=X(Y) WHERE (N-N),N;
A=H(F,1);
B=H(G,1);
END
```

```
LISTING OF ATTRIBUTES OF PROGRAM NODE

          DISPLAY OF FLOWCHART-F
NUL


          DISPLAY OF PROCLIB-P
F      ----------
     0      ----------
FUN       (SEL(1,S) + SEL(1,S))
NXT


G      ----------
     0      ----------
FUN       (SEL(1,S) + SEL(2,S))
NXT


H      ----------
     0      ----------
FUN       APPLI(SEL(SEL(1,S),PROCLIB),U(S,UO(1,SEL(2,S))))
NXT


SQ     ----------
     0      ----------
FUN       SQ(SEL(1,S))
NXT


          DISPLAY OF CHECKTYPE-C
ILLEGAL
```

APPLY FLOWCHART TO NULL STATE VECTOR S

FINAL STATE VECTOR

```
TIME TO PROCESS SEMANTIC DESCRIPTION 2158 MILLISEC
TIME TO PARSE 2810 MILLISEC
TIME TO EVALUATE SEMANTIC ATTRIBUTES 36155 MILLISEC
'EXECUTION' TIME 8 MILLISEC
'STATEMENTS' EXECUTED 0
```

MINI-LANGUAGE 7

SYNTACTIC AND SEMANTIC DESCRIPTION

```
<PROGRAM>A=<DEC LIST>B ';' <COMMAND LIST>C.';END'
         <FLOWCHART>A↑F=C↑F
         <PROCLIB>A↑P=U(B↑P,
         .U(
         .MAKESUB('SQ',1,'EQU(SEL(TYPEL,SEL(1,S)),"N")',U(UO(TYPEL,"N"),
         .UO(VALL,SEL(VALL,SEL(1,S)) * SEL(VALL,SEL(1,S))))')
         .,U(
         .MAKESUB('ADD',2,'AND(EQU(SEL(TYPEL,SEL(1,S)),"N"),EQU(SEL(TYPEL,
         .SEL(2,S)),"N"))',U(UO(TYPEL,"N"),UO(VALL,SEL(VALL,SEL(1,S)) + SEL(VAL
         .L,
         .SEL(2,S))))')
         .,U(
         .MAKESUB('SELECT',4,'AND(EQU(SEL(TYPEL,SEL(1,S)),"N"),
         .EQU(SEL(TYPEL,SEL(2,S)),"N"))',
         .'SELECT(SEL(VALL,SEL(1,S)),SEL(VALL,SEL(2,S)),SEL(3,S),SEL(4,S))'),
         .MAKESUB('APPLI',2,'EQU(SEL("ARGNO",SEL(SEL(VALL,SEL(1,S)),PROCLIB)),
         .SEL("NO",SEL(2,S)))',
         .'APPLI(SEL("FLOWCHART",SEL(SEL(VALL,SEL(1,S)),PROCLIB)),SEL(2,S))')
         .))))
         C↑N='NUL';C↑S=B↑S

<DECLARATION>A='DEC ' <$ALPHA>B '(' <FPLIST>C ')=' <EXPRESSION>D
         <SYMBOLTABLE>A↑S=UO('B↑VAL',UO('KIND','NAME'))
         <PROCLIB>A↑P=UO('B↑VAL',U(UO('ARGNO',C↑N),UO('FLOWCHART',
         .UO(0,U(UO('FUN','D↑F'),UO('NXT',NUL))))))
         C↑I=1
         D↑S=C↑S

<DEC LIST>A=<DECLARATION>B ';' <DEC LIST>C;D=<DECLARATION>E
         <PROCLIB>A↑P=U(B↑P,C↑P);D↑P=E↑P
         <SYMBOLTABLE>A↑S=U(B↑S,C↑S);D↑S=E↑S

<FPLIST>A=<$ALPHA>B ',' <FPLIST>C;D=<$ALPHA>E
         <INDEX>I
         <NO>A↑N=1 + C N;D↑N=1
         <SYMBOLTABLE>A↑S=U(UO('B↑VAL',U(UO('NO',A↑I),UO('KIND','PARAM'))),C↑S);
         .D S=UO('E↑VAL',U(UO('NO',D↑I),UO('KIND','PARAM')))
         C↑I=A↑I + 1

<IDENTIFIER>A=<$ALPHA>B
         <SYMBOLTABLE>S
         <VALUE>A↑V='B↑VAL'
         <FUNCTION>A↑F=COND(EQU(SEL('KIND',SEL('B↑VAL',A↑S)),'NAME'),
         .'U(UO(VALL,"B↑VAL"),UO(TYPEL,"NAME"))',
         .COND(EQU(SEL('KIND',SEL('B↑VAL',A↑S)),'PARAM'),
         .'SEL(' SEL('NO',SEL('B↑VAL',A↑S)) ',S)',
         .'SEL("B↑VAL",S)'))

<ASSIGNMENT COMMAND>A=<$ALPHA>B '=' <EXPRESSION>C
          <NEXT>N
          <SYMBOLTABLE>S
          <LINENO>A↑L=UNO()
          <FLOWCHART>A↑F=UO(A↑L,U(UO('FUN','U1(S,"B↑VAL",C↑F)'),UO('NXT',A↑N)))
          C↑S=A↑S
```

```
<COMMAND LIST>A=<ASSIGNMENT COMMAND>B ';' <COMMAND LIST>C;
.D=<ASSIGNMENT COMMAND>E
          <NEXT>N
          <SYMBOLTABLE>S
          <LINENO>A↑L=B↑L;D↑L=E↑L
          <FLOWCHART>A↑F=U(B↑F,C↑F);D↑F=E↑F
          B↑N=C↑L;B↑S=A↑S
          C↑N=A↑N;C↑S=A↑S
          E↑N=D↑N;E↑S=D↑S

<PRIMITIVE OBJECT>A=<$NUMBR>B;C='SQ';D='+'
           <VALUE>A↑V=B↑VAL;C↑V='"SQ"';D↑V='"ADD"'
           <TYPE>A↑T='N';C↑T='(N_N)';D↑T='(N,N_N)'

<EXPRESSION>A=<PRIMITIVE OBJECT>B;
.C=<IDENTIFIER>D;
.E='SQ(' <EXPRESSION>F ')';
.G='(' <EXPRESSION>H '+' <EXPRESSION>I ')';
.J=<IDENTIFIER>K '(' <ARGLIST>L ')';
.M='(IF ' <EXPRESSION>N '>' <EXPRESSION>O ' THEN ' <EXPRESSION>P ' ELSE '
. <EXPRESSION>Q ')'
          <SYMBOLTABLE>S
          <FUNCTION>A↑F='U(UO(TYPEL,"B↑T"),UO(VALL,B↑V ))';
          .C↑F=D↑F;
          .E↑F='APPLI(SEL("FLOWCHART",SEL("SQ",PROCLIB)),UO(1,F↑F))';
          .G↑F='APPLI(SEL("FLOWCHART",SEL("ADD",PROCLIB)),U(UO(1,H↑F),UO(2,I↑F)))
          .';
          .J↑F='APPLI(SEL("FLOWCHART",SEL("APPLI",PROCLIB)),U(UO(1,K↑F),
          .UO(2,L↑F)))';
          .M↑F='APPLI(SEL("FLOWCHART",SEL("SELECT",PROCLIB)),
          .U(U(UO(1,N↑F),UO(2,O↑F)),U(UO(3,P↑F),UO(4,Q↑F))))'
          Q↑S=M↑S
          D↑S=C↑S
          F↑S=E↑S
          H↑S=G↑S
          I↑S=G↑S
          K↑S=J↑S
          L↑S=J↑S
          N↑S=M↑S
          O↑S=M↑S
          P↑S=M↑S

<ARGLIST>A=<EXPLIST>B
          <SYMBOLTABLE>S
          <FUNCTION>A F='U(UO("NO",B N),B F)'
          B↑S=A↑S;B↑I=1

<EXPLIST>A=<EXPRESSION>B ',' <EXPLIST>C;D=<EXPRESSION>E
          <SYMBOLTABTE>S
          <INDEX>I
          <NO>A↑N=1 + C↑N;D↑N=1
          <FUNCTION>A↑F='U(UO(A↑I,B↑F),C↑F)';D↑F='UO(D↑L,E↑F)'
          B↑S=A↑S
          C↑S=A↑S;C↑I=A↑I + 1
          E↑S=D↑S
END
```

SOURCE PROGRAM

```
DEC F(X,Y)=(IF Y>3 THEN (X+Y) ELSE X(Y));
A=5;
B=6;
C=F(A,B);
A=SQ;
B=2;
D=F(A,B);
END
```

```
LISTING OF ATTRIBUTES OF PROGRAM NODE


          DISPLAY OF FLOWCHART-F
61      ----------
FUN      U1(S,"A",U(UO(TYPEL,"N"),UO(VALL,5 )))
NXT      62

62      ----------
FUN      U1(S,"B",U(UO(TYPEL,"N"),UO(VALL,6 )))
NXT      63

63      ----------
FUN      U1(S,"C",APPLI(SEL("FLOWCHART",SEL("APPLI",PROCLIB)),U(UO(1,U(UO(VALL,"F
"),UO(TYPEL,"NAME")))),UO(2,U(UO("NO",2),U(UO(1,SEL("A",S)),UO(2,SEL("B",S)))))))
)
NXT      64

64      ----------
FUN      U1(S,"A",U(UO(TYPEL,"(N_N)"),UO(VALL,"SQ" )))
NXT      65

65      ----------
FUN      U1(S,"B",U(UO(TYPEL,"N"),UO(VALL,2 )))
NXT      66

66      ----------
FUN      U1(S,"D",APPLI(SEL("FLOWCHART",SEL("APPLI",PROCLIB)),U(UO(1,U(UO(VALL,"F
"),UO(TYPEL,"NAME")))),UO(2,U(UO("NO",2),U(UO(1,SEL("A",S)),UO(2,SEL("B",S))))))
)
NXT


          DISPLAY OF PROCLIB-P
F       ----------
ARGNO      2
     FLOWCHART      ----------
           0       ----------
FUN      APPLI(SEL("FLOWCHART",SEL("SELECT",PROCLIB)),U(U(UO(1,SEL(2,S)),UO(2,U(U
O(TYPEL,"N"),UO(VALL,3 )))),U(UO(3,APPLI(SEL("FLOWCHART",SEL("ADD",PROCLIB)),U(U
O(1,SEL(1,S)),UO(2,SEL(2,S))))),UO(4,APPLI(SEL("FLOWCHART",SEL("APPLI",PROCLIB))
,U(UO(1,SEL(1,S)),UO(2,U(UO("NO",1),UO(1,SEL(2,S))))))))))
NXT


SQ      ----------
ARGNO      1
     FLOWCHART      ----------
           0       ----------
FUN      S
NXT      COND(EQU(SEL(TYPEL,SEL(1,S)),"N"),1,2)

           1       ----------
FUN      U(UO(TYPEL,"N"),UO(VALL,SEL(VALL,SEL(1,S)) * SEL(VALL,SEL(1,S))))
NXT

           2       ----------
```

```
FUN     UO(TYPEL, "TYPE ERROR" )
NXT


ADD     ----------
ARGNO     2
     FLOWCHART      ----------
          0         ----------
FUN     S
NXT     COND(AND(EQU(SEL(TYPEL,SEL(1,S)),"N"),EQU(SEL(TYPEL,SEL(2,S)),"N")),1,2)

          1         ----------
FUN     U(UO(TYPEL,"N"),UO(VALL,SEL(VALL,SEL(1,S)) + SEL(VALL,SEL(2,S))))
NXT

          2         ----------
FUN     UO(TYPEL, "TYPE ERROR" )
NXT


SELECT    ----------
ARGNO     4
     FLOWCHART      ----------
          0         ----------
FUN     S
NXT     COND(AND(EQU(SEL(TYPEL,SEL(1,S)),"N"),EQU(SEL(TYPEL,SEL(2,S)),"N")),1,2)

          1         ----------
FUN     SELECT(SEL(VALL,SEL(1,S)),SEL(VALL,SEL(2,S)),SEL(3,S),SEL(4,S))
NXT

          2         ----------
FUN     UO(TYPEL, "TYPE ERROR" )
NXT


APPLI     ----------
ARGNO     2
     FLOWCHART      ----------
          0         ----------
FUN     S
NXT     COND(EQU(SEL("ARGNO",SEL(SEL(VALL,SEL(1,S)),PROCLIB)),SEL("NO",SEL(2,S))
),1,2)

          1         ----------
FUN     APPLI(SEL("FLOWCHART",SEL(SEL(VALL,SEL(1,S)),PROCLIB)),SEL(2,S))
NXT

          2         ----------
FUN     UO(TYPEL, "TYPE ERROR" )
NXT
```

APPLY FLOWCHART TO NULL STATE VECTOR S

FINAL STATE VECTOR

```
C       ----------
TYPE-       N
VAL-       11

A       ----------
TYPE-       (N_N)
VAL-       SQ

B       ----------
TYPE-       N
VAL-       2

D       ----------
TYPE-       N
VAL-       4
```

TIME TO PROCESS SEMANTIC DESCRIPTION 1410 MILLISEC
TIME TO PARSE 2919 MILLISEC
TIME TO EVALUATE SEMANTIC ATTRIBUTES 15790 MILLISEC
'EXECUTION' TIME 690 MILLISEC
'STATEMENTS' EXECUTED 26

SOURCE PROGRAM

```
DEC F(P,X,Y)=P(X,Y);
DEC G(X,Y)=(X+Y);
DEC H(P,Q)=P(Q(2));
A=F(G,1,2);
B=F(H,SQ,SQ);
END
```

```
LISTING OF ATTRIBUTES OF PROGRAM NODE


          DISPLAY OF FLOWCHART-F
28      ----------
FUN       U1(S,"A",APPLI(SEL("FLOWCHART",SEL("APPLI",PROCLIB)),U(UO(1,U(UO(VALL,"F
")),UO(TYPEL,"NAME"))),UO(2,U(UO("NO",3),U(UO(1,U(UO(VALL,"G"),UO(TYPEL,"NAME")))
,U(UO(2,U(UO(TYPEL,"N"),UO(VALL,1 ))),UO(3,U(UO(TYPEL,"N"),UO(VALL,2 ))))))))))
NXT       29

29      ----------
FUN       U1(S,"B",APPLI(SEL("FLOWCHART",SEL("APPLI",PROCLIB)),U(UO(1,U(UO(VALL,"F
")),UO(TYPEL,"NAME"))),UO(2,U(UO("NO",3),U(UO(1,U(UO(VALL,"H"),UO(TYPEL,"NAME")))
,U(UO(2,U(UO(TYPEL,"(N_N)"),UO(VALL,"SQ" ))),UO(3,U(UO(TYPEL,"(N_N)"),UO(VALL,"S
Q" ))))))))))
NXT



          DISPLAY OF PROCLIB-P
F       ----------
ARGNO       3
      FLOWCHART       ----------
            0       ----------
FUN       APPLI(SEL("FLOWCHART",SEL("APPLI",PROCLIB)),U(UO(1,SEL(1,S)),UO(2,U(UO("
NO",2),U(UO(1,SEL(2,S)),UO(2,SEL(3,S)))))))
NXT



G       ----------
ARGNO       2
      FLOWCHART       ----------
            0       ----------
FUN       APPLI(SEL("FLOWCHART",SEL("ADD",PROCLIB)),U(UO(1,SEL(1,S)),UO(2,SEL(2,S)
)))
NXT



H       ----------
ARGNO       2
      FLOWCHART       ----------
            0       ----------
FUN       APPLI(SEL("FLOWCHART",SEL("APPLI",PROCLIB)),U(UO(1,SEL(1,S)),UO(2,U(UO("
NO",1),UO(1,APPLI(SEL("FLOWCHART",SEL("APPLI",PROCLIB)),U(UO(1,SEL(2,S)),UO(2,U(
UO("NO",1),UO(1,U(UO(TYPEL,"N"),UO(VALL,2 ))))))))))))
NXT



SQ      ----------
ARGNO       1
      FLOWCHART       ----------
            0       ----------
FUN       S
NXT       COND(EQU(SEL(TYPEL,SEL(1,S)),"N"),1,2)

            1       ----------
```

```
FUN     U(UO(TYPEL,"N"),UO(VALL,SEL(VALL,SEL(1,S)) * SEL(VALL,SEL(1,S))))
NXT

          2     ----------
FUN     UO(TYPEL, "TYPE ERROR" )
NXT


ADD     ----------
ARGNO     2
     FLOWCHART      ----------
          0     ----------
FUN     S
NXT     COND(AND(EQU(SEL(TYPEL,SEL(1,S)),"N"),EQU(SEL(TYPEL,SEL(2,S)),"N")),1,2)

          1     ----------
FUN     U(UO(TYPEL,"N"),UO(VALL,SEL(VALL,SEL(1,S)) + SEL(VALL,SEL(2,S))))
NXT

          2     ----------
FUN     UO(TYPEL, "TYPE ERROR" )
NXT


SELECT     ----------
ARGNO     4
     FLOWCHART      ----------
          0     ----------
FUN     S
NXT     COND(AND(EQU(SEL(TYPEL,SEL(1,S)),"N"),EQU(SEL(TYPEL,SEL(2,S)),"N")),1,2)

          1     ----------
FUN     SELECT(SEL(VALL,SEL(1,S)),SEL(VALL,SEL(2,S)),SEL(3,S),SEL(4,S))
NXT

          2     ----------
FUN     UO(TYPEL, "TYPE ERROR" )
NXT


APPLI     ----------
ARGNO     2
     FLOWCHART      ----------
          0     ----------
FUN     S
NXT     COND(EQU(SEL("ARGNO",SEL(SEL(VALL,SEL(1,S)),PROCLIB)),SEL("NO",SEL(2,S))
),1,2)

          1     ----------
FUN     APPLI(SEL("FLOWCHART",SEL(SEL(VALL,SEL(1,S)),PROCLIB)),SEL(2,S))
NXT

          2     ----------
FUN     UO(TYPEL, "TYPE ERROR" )
NXT
```

APPLY FLOWCHART TO NULL STATE VECTOR S

FINAL STATE VECTOR

```
A       ----------
TYPE-       N
VAL-        3

B       ----------
TYPE-       N
VAL-        16
```

TIME TO PROCESS SEMANTIC DESCRIPTION 1410 MILLISEC
TIME TO PARSE 3221 MILLISEC
TIME TO EVALUATE SEMANTIC ATTRIBUTES 15884 MILLISEC
'EXECUTION' TIME 463 MILLISEC
'STATEMENTS' EXECUTED 24

SOURCE PROGRAM

```
DEC F(X,Y)=(IF Y>3 THEN (X+Y) ELSE X(Y));
A=5;
8=6;
C=F(A,B);
A=SQ;
D=F(A,B);
END
```

```
LISTING OF ATTRIBUTES OF PROGRAM NODE


          DISPLAY OF FLOWCHART-F
42      ----------
FUN       U1(S,"A",U(UO(TYPEL,"N"),UO(VALL,5 )))
NXT       43

43      ----------
FUN       U1(S,"B",U(UO(TYPEL,"N"),UO(VALL,6 )))
NXT       44

44      ----------
FUN       U1(S,"C",APPLI(SEL("FLOWCHART",SEL("APPLI",PROCLIB)),U(UO(1,U(UO(VALL,"F
"),UO(TYPEL,"NAME"))),UO(2,U(UO("NO",2),U(UO(1,SEL("A",S)),UO(2,SEL("B",S)))))))
)
NXT       45

45      ----------
FUN       U1(S,"A",U(UO(TYPEL,"(N_N)"),UO(VALL,"SQ" )))
NXT       46

46      ----------
FUN       U1(S,"D",APPLI(SEL("FLOWCHART",SEL("APPLI",PROCLIB)),U(UO(1,U(UO(VALL,"F
"),UO(TYPEL,"NAME"))),UO(2,U(UO("NO",2),U(UO(1,SEL("A",S)),UO(2,SEL("B",S)))))))
)
NXT


          DISPLAY OF PROCLIB-P
F       ----------
ARGNO       2
      FLOWCHART       ----------
            0       ----------
FUN       APPLI(SEL("FLOWCHART",SEL("SELECT",PROCLIB)),U(U(UO(1,SEL(2,S)),UO(2,U(U
O(TYPEL,"N"),UO(VALL,3 )))),U(UO(3,APPLI(SEL("FLOWCHART",SEL("ADD",PROCLIB)),U(U
O(1,SEL(1,S)),UO(2,SEL(2,S))))),UO(4,APPLI(SEL("FLOWCHART",SEL("APPLI",PROCLIB))
,U(UO(1,SEL(1,S)),UO(2,U(UO("NO",1),UO(1,SEL(2,S))))))))))
NXT


SQ      ----------
ARGNO       1
      FLOWCHART       ----------
            0       ----------
FUN       S
NXT       COND(EQU(SEL(TYPEL,SEL(1,S)),"N"),1,2)

            1       ----------
FUN       U(UO(TYPEL,"N"),UO(VALL,SEL(VALL,SEL(1,S)) * SEL(VALL,SEL(1,S))))
NXT

            2       ----------
FUN       UO(TYPEL, "TYPE ERROR" )
NXT
```

```
ADD      ----------
ARGNO      2
     FLOWCHART      ----------
                0      ----------
FUN        S
NXT        COND(AND(EQU(SEL(TYPEL,SEL(1,S)),"N"),EQU(SEL(TYPEL,SEL(2,S)),"N")),1,2)

                1      ----------
FUN        U(UO(TYPEL,"N"),UO(VALL,SEL(VALL,SEL(1,S)) + SEL(VALL,SEL(2,S))))
NXT

                2      ----------
FUN        UO(TYPEL, "TYPE ERROR" )
NXT



SELECT      ----------
ARGNO      4
     FLOWCHART      ----------
                0      ----------
FUN        S
NXT        COND(AND(EQU(SEL(TYPEL,SEL(1,S)),"N"),EQU(SEL(TYPEL,SEL(2,S)),"N")),1,2)

                1      ----------
FUN        SELECT(SEL(VALL,SEL(1,S)),SEL(VALL,SEL(2,S)),SEL(3,S),SEL(4,S))
NXT

                2      ----------
FUN        UO(TYPEL, "TYPE ERROR" )
NXT



APPLI      ----------
ARGNO      2
     FLOWCHART      ----------
                0      ----------
FUN        S
NXT        COND(EQU(SEL("ARGNO",SEL(SEL(VALL,SEL(1,S)),PROCLIB)),SEL("NO",SEL(2,S))
),1,2)

                1      ----------
FUN        APPLI(SEL("FLOWCHART",SEL(SEL(VALL,SEL(1,S)),PROCLIB)),SEL(2,S))
NXT

                2      ----------
FUN        UO(TYPEL, "TYPE ERROR" )
NXT
```

SOURCE PROGRAM

```
DEC F(X)=SQ(SQ(X));
A=1;
B=2;
C=F;
D=C(B);
END
```

```
LISTING OF ATTRIBUTES OF PROGRAM NODE


          DISPLAY OF FLOWCHART-F
3     ----------
FUN     U1(S,"A",U(UO(TYPEL,"N"),UO(VALL,1 )))
NXT     4

4     ----------
FUN     U1(S,"B",U(UO(TYPEL,"N"),UO(VALL,2 )))
NXT     5

5     ----------
FUN     U1(S,"C",U(UO(VALL,"F"),UO(TYPEL,"NAME")))
NXT     6

6     ----------
FUN     U1(S,"D",APPLI(SEL("FLOWCHART",SEL("APPLI",PROCLIB)),U(UO(1,SEL("C",S)),
UO(2,U(UO("NO",1),UO(1,SEL("B",S)))))))
NXT



          DISPLAY OF PROCLIB-P
F     ----------
ARGNO     1
     FLOWCHART     ----------
          0     ----------
FUN     APPLI(SEL("FLOWCHART",SEL("SQ",PROCLIB)),UO(1,APPLI(SEL("FLOWCHART",SEL(
"SQ",PROCLIB)),UO(1,SEL(1,S)))))
NXT


SQ     ----------
ARGNO     1
     FLOWCHART     ----------
          0     ----------
FUN     S
NXT     COND(EQU(SEL(TYPEL,SEL(1,S)),"N"),1,2)

          1     ----------
FUN     U(UO(TYPEL,"N"),UO(VALL,SEL(VALL,SEL(1,S)) * SEL(VALL,SEL(1,S))))
NXT

          2     ----------
FUN     UO(TYPEL, "TYPE ERROR" )
NXT


ADD     ----------
ARGNO     2
     FLOWCHART     ----------
          0     ----------
FUN     S
NXT     COND(AND(EQU(SEL(TYPEL,SEL(1,S)),"N"),EQU(SEL(TYPEL,SEL(2,S)),"N")),1,2)

          1     ----------
```

```
FUN     U(UO(TYPEL,"N"),UO(VALL,SEL(VALL,SEL(1,S)) + SEL(VALL,SEL(2,S))))
NXT

          2     ----------
FUN     UO(TYPEL, "TYPE ERROR" )
NXT


SELECT    ----------
ARGNO     4
     FLOWCHART     ----------
          0     ----------
FUN     S
NXT     COND(AND(EQU(SEL(TYPEL,SEL(1,S)),"N"),EQU(SEL(TYPEL,SEL(2,S)),"N")),1,2)

          1     ----------
FUN     SELECT(SEL(VALL,SEL(1,S)),SEL(VALL,SEL(2,S)),SEL(3,S),SEL(4,S))
NXT

          2     ----------
FUN     UO(TYPEL, "TYPE ERROR" )
NXT


APPLI     ----------
ARGNO     2
     FLOWCHART     ----------
          0     ----------
FUN     S
NXT     COND(EQU(SEL("ARGNO",SEL(SEL(VALL,SEL(1,S)),PROCLIB)),SEL("NO",SEL(2,S))
),1,2)

          1     ----------
FUN     APPLI(SEL("FLOWCHART",SEL(SEL(VALL,SEL(1,S)),PROCLIB)),SEL(2,S))
NXT

          2     ----------
FUN     UO(TYPEL, "TYPE ERROR" )
NXT
```

```
                APPLY FLOWCHART TO NULL STATE VECTOR S

                FINAL STATE VECTOR
A       ----------
TYPE-       N
VAL-        1

B       ----------
TYPE-       N
VAL-        2

C       ----------
VAL-        F
TYPE-       NAME

D       ----------
TYPE-       N
VAL-        16

TIME TO PROCESS SEMANTIC DESCRIPTION 1410 MILLLSEC
TIME TO PARSE 1966 MILLISEC
TIME TO EVALUATE SEMANTIC ATTRIBUTES 7057 MILLLSEC
'EXECUTION' TIME 189 MILLISEC
'STATEMENTS' EXECUTED 11
```

MINI-LANGUAGE 8

SYNTACTIC AND SEMANTIC DESCRIPTION

```
<PROGRAM>A=<DECLARATION LIST>B ';' <COMMAND LIST>C ';END'
        <FLOWCHART>A F=U(B↑F,C↑F)
        <IDLIST>A I=B↑I
        C↑T=B↑T;C↑N='NUL'
        B↑N=C↑S

<DECLARATION LIST>A=<DECLARATION>B ';' <DECLARATION LIST>C;D=<DECLARATION>E
        <NEXT>N
        <STATENOO>A↑S=B↑S;D↑S=E↑S
        <IDLIST>A↑I=U(B↑I,C↑I);D↑I=E↑I
        <TYPETABLE>A↑T=U(B↑T,C↑T);D↑T=E↑T
        <FLOWCHART>A↑F=U(B↑F,C↑F);D↑F=E↑F
        B↑N=C↑S
        C↑N=A↑N
        E↑N=D↑N

<DECLARATION>A='DEC ' <VARLIST>B;
          .C='DEC ' <$IDENT>D '=' <COMPONENTS DESCRIPTOR>E;
          .F='DEC ' <$IDENT>G '=' <STRUCTURE LIST>H
        <NEXT>N
        <STATENO>A↑S=UNO();C↑S=C↑N;F↑S=F↑N
        <IDLIST>A↑I='NUL';C↑I=UO('D↑VAL',UO('D↑VAL','XXX'));F↑I=UO('G↑VAL',H↑L)
        <TYPETABLE>A↑T='NUL';C↑T=UO('D↑VAL',E↑S);F↑T='NUL'
        <FLOWCHART>A↑F=UO(A↑S,U(UO('FUN','B↑F'),UO('NEXT',A↑N)));C↑F='NUL';
        .F↑F='NUL'
        B↑I='1'

<COMPONENTS DESCRIPTOR>A='<' <COMPONENT LIST>B '>'
        <STRUCTURE>A↑S=B↑S
        B↑I='1'

<COMPONENT LIST>A=<COMPONENT DESC>B ' / ' <COMPONENT LIST>C;D=<COMPONENT DESC>E
        <INDEX>I
        <STRUCTURE>A↑S=U(B↑S,C↑S);D↑S=E↑S
        C↑I=A↑I + 1
        B↑I=A↑I
        E↑I=D↑I

<COMPONENT DESC>A=<$IDENT>B ':' <$IDENT>C
        <INDEX>I
        <STRUCTURE>A↑S=UO(A↑I,'B↑VAL')

<STRUCTURE LIST>A=<$IDENT>B ' & ' <STRUCTURE LIST>C;D=<$IDENT>E
        <LIST>A↑L=U(C↑L,UO('B↑VAL','XXX'));D L=UO('E↑VAL','XXX')

<VARLIST>A=<$IDENT>B ',' <VARLIST>C;D=<$IDENT>E
        <INDEX>I
        <FUNCTION>A↑F='U(C↑F,UO("B↑VAL",OMEGA))';D↑F='UO("E↑VAL",OMEGA)'
        C↑I=A I + 1

<COMMAND LIST>A=<COMMAND>B ';' <COMMAND LIST>C;D=<COMMAND>E
        <NEXT>N
        <TYPETABLE>T
        <STATENO>A↑S=B↑S;D↑S=E↑S
        <FLOWCHART>A↑F=U(B↑F,C↑F);D↑F=E↑F
```

```
          B↑T=A↑T;B↑N=C↑S
          C↑T=A↑T;C↑N=A↑N
          E↑T=D↑T;E↑N=D↑N

<COMMAND>A=<ASSIGNMENT STATEMENT>B;C=<REPEAT STATEMENT>D
          <NEXT>N
          <TYPETABLE>T
          <STATENO>A↑S=B↑S;C↑S=D↑S
          <FLOWCHART>A↑F=B↑F;C↑F=D↑F
          B↑N=A↑N;B↑T=A↑T
          D↑N=C↑N;D↑T=C↑T

<ASSIGNMENT STATEMENT>A=<$IDENT>B ':=' <EXPRESSION>C;D=<$IDENT>E
. '(' <EXPRESSION>F ')=' <EXPRESSION>G
          <NEXT>N
          <TYPETABEE>T
          <STATENO>A↑S=UNO();D↑S=UNO()
          <FLOWCHART>A↑F=UO(A S,U(UO('FUN','U(S,UO("B↑VAL",C↑R))'),UO('NEXT',A↑N)
          .));
          .D↑F=UO(D↑S,U(UO('FUN','U1(S,COMPOS("E↑VAL",F↑L1,G↑R)')
          .,UO('NEXT',D↑N)))
          C↑T=A↑T
          F↑T=D↑T
          G↑T=D↑T

<REPEAT STATEMENT>A='IF ' <PREDICATE>B ' THEN ' <ASSIGNMENT STATEMENT>C
. ';ELSE REPEAT AFTER ' <ASSIGNMENT STATEMENT>D
          <NEXT>N
          <TYPETABLE>T
          <STATENO>A↑S=UNO()
          <FLOWCHART>A↑F=U(UO(A↑S,U(UO('FUN','ID(S)')),UO('NEXT','COND(B↑P,C↑S,D↑S
          .)'))),U(C↑F,D↑F))
          C↑T=A↑T;C↑N=A↑N
          D↑T=A↑T;D↑N=A↑S
          B↑T=A↑T

<PREDICATE>A='(IS ' <$IDENT>B ')(' <EXPRESSION>C ')'
          <TYPETABLE>T
          <PREDICATE>A↑P='ELEM(SEL(TYPEL,C↑R),SEL("B↑VAL",IDLIST))'
          C↑T=A↑T

<EXPRESSION>A=<$IDENT>B;
.C=<$NUMBR>D;
.E=<CONSTRUCTOR EXPRESSION>F;
.I=<POINTER EXPRESSION>J;
.K=<VALUE EXPRESSION>L;
.G=<SELECTOR EXPRESSION>H
          <TYPETABLE>T
          <LHSFUN>A↑L='"B↑VAL"';C↑L='NUL';
          .E↑L=F↑L;G↑L=H↑L;I↑L=J↑L;K↑L=L↑L
          <RHSFUN>A↑R='SEL("B↑VAL",S)';
          .C↑R=D↑VAL;
          .E↑R=F↑R;G↑R=H↑R;I↑R=J↑R;K↑R=L↑R
          F↑T=E↑T;H↑T=G↑T;J↑T=I↑T;L↑T=K↑T

<CONSTRUCTOR EXPRESSION>A='(CONS ' <$IDENT>B ')(' <CONSTRUCTOR LIST>C ')'
          <TYPETABLE>T
          <RHSFUN>A↑R='U(UO(TYPEL,"B↑VAL" ),C↑F)'
          <LHSFUN>A↑L='SLX(UO(UNO(),A↑R))'
          C C=SEL('B↑VAL',A↑T)
```

```
        C↑I='1'
        C↑T=A↑T

<CONSTRUCTOR LIST>A=<EXPRESSION>B ',' <CONSTRUCTOR LIST>C;D=<EXPRESSION>E
        <TYPETABLE>T
        <COMPONENTLIST>C
        <INDEX>I
        <FUNCTION>A↑F='U(UO("' SEL(A↑I,A↑C) '",B↑R),C↑F)';
        .D↑F='UO("' SEL(D↑I,D↑C) '",E↑R)'
        B↑T=A↑T
        C↑T=A↑T;C↑C=A↑C;C↑I=A↑I + 1
        E↑T=D↑T

<SELECTOR EXPRESSION>A=<$IDENT>B '(' <EXPRESSION>C ')'
        <TYPETABLE>T
        <LHSFUN>A↑L='COMPOS("B↑VAL",C↑L)'
        <RHSFUN>A↑R='SEL("B↑VAL",C↑R)'
        C↑T=A↑T

<POINTER EXPRESSION>A='PTR(' <EXPRESSION>B ')';C='*'
        <TYPETABLE>T
        <LHSFUN>A↑L='SLX(UO(UNO(),A↑R))';C↑L='NUL'
        <RHSFUN>A↑R=B↑L;C↑R='"*"'
        B↑T=A↑T

<VALUE EXPRESSION>A='VAL(' <EXPRESSION>B ')'
        <TYPETABLE>T
        <LHSFUN>A↑L=B↑R
        <RHSFUN>A↑R='SEL(B↑R,S)'
        B↑T=A↑T
END
```

SOURCE PROGRAM

```
DEC A,B,C,P;
DEC UNILIST=<ATOM:PRIM>;
DEC PAIR=<HD:PRIM / TL:LIST>;
DEC LIST=UNILIST & PAIR;
DEC UNITREE=<LEAF:NUM>;
DEC BINTREE=<NODE:NUM / LB:TREE / RB:TREE>;
DEC TREE=UNITREE & BINTREE;
A=(CONS BINTREE)(1,2,3);
B=(CONS BINTREE)(4,5,6);
C=(CONS PAIR)(1,(CONS PAIR)(PTR(A),3));
P=PTR(C);
IF (IS BINTREE)(VAL(HD(VAL(P)))) THEN HD(VAL(P))=PTR(B);
   ELSE REPEAT AFTER P=PTR(TL(VAL(P)));
END
```

```
LISTING OF ATTRIBUTES OF PROGRAM NODE


          DISPLAY OF FLOWCHART-F
1      ----------
FUN      U(U(U(UO("P",OMEGA),UO("C",OMEGA)),UO("B",OMEGA)),UO("A",OMEGA))
NEXT      38

38     ----------
FUN      U(S,UO("A",U(UO(TYPEL,"BINTREE" ),U(UO("NODE",1),U(UO("LB",2),UO("RB",3)
))))))
NEXT      39

39     ----------
FUN      U(S,UO("B",U(UO(TYPEL,"BINTREE" ),U(UO("NODE",4),U(UO("LB",5),UO("RB",6)
))))))
NEXT      40

40     ----------
FUN      U(S,UO("C",U(UO(TYPEL,"PAIR" ),U(UO("HD",1),UO("TL",U(UO(TYPEL,"PAIR" ),
U(UO("HD","A"),UO("TL",3))))))))
NEXT      41

41     ----------
FUN      U(S,UO("P","C"))
NEXT      42

42     ----------
FUN      ID(S)
NEXT      COND(ELEM(SEL(TYPEL,SEL(SEL("HD",SEL(SEL("P",S),S)),S)),SEL("BINTREE",I
DLIST)),43,45)

43     ----------
FUN      U1(S,COMPOS("HD",SEL("P",S)),"B")
NEXT

45     ----------
FUN      U(S,UO("P",COMPOS("TL",SEL("P",S))))
NEXT      42


          DISPLAY OF IDLIST-I
UNILIST     ----------
UNILIST       XXX

PAIR        ----------
PAIR        XXX

LIST        ----------
PAIR        XXX
UNILIST       XXX

UNITREE       ----------
UNITREE       XXX

BINTREE       ----------
BINTREE       XXX
```

TREE	----------
BINTREE	XXX
UNITREE	XXX

APPLY FLOWCHART TO NULL STATE VECTOR S

```
          FINAL STATE VECTOR
A      ----------
TYPE-       BINTREE
NODE       1
LB       2
RB       3

B      ----------
TYPE-       BINTREE
NODE       4
LB       5
RB       6

P       TL?C
C      ----------
TYPE-       PAIR
HD       1
    TL      ----------
TYPE-       PAIR
TL       3
HD       8
```

TIME TO PROCESS SEMANTIC DESCRIPTION 2156 MILLISEC
TIME TO PARSE 26398 MILLISEC
TIME TO EVALUATE SEMANTIC ATTRIBUTES 36820 MILLISEC
'EXECUTION' TIME 220 MILLISEC
'STATEMENTS' EXECUTED 9

SOURCE PROGRAM

```
DEC A,I,J,X;
DEC PERSON=<BIRTHDAY:NUM / DAD:PTR / YOUNGESTKID:PTR>;
A=(CONS PERSON)(1900,*,*);
I=(CONS PERSON)(1930,PTR(A),*);
YOUNGESTKID(A)=PTR(I);
J=(CONS PERSON)(1932,PTR(A),*);
YOUNGESTKID(A)=PTR(J);
X=(CONS PERSON)(1955,PTR(2),*);
YOUNGESTKID(I)=PTR(X);
END
```

```
LISTING OF ATTRIBUTES OF PROGRAM NODE


          DISPLAY OF FLOWCHART-F
63      ----------
FUN      U(U(U(UO("X",OMEGA),UO("J",OMEGA)),UO("I",OMEGA)),UO("A",OMEGA))
NEXT      73

73      ----------
FUN      U(S,UO("A",U(UO(TYPEL,"PERSON" ),U(UO("BIRTHDAY",1900),U(UO("DAD","*"),U
O("YOUNGESTKID","*"))))))
NEXT      74

74      ----------
FUN      U(S,UO("I",U(UO(TYPEL,"PERSON" ),U(UO("BIRTHDAY",1930),U(UO("DAD","A"),U
O("YOUNGESTKID","*"))))))
NEXT      75

75      ----------
FUN      U1(S,COMPOS("YOUNGESTKID","A"),"I")
NEXT      76

76      ----------
FUN      U(S,UO("J",U(UO(TYPEL,"PERSON" ),U(UO("BIRTHDAY",1932),U(UO("DAD","A"),U
O("YOUNGESTKID","*"))))))
NEXT      77

77      ----------
FUN      U1(S,COMPOS("YOUNGESTKID","A"),"J")
NEXT      78

78      ----------
FUN      U(S,UO("X",U(UO(TYPEL,"PERSON" ),U(UO("BIRTHDAY",1955),U(UO("DAD",NUL),U
O("YOUNGESTKID","*"))))))
NEXT      79

79      ----------
FUN      U1(S,COMPOS("YOUNGESTKID","I"),"X")
NEXT


          DISPLAY OF IDLIST-I
PERSON    ----------
PERSON     XXX
```

APPLY FLOWCHART TO NULL STATE VECTOR S

```
          FINAL STATE VECTOR
J     ----------
TYPE-      PERSON
BIRTHDAY      1932
DAD      A
YOUNGESTKID      *

A     ----------
TYPE-      PERSON
BIRTHDAY      1900
DAD      *
YOUNGESTKID      J

X     ----------
TYPE-      PERSON
BIRTHDAY      1955
DAD
YOUNGESTKID      *

I     ----------
TYPE-      PERSON
BIRTHDAY      1930
DAD      A
YOUNGESTKID      X

TIME TO PROCESS SEMANTIC DESCRIPTION 2156 MILLISEC
TIME TO PARSE 3486 MILLISEC
TIME TO EVALUATE SEMANTIC ATTRIBUTES 26504 MILLISEC
'EXECUTION' TIME 203 MILLISEC
'STATEMENTS' EXECUTED 8
```

MINI-LANGUAGE 9

SYNTACTIC AND SEMANTIC DESCRIPTION

```
<PROGRAM>A=<VARIABLE DEF LIST>B ';' <TRANSFORMATION DEF LIST>C ';'
. <RESULT EXPRESSION>D ';END'
        <EXPRESSION>A E=UO(O,U(UO('FUN','D↑E'),UO('NXT',NUL)))
        <VARIABLETABLE>A↑V=B↑V
        <TRANSTABLE>A↑T=C↑T
        <CONCATLIB>A↑C=U(B↑C,C↑C)

<VARIABLE DEF LIST>A=<VARIABLE DEFINITION>B ';' <VARIABLE DEF LIST>C;
.D=<VARIABLE DEFINITION>E
        <VARIABLETABLE>A↑V=U(B↑V,C↑V);D↑V=E↑V
        <CONCALIB>A↑C=U(B↑C,C↑C);D↑C=E↑C

<VARIABLE DEFINITION>A=<$LETTR>B '=' <PATTERN>C
        <VARIABLETABLE>A↑V=UO('B↑VAL',U(
        .MKFLC(0,'S','COND(EQU(SEL("B↑VAL" "LNX",S), LNX(SEL("STR",S))),1,2)'),
        .MKFLC(1,'"F"',),
        .MKFLC(2,'U1(S,"B↑VAL" "LNX", LNX(SEL("STR",S)))',3),
        .MKFLC(3,'C↑P',)
        .))
        <CONCALIB>A↑C=C↑C

<PATTERN>A=<SIMPLE PATTERN>B  ' / ' <PATTERN>C;D=<SIMPLE PATTERN>E
        <PREDICATE>A↑P='OR(B↑P,C↑P)';D↑P=E↑P
        <CONCALIB>A↑C=U(B↑C,C↑C);D↑C=E↑C

<SIMPLE PATTERN>A=<CONCAT LIST>B;C='&'
         <PREDICATE>A↑P='SEL("TF",A↑F)';C↑P='SEL("TF",C↑F)'
         <FUNCTION>A↑F='APPLI(SEL(A↑I,CONCATLIB),S)';C↑F='UO("TF",EQU(O,
         .LEN(SEL("STR",S))))'
         <ID>A↑I=UNO();C↑I='NUL'
         <CONCATLIB>A↑C=UO(A↑I,U(MKFLC(0,'U1(S,"N",1)',1),
         .MKFLC(1,'S','COND(B↑P,2,3)'),
         .MKFLC(2,'U(S,B↑F,UO("TF","T"))',),
         .MKFLC(3,'U1(S,"N",SEL("N",S) + 1)',4),
         .MKFLC(4,'S','COND(LEQ(SEL("N",S),PLIM(B↑T,LNX(SEL("STR",S)))),1,5)'),
         .MKFLC(5,'UO("TF","F")',)))
        B↑T=B↑L;B↑N=1;B↑V='NUL'

<CONCAT LIST>A=<PATTERN ELEMENT>B <CONCAT LIST>C;D=<PATTERN ELEMENT>E
        <TOTALEN>T
        <NUMBER>N
        <VARTABLE>V
        <LENGTH>A↑L=1 + C↑L;D↑L=1
        <PREDICATE>A↑P='AND(B↑P,C↑P)';D↑P=E↑P
        <FUNCTION>A↑F='U(B↑F,C↑F)';D↑F=E↑F
        B↑T=A↑T;B↑N=A↑N;B↑V=A↑V
        C↑T=A↑T;C↑N=A↑N + 1;C↑V=U(A↑V,B↑X)
        E↑T=D↑T;E↑N=D↑N;E↑V=D↑V

<PATTERN ELEMENT>A='-' <$ALPHX>B '-';C=<$LETTR>D
        <TOTALEN>T
        <NUMBER>N
        <VARTABLE>V
        <XVARTABLEENTRY>A↑X='NUL';C↑X=UO('D↑VAL',C↑N)
        <SPECIAL>A↑P='P1K("B↑VAL",A↑T,A↑N)';
```

```
        .C↑P=COND(ELEM('D↑VAL',C↑V),    'P2K(' SEL('D↑VAL',C↑V) ',C↑T,C↑N)'
           .,'P3K("D↑VAL",C↑T,C↑N)')
            <PREDICATE>A↑Z='IDEN("B↑VAL",SUBSTR(PARTITION(A↑T,2 * A↑N - 1,
           .SEL("N",S),LNX(SEL("STR",S))),PARTITION(A T,2 * A↑N,SEL("N",S),
           .LNX(SEL("STR",S))),SEL("STR",S))';
           .C↑Z=COND(ELEM('D↑VAL',C↑V),
           .'IDEN(SUBSTR(PARTITION(C↑T,2 * C↑N - 1,SEL("N",S),LNX(SEL("STR",S))),
           .PARTITION(C↑T,2 * C↑N,SEL("N",S),LNX(SEL("STR",S))),SEL("STR",S)),
           .SUBSTR(PARTITION(C↑T,2 * ' SEL('D↑VAL',C↑V) ' - 1,SEL("N",S),
           .LNX(SEL("STR",S))),PARTITION(C↑T,2 * '  SEL('D↑VAL',C↑V) ',SEL("N",S),
           .LNX(SEL("STR",S))),SEL("STR",S))',
           .'APPLI(SEL("D↑VAL",VARLIB),U1(S,"STR",SUBSTR(PARTITION(C↑T,2 * C↑N - 1
           .,
           .SEL("N",S),LNX(SEL("STR",S))),PARTITION(C↑T,2 * C↑N,SEL("N",S),
           .LNX(SEL("STR",S))),SEL("STR",S))))')
           <FUNCTION>A↑F='NUL';C↑F='UO("D↑VAL",SUBSTR(PARTITION(C↑T,2 * C↑N - 1,
           .SEL("N",S),LNX(SEL("STR",S))),PARTITION(C↑T,2 * C↑N,SEL("N",S),
           .LNX(SEL("STR",S))),SEL("STR",S))'

<TRANSFORMATION DEF LIST>D=<TRANSFORMATION DEFINITION>E;
.<TRANSFORMATION DEF LIST>A=<TRANSFORMATION DEFINITION>B ';'
. <TRANSFORMATION DEF LIST>C
           <TRANSTABLE>A↑T=U(B↑T,C↑T);D↑T=E↑T
           <CONCATLIB>A↑C=U(B↑C,C↑C);D↑C=E↑C

<TRANSFORMATION DEFINITION>A='LET T' <$NUMBR>B '=<<' <ACTION LIST>C '>>'
           <CONCATLIB>A↑C=C↑C
           <TRANSTABLE>A↑T=UO(B↑VAL,U(MKFLC(0,'S',C↑L),C↑F))
           C↑N='NUL'

<ACTION LIST>A=<ACTION>B ';' <ACTION LIST>C;D=<ACTION>E
           <NEXT>N
           <FLOWCHART>A↑F=U(B↑F,C↑F);D↑F=E↑F
           <LINENO>A↑L=B↑L;D↑L=E↑L
           B↑N=C↑L
           C↑N=A↑N
           E↑N=D↑N
           <CONCATLIB>A↑C=U(B↑C,C↑C);D↑C=E↑C

<ACTION>A=<SIMPLE PATTERN>B ' #' <TERMINATION CHARACTER>C <REPLACEMENT STRING>D
           <NEXT>N
           <CONCATLIB>A↑C=B↑C
           <LINENO>A↑L=UNO()
           <UNIQUENOS>A↑U=U(UO(1,UNO()),UO(2,UNO()),UO(3,UNO()),UO(4,UNO()),
           .UO(5,UNO()))
           <S>A↑S=UNO()
           <KLUDGE>A↑F=U(A↑X,A↑Y)
           <FLOWCHART>A↑X=U(
           .MKFLC(A↑L,  'U(UO("P1",1),UO("P2",1),UO("STR",S))',SEL(1,A↑U)),
           .MKFLC(SEL(1,A↑U),'S','COND(APPLI( B↑P ,U1(S,"STR",SUBSTR(SEL("P1",S),
           .SEL("P2",S),SEL("STR",S)))),' SEL(2,A↑U) ',' SEL(3,A↑U) ')'),
           .MKFLC(SEL(2,A↑U),'SUBSTR(1,SEL("P1",S)    ,SEL("STR",S))
          . APPLI( D↑F ,APPLI("B↑F",U1(S,"STR",SUBSTR(SEL("P1",S),SEL("P2",S),SEL("S
          .TR",
           .S)))))
           . SUBSTR(SEL("P2",S),LNX(SEL("STR",S)) + 1,SEL("STR",S))',
           .COND('C↑T',,0)),
           .)
           <Y>A↑Y=U(
           .MKFLC(SEL(3,A↑U),'U1(S,"P2",SEL("P2",S) + 1)','COND(LEQ(SEL("P2",S) -
```

```
          . 1,
          .LNX(SEL("STR",S))),' SEL(1,A↑U) ',' SEL(4,A↑U) ')'),
          .MKFLC(SEL(4,A↑U),'U(S,UO("P1",SEL("P1",S) + 1),UO("P2",SEL("P1",S) + 2
          .))',
          .'COND(LEQ(SEL("P1",S),LNX(SEL("STR",S))),' SEL(1,A↑U) ',' SEL(5,A↑U)
          . ')'),
          .MKFLC(SEL(5,A↑U),'SEL("STR",S)',A↑N))

<TERMINATION CHARACTER>A=' ';B=' .'
          <TERMINAL>A↑T='F';B↑T='T'

<REPLACEMENT STRING>A=<STRING REF LIST>B;C='&'
          <FUNCTION>A↑F=B↑F;C↑F='NUL'

<STRING REF LIST>A=<STRING REF>B <STRING REF LIST>C;D=<STRING REF>E
          <FUNCTION>A↑F='B↑F C↑F';D↑F=E↑F

<STRING REF>A='-' <$ALPHX>B '-';C=<$LETTR>D
          <FUNCTION>A↑F='"B↑VAL"';
          .C↑F='SEL("D↑VAL",S)'

<RESULT EXPRESSION>A='-' <$ALPHX>B '-';C='T' <$NUMBR>D '<' <RESULT EXPRESSION>E
. '>'
          <EXPRESSION>A↑E='"B↑VAL"';C↑E='APPLI(SEL(D↑VAL,TRANSLIB),E↑E)'
END
```

SOURCE PROGRAM

```
DEC A,B,C,P;
DEC UNILIST=<ATOM:PRIM>;
DEC PAIR=<HD:PRIM / TL:LIST>;
DEC LIST=UNILIST & PAIR;
DEC UNITREE=<LEAF:NUM>;
DEC BINTREE=<NODE:NUM / LB:TREE / RB:TREE>;
DEC TREE=UNITREE & BINTREE;
A=(CONS BINTREE)(1,2,3);
B=(CONS BINTREE)(4,5,6);
C=(CONS PAIR)(1,(CONS PAIR)(PTR(A),3));
P=PTR(C);
IF (IS BINTREE)(VAL(HD(VAL(P)))) THEN HD(VAL(P))=PTR(B);
   ELSE REPEAT AFTER P=PTR(TL(VAL(P)));
END
```

```
LISTING OF ATTRIBUTES OF PROGRAM NODE


          DISPLAY OF FLOWCHART-F
1       ----------
FUN       U(U(U(U(UO("P",OMEGA),UO("C",OMEGA)),UO("B",OMEGA)),UO("A",OMEGA))
NEXT       38

38      ----------
FUN       U(S,UO("A",U(UO(TYPEL,"BINTREE" ),U(UO("NODE",1),U(UO("LB",2),UO("RB",3)
)))))
NEXT       39

39      ----------
FUN       U(S,UO("B",U(UO(TYPEL,"BINTREE" ),U(UO("NODE",4),U(UO("LB",5),UO("RB",6)
)))))
NEXT       40

40      ----------
FUN       U(S,UO("C",U(UO(TYPEL,"PAIR" ),U(UO("HD",1),UO("TL",U(UO(TYPEL,"PAIR" ),
U(UO("HD","A"),UO("TL",3)))))))))
NEXT       41

41      ----------
FUN       U(S,UO("P","C"))
NEXT       42

42      ----------
FUN       ID(S)
NEXT       COND(ELEM(SEL(TYPEL,SEL(SEL("HD",SEL(SEL("P",S),S)),S)),SEL("BINTREE",I
DLIST)),43,45)

43      ----------
FUN       U1(S,COMPOS("HD",SEL("P",S)),"B")
NEXT

45      ----------
FUN       U(S,UO("P",COMPOS("TL",SEL("P",S))))
NEXT       42


          DISPLAY OF IDLIST-I
UNILIST     ----------
UNILIST       XXX

PAIR        ----------
PAIR        XXX

LIST        ----------
PAIR        XXX
UNILIST       XXX

UNITREE       ----------
UNITREE       XXX

BINTREE       ----------
BINTREE       XXX
```

```
TREE      ----------
BINTREE     XXX
UNITREE     XXX
```

APPLY FLOWCHART TO NULL STATE VECTOR S

FINAL STATE VECTOR

```
A       ----------
TYPE-       BINTREE
NODE        1
LB      2
RB      3

B       ----------
TYPE-       BINTREE
NODE        4
LB      5
RB      6

P       TL?C
C       ----------
TYPE-       PAIR
HD      1
      TL        ----------
TYPE-       PAIR
TL      3
HD      8
```

TIME TO PROCESS SEMANTIC DESCRIPTION 2156 MILLISEC
TIME TO PARSE 26398 MILLISEC
TIME TO EVALUATE SEMANTIC ATTRIBUTES 36820 MILLISEC
'EXECUTION' TIME 220 MILLISEC
'STATEMENTS' EXECUTED 9

SOURCE PROGRAM

```
DEC A,I,J,X;
DEC PERSON=<BIRTHDAY:NUM / DAD:PTR / YOUNGESTKID:PTR>;
A=(CONS PERSON)(1900,*,*);
I=(CONS PERSON)(1930,PTR(A),*);
YOUNGESTKID(A)=PTR(I);
J=(CONS PERSON)(1932,PTR(A),*);
YOUNGESTKID(A)=PTR(J);
X=(CONS PERSON)(1955,PTR(2),*);
YOUNGESTKID(I)=PTR(X);
END
```

```
LISTING OF ATTRIBUTES OF PROGRAM NODE


          DISPLAY OF FLOWCHART-F
63      ----------
FUN      U(U(U(U(UO("X",OMEGA),UO("J",OMEGA)),UO("I",OMEGA)),UO("A",OMEGA))
NEXT      73

73      ----------
FUN      U(S,UO("A",U(UO(TYPEL,"PERSON" ),U(UO("BIRTHDAY",1900),U(UO("DAD","*"),U
O("YOUNGESTKID","*"))))))
NEXT      74

74      ----------
FUN      U(S,UO("I",U(UO(TYPEL,"PERSON" ),U(UO("BIRTHDAY",1930),U(UO("DAD","A"),U
O("YOUNGESTKID","*"))))))
NEXT      75

75      ----------
FUN      U1(S,COMPOS("YOUNGESTKID","A"),"I")
NEXT      76

76      ----------
FUN      U(S,UO("J",U(UO(TYPEL,"PERSON" ),U(UO("BIRTHDAY",1932),U(UO("DAD","A"),U
O("YOUNGESTKID","*"))))))
NEXT      77

77      ----------
FUN      U1(S,COMPOS("YOUNGESTKID","A"),"J")
NEXT      78

78      ----------
FUN      U(S,UO("X",U(UO(TYPEL,"PERSON" ),U(UO("BIRTHDAY",1955),U(UO("DAD",NUL),U
O("YOUNGESTKID","*"))))))
NEXT      79

79      ----------
FUN      U1(S,COMPOS("YOUNGESTKID","I"),"X")
NEXT


          DISPLAY OF IDLIST-I
PERSON    ----------
PERSON     XXX
```

```
APPLY FLOWCHART TO NULL STATE VECTOR S


          FINAL STATE VECTOR
J     ----------
TYPE-       PERSON
BIRTHDAY      1932
DAD       A
YOUNGESTKID       *

A     ----------
TYPE--      PERSON
BIRTHDAY      1900
DAD       *
YOUNGESTKID       J

X     ----------
TYPE-       PERSON
BIRTHDAY      1955
DAD
YOUNGESTKID       *

I     ----------
TYPE-       PERSON
BIRTHDAY      1930
DAD       A
YOUNGESTKID       X


TIME TO PROCESS SEMANTIC DESCRIPTION 2156 MILLISEC
TIME TO PARSE 3486 MILLISEC
TIME TO EVALUATE SEMANTIC ATTRIBUTES 26504 MILLISEC
*EXECUTION* TIME 203 MILLISEC
*STATEMENTS* EXECUTED 8
```

```
                        SOURCE PROGRAM
L=-N- / -I- / -X- / -O-;
L=M;
LET T1=<<LM-*- # M-*-L;
          -(-M-*- # M-(-;
          -()- # .&;
          -)- # -*)->>;
T1<-(NOXIN)->;
END
```

```
LISTING OF ATTRIBUTES OF PROGRAM NODE

          DISPLAY OF EXPRESSION-E
0      ----------
FUN      APPLI(SEL(1,TRANSLIB),"(NOXIN)")
NXT


          DISPLAY OF VARIABLETABLE-V
L      ----------
     0     ----------
FUN      S
NXT      COND(EQU(SEL("L" "LNX",S), LNX(SEL("STR",S))),1,2)

     1     ----------
FUN      "F"
NXT

     2     ----------
FUN      U1(S,"L" "LNX", LNX(SEL("STR",S)))
NXT      3

     3     ----------
FUN      SEL("TF",APPLI(SEL(5,CONCATLIB),S))
NXT


          DISPLAY OF TRANSTABLE-T
1      ----------
     0     ----------
FUN      S
NXT      7

     7     ----------
FUN      U(U0("P1",1),U0("P2",1),U0("STR",S))
NXT      8

     8     ----------
FUN      S
NXT      COND(APPLI('SEL("TF",APPLI(SEL(15,CONCATLIB),S))',U1(S,"STR",SUBSTR(SEL(
"P1",S),SEL("P2",S),SEL("STR",S)))),9,10)

     9     ----------
FUN      SUBSTR(1,SEL("P1",S)     ,SEL("STR",S)) APPLI('SEL("M",S) "*" SEL("L",S)'
,APPLI("APPLI(SEL(15,CONCATLIB),S)",U1(S,"STR",SUBSTR(SEL("P1",S),SEL("P2",S),SE
L("STR",S))))) SUBSTR(SEL("P2",S),LNX(SEL("STR",S)) + 1,SEL("STR",S))
NXT      0

    10     ----------
FUN      U1(S,"P2",SEL("P2",S) + 1)
NXT      COND(LEQ(SEL("P2",S) - 1,LNX(SEL("STR",S))),8,11)

    11     ----------
FUN      U(S,U0("P1",SEL("P1",S) + 1),U0("P2",SEL("P1",S) + 2))
NXT      COND(LEQ(SEL("P1",S),LNX(SEL("STR",S))),8,12)
```

```
     12      ----------
FUN      SEL("STR",S)
NXT      19

     19      ----------
FUN      U(UO("P1",1),UO("P2",1),UO("STR",S))
NXT      20

     20      ----------
FUN      S
NXT      COND(APPLI('SEL("TF",APPLI(SEL(27,CONCATLIB),S))',U1(S,"STR",SUBSTR(SEL(
"P1",S),SEL("P2",S),SEL("STR",S)))),21,22)

     21      ----------
FUN      SUBSTR(1,SEL("P1",S)     ,SEL("STR",S)) APPLI('SEL("M",S) "(",APPLI("APP
LI(SEL(27,CONCATLIB),S)",U1(S,"STR",SUBSTR(SEL("P1",S),SEL("P2",S),SEL("STR",S))
))) SUBSTR(SEL("P2",S),LNX(SEL("STR",S)) + 1,SEL("STR",S))
NXT      0

     22      ----------
FUN      U1(S,"P2",SEL("P2",S) + 1)
NXT      COND(LEQ(SEL("P2",S) - 1,LNX(SEL("STR",S))),20,23)

     23      ----------
FUN      U(S,UO("P1",SEL("P1",S) + 1),UO("P2",SEL("P1",S) + 2))
NXT      COND(LEQ(SEL("P1",S),LNX(SEL("STR",S))),20,24)

     24      ----------
FUN      SEL("STR",S)
NXT      29

     29      ----------
FUN      U(UO("P1",1),UO("P2",1),UO("STR",S))
NXT      30

     30      ----------
FUN      S
NXT      COND(APPLI('SEL("TF",APPLI(SEL(37,CONCATLIB),S))',U1(S,"STR",SUBSTR(SEL(
"P1",S),SEL("P2",S),SEL("STR",S)))),31,32)

     31      ----------
FUN      SUBSTR(1,SEL("P1",S)     ,SEL("STR",S)) APPLI('NUL',APPLI("APPLI(SEL(37,C
ONCATLIB),S)",U1(S,"STR",SUBSTR(SEL("P1",S),SEL("P2",S),SEL("STR",S))))) SUBSTR(
SEL("P2",S),LNX(SEL("STR",S)) + 1,SEL("STR",S))
NXT

     32      ----------
FUN      U1(S,"P2",SEL("P2",S) + 1)
NXT      COND(LEQ(SEL("P2",S) - 1,LNX(SEL("STR",S))),30,33)

     33      ----------
FUN      U(S,UO("P1",SEL("P1",S) + 1),UO("P2",SEL("P1",S) + 2))
NXT      COND(LEQ(SEL("P1",S),LNX(SEL("STR",S))),30,34)

     34      ----------
FUN      SEL("STR",S)
NXT      38

     38      ----------
```

```
FUN     U(UO("P1",1),UO("P2",1),UO("STR",S))
NXT     39

    39      ----------
FUN     S
NXT     COND(APPLI('SEL("TF",APPLI(SEL(47,CONCATLIB),S))',U1(S,"STR",SUBSTR(SEL(
"P1",S),SEL("P2",S),SEL("STR",S)))),40,41)

    40      ----------
FUN     SUBSTR(1,SEL("P1",S)     ,SEL("STR",S)) APPLI('"*)"',APPLI("APPLI(SEL(47,
CONCATLIB),S)",U1(S,"STR",SUBSTR(SEL("P1",S),SEL("P2",S),SEL("STR",S)))))) SUBSTR
(SEL("P2",S),LNX(SEL("STR",S)) + 1,SEL("STR",S))
NXT     0

    41      ----------
FUN     U1(S,"P2",SEL("P2",S) + 1)
NXT     COND(LEQ(SEL("P2",S) - 1,LNX(SEL("STR",S))),39,42)

    42      ----------
FUN     U(S,UO("P1",SEL("P1",S) + 1),UO("P2",SEL("P1",S) + 2))
NXT     COND(LEQ(SEL("P1",S),LNX(SEL("STR",S))),39,43)

    43      ----------
FUN     SEL("STR",S)
NXT
```

DISPLAY OF CONCATLIB-C

```
1       ----------
    0       ----------
FUN     U1(S,"N",1)
NXT     1

    1       ----------
FUN     S
NXT     COND(P1K("N",1,1),2,3)

    2       ----------
FUN     U(S,NUL,UO("TF","T"))
NXT

    3       ----------
FUN     U1(S,"N",SEL("N",S) + 1)
NXT     4

    4       ----------
FUN     S
NXT     COND(LEQ(SEL("N",S),PLIM(1,LNX(SEL("STR",S)))),1,5)

    5       ----------
FUN     UO("TF","F")
NXT


2       ----------
    0       ----------
FUN     U1(S,"N",1)
NXT     1
```

```
      1      ----------
FUN      S
NXT      COND(P1K("I",1,1),2,3)

      2      ----------
FUN      U(S,NUL,U0("TF","T"))
NXT

      3      ----------
FUN      U1(S,"N",SEL("N",S) + 1)
NXT      4

      4      ----------
FUN      S
NXT      COND(LEQ(SEL("N",S),PLIM(1,LNX(SEL("STR",S)))),1,5)

      5      ----------
FUN      U0("TF","F")
NXT


3     ----------
      0      ----------
FUN      U1(S,"N",1)
NXT      1

      1      ----------
FUN      S
NXT      COND(P1K("X",1,1),2,3)

      2      ----------
FUN      U(S,NUL,U0("TF","T"))
NXT

      3      ----------
FUN      U1(S,"N",SEL("N",S) + 1)
NXT      4

      4      ----------
FUN      S
NXT      COND(LEQ(SEL("N",S),PLIM(1,LNX(SEL("STR",S)))),1,5)

      5      ----------
FUN      U0("TF","F")
NXT


4     ----------
      0      ----------
FUN      U1(S,"N",1)
NXT      1

      1      ----------
FUN      S
NXT      COND(P1K("O",1,1),2,3)

      2      ----------
FUN      U(S,NUL,U0("TF","T"))
NXT
```

```
     3      ----------
FUN      U1(S,"N",SEL("N",S) + 1)
NXT      4

     4      ----------
FUN      S
NXT      COND(LEQ(SEL("N",S),PLIM(1,LNX(SEL("STR",S)))),1,5)

     5      ----------
FUN      UO("TF","F")
NXT


5      ----------
     0      ----------
FUN      U1(S,"N",1)
NXT      1

     1      ----------
FUN      S
NXT      COND(P3K("M",1,1),2,3)

     2      ----------
FUN      U(S,UO("M",SUBSTR(PARTITION(1,2 * 1 - 1,SEL("N",S),LNX(SEL("STR",S))),PA
RTITION(1,2 * 1,SEL("N",S),LNX(SEL("STR",S))),SEL("STR",S))),UO("TF","T"))
NXT

     3      ----------
FUN      U1(S,"N",SEL("N",S) + 1)
NXT      4

     4      ----------
FUN      S
NXT      COND(LEQ(SEL("N",S),PLIM(1,LNX(SEL("STR",S)))),1,5)

     5      ----------
FUN      UO("TF","F")
NXT


15     ----------
     0      ----------
FUN      U1(S,"N",1)
NXT      1

     1      ----------
FUN      S
NXT      COND(AND(P3K("L",3,1),AND(P3K("M",3,2),P1K("*",3,3))),2,3)

     2      ----------
FUN      U(S,U(UO("L",SUBSTR(PARTITION(3,2 * 1 - 1,SEL("N",S),LNX(SEL("STR",S))),
PARTITION(3,2 * 1,SEL("N",S),LNX(SEL("STR",S))),SEL("STR",S))),U(UO("M",SUBSTR(P
ARTITION(3,2 * 2 - 1,SEL("N",S),LNX(SEL("STR",S))),PARTITION(3,2 * 2,SEL("N",S),
LNX(SEL("STR",S))),SEL("STR",S))),NUL)),UO("TF","T"))
NXT

     3      ----------
FUN      U1(S,"N",SEL("N",S) + 1)
NXT      4
```

```
     4     ----------
FUN        S
NXT        COND(LEQ(SEL("N",S),PLIM(3,LNX(SEL("STR",S)))),1,5)

     5     ----------
FUN        UO("TF","F")
NXT


27   ----------
     0     ----------
FUN        U1(S,"N",1)
NXT        1

     1     ----------
FUN        S
NXT        COND(AND(P1K("(",3,1),AND(P3K("M",3,2),P1K("*",3,3))),2,3)

     2     ----------
FUN        U(S,U(NUL,U(UO("M",SUBSTR(PARTITION(3,2 * 2 - 1,SEL("N",S),LNX(SEL("STR"
,S))),PARTITION(3,2 * 2,SEL("N",S),LNX(SEL("STR",S))),SEL("STR",S))),NUL)),UO("T
F","T"))
NXT

     3     ----------
FUN        U1(S,"N",SEL("N",S) + 1)
NXT        4

     4     ----------
FUN        S
NXT        COND(LEQ(SEL("N",S),PLIM(3,LNX(SEL("STR",S)))),1,5)

     5     ----------
FUN        UO("TF","F")
NXT


37   ----------
     0     ----------
FUN        U1(S,"N",1)
NXT        1

     1     ----------
FUN        S
NXT        COND(P1K(")","N",1),2,3)

     2     ----------
FUN        U(S,NUL,UO("TF","T"))
NXT

     3     ----------
FUN        U1(S,"N",SEL("N",S) + 1)
NXT        4

     4     ----------
FUN        S
NXT        COND(LEQ(SEL("N",S),PLIM(1,LNX(SEL("STR",S)))),1,5)

     5     ----------
```

```
FUN     U0("TF","F")
NXT

47      ----------
     0      ----------
FUN     U1(S,"N",1)
NXT     1

     1      ----------
FUN     S
NXT     COND(P1K(")",1,1),2,3)

     2      ----------
FUN     U(S,NUL,U0("TF","T"))
NXT

     3      ----------
FUN     U1(S,"N",SEL("N",S) + 1)
NXT     4

     4      ----------
FUN     S
NXT     COND(LEQ(SEL("N",S),PLIM(1,LNX(SEL("STR",S)))),1,5)

     5      ----------
FUN     U0("TF","F")
NXT
```

APPLY FLOWCHART TO NULL STATE VECTOR S

FINAL STATE VECTOR

NIXON

TIME TO PROCESS SEMANTIC DESCRIPTION 1834 MLLLISEC
TIME TO PARSE 2488 MILLISEC
TIME TO EVALUATE SEMANTIC ATTRIBUTES 37662 MILLISEC
'EXECUTION' TIME 3450036 MILLISEC
'STATEMENTS' EXECUTED 198044 MILLISEC

SOURCE PROGRAM

```
L=-H- / -E- / -S- / -M- / -A- / -N- / -K- / -F-;
V=W;
W=L / LW;
LET T1=<<V-,-W-*- # W-*,-V;
          -(-W-*,- # W-,(-;
          -(-W-*)- # W;
          -( )- # .&;
          -)- # -*)->>;
T1<-(HESSE,KAFKA,MANN)->;
END
```

```
LISTING OF ATTRIBUTES OF PROGRAM NODE


          DISPLAY OF EXPRESSION-E
0     ----------
FUN      APPLI(SEL(1,TRANSLIB),"(HESSE,KAFKA,MANN)")
NXT


          DISPLAY OF VARIABLETABLE-V
L     ----------
    0     ----------
FUN      S
NXT      COND(EQU(SEL("L" "LNX",S), LNX(SEL("STR",S))),1,2)

    1     ----------
FUN      "F"
NXT

    2     ----------
FUN      U1(S,"L" "LNX", LNX(SEL("STR",S)))
NXT      3

    3     ----------
FUN      OR(SEL("TF",APPLI(SEL(86,CONCATLIB),S)),OR(SEL("TF",APPLI(SEL(87,CONCATL
IB),S)),OR(SEL("TF",APPLI(SEL(88,CONCATLIB),S)),OR(SEL("TF",APPLI(SEL(89,CONCATL
IB),S)),OR(SEL("TF",APPLI(SEL(90,CONCATLIB),S)),OR(SEL("TF",APPLI(SEL(91,CONCATL
IB),S)),OR(SEL("TF",APPLI(SEL(92,CONCATLIB),S)),SEL("TF",APPLI(SEL(93,CONCATLIB)
,S)))))))))
NXT


V     ----------
    0     ----------
FUN      S
NXT      COND(EQU(SEL("V" "LNX",S), LNX(SEL("STR",S))),1,2)

    1     ----------
FUN      "F"
NXT

    2     ----------
FUN      U1(S,"V" "LNX", LNX(SEL("STR",S)))
NXT      3

    3     ----------
FUN      SEL("TF",APPLI(SEL(94,CONCATLIB),S))
NXT


W     ----------
    0     ----------
FUN      S
NXT      COND(EQU(SEL("W" "LNX",S), LNX(SEL("STR",S))),1,2)

    1     ----------
FUN      "F"
NXT
```

```
     2      ----------
FUN      U1(S,"W" "LNX", LNX(SEL("STR",S)))
NXT      3

     3      ----------
FUN      OR(SEL("TF",APPLI(SEL(96,CONCATLIB),S)),SEL("TF",APPLI(SEL(98,CONCATLIB)
,S)))
NXT


          DISPLAY OF TRANSTABLE-T
1     ----------
     0      ----------
FUN      S
NXT      102

     102      ----------
FUN      U(UO("P1",1),UO("P2",1),UO("STR",S))
NXT      103

     103      ----------
FUN      S
NXT      COND(APPLI('SEL("TF",APPLI(SEL(110,CONCATLIB),S))',U1(S,"STR",SUBSTR(SEL
("P1",S),SEL("P2",S),SEL("STR",S)))),104,105)

     104      ----------
FUN      SUBSTR(1,SEL("P1",S)     ,SEL("STR",S)) APPLI('SEL("W",S) "*," SEL("V",S)
',APPLI("APPLI(SEL(110,CONCATLIB),S)",U1(S,"STR",SUBSTR(SEL("P1",S),SEL("P2",S),
SEL("STR",S))))) SUBSTR(SEL("P2",S),LNX(SEL("STR",S)) + 1,SEL("STR",S))
NXT      0

     105      ----------
FUN      U1(S,"P2",SEL("P2",S) + 1)
NXT      COND(LEQ(SEL("P2",S) - 1,LNX(SEL("STR",S))),103,106)

     106      ----------
FUN      U(S,UO("P1",SEL("P1",S) + 1),UO("P2",SEL("P1",S) + 2))
NXT      COND(LEQ(SEL("P1",S),LNX(SEL("STR",S))),103,107)

     107      ----------
FUN      SEL("STR",S)
NXT      114

     114      ----------
FUN      U(UO("P1",1),UO("P2",1),UO("STR",S))
NXT      115

     115      ----------
FUN      S
NXT      COND(APPLI('SEL("TF",APPLI(SEL(122,CONCATLIB),S))',U1(S,"STR",SUBSTR(SEL
("P1",S),SEL("P2",S),SEL("STR",S)))),116,117)

     116      ----------
FUN      SUBSTR(1,SEL("P1",S)     ,SEL("STR",S)) APPLI('SEL("W",S) ",("',APPLI("AP
PLI(SEL(122,CONCATLIB),S)",U1(S,"STR",SUBSTR(SEL("P1",S),SEL("P2",S),SEL("STR",S
))))) SUBSTR(SEL("P2",S),LNX(SEL("STR",S)) + 1,SEL("STR",S))
NXT      0
```

```
     117     ----------
FUN      U1(S,"P2",SEL("P2",S) + 1)
NXT      COND(LEQ(SEL("P2",S) - 1,LNX(SEL("STR",S))),115,118)

     118     ----------
FUN      U(S,UO("P1",SEL("P1",S) + 1),UO("P2",SEL("P1",S) + 2))
NXT      COND(LEQ(SEL("P1",S),LNX(SEL("STR",S))),115,119)

     119     ----------
FUN      SEL("STR",S)
NXT      124

     124     ----------
FUN      U(UO("P1",1),UO("P2",1),UO("STR",S))
NXT      125

     125     ----------
FUN      S
NXT      COND(APPLI('SEL("TF",APPLI(SEL(132,CONCATLIB),S))',U1(S,"STR",SUBSTR(SEL
("P1",S),SEL("P2",S),SEL("STR",S)))),126,127)

     126     ----------
FUN      SUBSTR(1,SEL("P1",S)     ,SEL("STR",S)) APPLI('SEL("W",S)',APPLI("APPLI(S
EL(132,CONCATLIB),S)",U1(S,"STR",SUBSTR(SEL("P1",S),SEL("P2",S),SEL("STR",S)))))
 SUBSTR(SEL("P2",S),LNX(SEL("STR",S)) + 1,SEL("STR",S))
NXT      0

     127     ----------
FUN      U1(S,"P2",SEL("P2",S) + 1)
NXT      COND(LEQ(SEL("P2",S) - 1,LNX(SEL("STR",S))),125,128)

     128     ----------
FUN      U(S,UO("P1",SEL("P1",S) + 1),UO("P2",SEL("P1",S) + 2))
NXT      COND(LEQ(SEL("P1",S),LNX(SEL("STR",S))),125,129)

     129     ----------
FUN      SEL("STR",S)
NXT      134

     134     ----------
FUN      U(UO("P1",1),UO("P2",1),UO("STR",S))
NXT      135

     135     ----------
FUN      S
NXT      COND(APPLI('SEL("TF",APPLI(SEL(142,CONCATLIB),S))',U1(S,"STR",SUBSTR(SEL
("P1",S),SEL("P2",S),SEL("STR",S)))),136,137)

     136     ----------
FUN      SUBSTR(1,SEL("P1",S)     ,SEL("STR",S)) APPLI('NUL',APPLI("APPLI(SEL(142,
CONCATLIB),S)",U1(S,"STR",SUBSTR(SEL("P1",S),SEL("P2",S),SEL("STR",S))))) SUBSTR
(SEL("P2",S),LNX(SEL("STR",S)) + 1,SEL("STR",S))
NXT

     137     ----------
FUN      U1(S,"P2",SEL("P2",S) + 1)
NXT      COND(LEQ(SEL("P2",S) - 1,LNX(SEL("STR",S))),135,138)

     138     ----------
```

```
FUN     U(S,U0("P1",SEL("P1",S) + 1),U0("P2",SEL("P1",S) + 2))
NXT     COND(LEQ(SEL("P1",S),LNX(SEL("STR",S))),135,139)

    139     ----------
FUN     SEL("STR",S)
NXT     143

    143     ----------
FUN     U(U0("P1",1),U0("P2",1),U0("STR",S))
NXT     144

    144     ----------
FUN     S
NXT     COND(APPLI('SEL("TF",APPLI(SEL(152,CONCATLIB),S))',U1(S,"STR",SUBSTR(SEL
("P1",S),SEL("P2",S),SEL("STR",S)))),145,146)

    145     ----------
FUN     SUBSTR(1,SEL("P1",S)     ,SEL("STR",S)) APPLI('"*)"',APPLI("APPLI(SEL(152
,CONCATLIB),S)",U1(S,"STR",SUBSTR(SEL("P1",S),SEL("P2",S),SEL("STR",S)))))) SUBST
R(SEL("P2",S),LNX(SEL("STR",S)) + 1,SEL("STR",S))
NXT     0

    146     ----------
FUN     U1(S,"P2",SEL("P2",S) + 1)
NXT     COND(LEQ(SEL("P2",S) - 1,LNX(SEL("STR",S))),144,147)

    147     ----------
FUN     U(S,U0("P1",SEL("P1",S) + 1),U0("P2",SEL("P1",S) + 2))
NXT     COND(LEQ(SEL("P1",S),LNX(SEL("STR",S))),144,148)

    148     ----------
FUN     SEL("STR",S)
NXT
```

```
          DISPLAY OF CONCATLIB-C
86      ----------
    0       ----------
FUN     U1(S,"N",1)
NXT     1

    1       ----------
FUN     S
NXT     COND(P1K("H",1,1),2,3)

    2       ----------
FUN     U(S,NUL,U0("TF","T"))
NXT

    3       ----------
FUN     U1(S,"N",SEL("N",S) + 1)
NXT     4

    4       ----------
FUN     S
NXT     COND(LEQ(SEL("N",S),PLIM(1,LNX(SEL("STR",S)))),1,5)

    5       ----------
```

```
FUN     UO("TF","F")
NXT

87    ----------
    0     ----------
FUN     U1(S,"N",1)
NXT     1

    1     ----------
FUN     S
NXT     COND(P1K("E",1,1),2,3)

    2     ----------
FUN     U(S,NUL,UO("TF","T"))
NXT

    3     ----------
FUN     U1(S,"N",SEL("N",S) + 1)
NXT     4

    4     ----------
FUN     S
NXT     COND(LEQ(SEL("N",S),PLIM(1,LNX(SEL("STR",S)))),1,5)

    5     ----------
FUN     UO("TF","F")
NXT

88    ----------
    0     ----------
FUN     U1(S,"N",1)
NXT     1

    1     ----------
FUN     S
NXT     COND(P1K("S",1,1),2,3)

    2     ----------
FUN     U(S,NUL,UO("TF","T"))
NXT

    3     ----------
FUN     U1(S,"N",SEL("N",S) + 1)
NXT     4

    4     ----------
FUN     S
NXT     COND(LEQ(SEL("N",S),PLIM(1,LNX(SEL("STR",S)))),1,5)

    5     ----------
FUN     UO("TF","F")
NXT

89    ----------
    0     ----------
FUN     U1(S,"N",1)
NXT     1
```

```
     1      ----------
FUN      S
NXT      COND(P1K("M",1,1),2,3)

     2      ----------
FUN      U(S,NUL,U0("TF","T"))
NXT

     3      ----------
FUN      U1(S,"N",SEL("N",S) + 1)
NXT      4

     4      ----------
FUN      S
NXT      COND(LEQ(SEL("N",S),PLIM(1,LNX(SEL("STR",S)))),1,5)

     5      ----------
FUN      U0("TF","F")
NXT


90    ----------
     0      ----------
FUN      U1(S,"N",1)
NXT      1

     1      ----------
FUN      S
NXT      COND(P1K("A",1,1),2,3)

     2      ----------
FUN      U(S,NUL,U0("TF","T"))
NXT

     3      ----------
FUN      U1(S,"N",SEL("N",S) + 1)
NXT      4

     4      ----------
FUN      S
NXT      COND(LEQ(SEL("N",S),PLIM(1,LNX(SEL("STR",S)))),1,5)

     5      ----------
FUN      U0("TF","F")
NXT


91    ----------
     0      ----------
FUN      U1(S,"N",1)
NXT      1

     1      ----------
FUN      S
NXT      COND(P1K("N",1,1),2,3)

     2      ----------
FUN      U(S,NUL,U0("TF","T"))
NXT
```

```
      3      ----------
FUN       U1(S,"N",SEL("N",S) + 1)
NXT       4

      4      ----------
FUN       S
NXT       COND(LEQ(SEL("N",S),PLIM(1,LNX(SEL("STR",S)))),1,5)

      5      ----------
FUN       UO("TF","F")
NXT


92    ----------
      0      ----------
FUN       U1(S,"N",1)
NXT       1

      1      ----------
FUN       S
NXT       COND(P1K("K",1,1),2,3)

      2      ----------
FUN       U(S,NUL,UO("TF","T"))
NXT

      3      ----------
FUN       U1(S,"N",SEL("N",S) + 1)
NXT       4

      4      ----------
FUN       S
NXT       COND(LEQ(SEL("N",S),PLIM(1,LNX(SEL("STR",S)))),1,5)

      5      ----------
FUN       UO("TF","F")
NXT


93    ----------
      0      ----------
FUN       U1(S,"N",1)
NXT       1

      1      ----------
FUN       S
NXT       COND(P1K("F",1,1),2,3)

      2      ----------
FUN       U(S,NUL,UO("TF","T"))
NXT

      3      ----------
FUN       U1(S,"N",SEL("N",S) + 1)
NXT       4

      4      ----------
FUN       S
NXT       COND(LEQ(SEL("N",S),PLIM(1,LNX(SEL("STR",S)))),1,5)
```

```
     5      ----------
FUN         U0("TF","F")
NXT

94        ----------
     0      ----------
FUN         U1(S,"N",1)
NXT         1

     1      ----------
FUN         S
NXT         COND(P3K("W",1,1),2,3)

     2      ----------
FUN         U(S,U0("W",SUBSTR(PARTITION(1,2 * 1 - 1,SEL("N",S),LNX(SEL("STR",S))),PA
RTITION(1,2 * 1,SEL("N",S),LNX(SEL("STR",S))),SEL("STR",S))),U0("TF","T"))
NXT

     3      ----------
FUN         U1(S,"N",SEL("N",S) + 1)
NXT         4

     4      ----------
FUN         S
NXT         COND(LEQ(SEL("N",S),PLIM(1,LNX(SEL("STR",S)))),1,5)

     5      ----------
FUN         U0("TF","F")
NXT

96        ----------
     0      ----------
FUN         U1(S,"N",1)
NXT         1

     1      ----------
FUN         S
NXT         COND(P3K("L",1,1),2,3)

     2      ----------
FUN         U(S,U0("L",SUBSTR(PARTITION(1,2 * 1 - 1,SEL("N",S),LNX(SEL("STR",S))),PA
RTITION(1,2 * 1,SEL("N",S),LNX(SEL("STR",S))),SEL("STR",S))),U0("TF","T"))
NXT

     3      ----------
FUN         U1(S,"N",SEL("N",S) + 1)
NXT         4

     4      ----------
FUN         S
NXT         COND(LEQ(SEL("N",S),PLIM(1,LNX(SEL("STR",S)))),1,5)

     5      ----------
FUN         U0("TF","F")
NXT
```

```
98      ----------
     0      ----------
FUN       U1(S,"N",1)
NXT       1

     1      ----------
FUN       S
NXT       COND(AND(P3K("L",2,1),P3K("W",2,2)),2,3)

     2      ----------
FUN       U(S,U(UO("L",SUBSTR(PARTITION(2,2 * 1 - 1,SEL("N",S),LNX(SEL("STR",S))),
PARTITION(2,2 * 1,SEL("N",S),LNX(SEL("STR",S))),SEL("STR",S))),UO("W",SUBSTR(PAR
TITION(2,2 * 2 - 1,SEL("N",S),LNX(SEL("STR",S))),PARTITION(2,2 * 2,SEL("N",S),LN
X(SEL("STR",S))),SEL("STR",S)))),UO("TF","T"))
NXT

     3      ----------
FUN       U1(S,"N",SEL("N",S) + 1)
NXT       4

     4      ----------
FUN       S
NXT       COND(LEQ(SEL("N",S),PLIM(2,LNX(SEL("STR",S)))),1,5)

     5      ----------
FUN       UO("TF","F")
NXT


110     ----------
     0      ----------
FUN       U1(S,"N",1)
NXT       1

     1      ----------
FUN       S
NXT       COND(AND(P3K("V",4,1),AND(P1K(",",4,2),AND(P3K("W",4,3),P1K("*",4,4)))),
2,3)

     2      ----------
FUN       U(S,U(UO("V",SUBSTR(PARTITION(4,2 * 1 - 1,SEL("N",S),LNX(SEL("STR",S))),
PARTITION(4,2 * 1,SEL("N",S),LNX(SEL("STR",S))),SEL("STR",S))),U(NUL,U(UO("W",SU
BSTR(PARTITION(4,2 * 3 - 1,SEL("N",S),LNX(SEL("STR",S))),PARTITION(4,2 * 3,SEL("
N",S),LNX(SEL("STR",S))),SEL("STR",S))),NUL))),UO("TF","T"))
NXT

     3      ----------
FUN       U1(S,"N",SEL("N",S) + 1)
NXT       4

     4      ----------
FUN       S
NXT       COND(LEQ(SEL("N",S),PLIM(4,LNX(SEL("STR",S)))),1,5)

     5      ----------
FUN       UO("TF","F")
NXT


122     ----------
```

```
     0     ----------
FUN        U1(S,"N",1)
NXT        1

     1     ----------
FUN        S
NXT        COND(AND(P1K("(",3,1),AND(P3K("W",3,2),P1K("*,",3,3))),2,3)

     2     ----------
FUN        U(S,U(NUL,U(U0("W",SUBSTR(PARTITION(3,2 * 2 - 1,SEL("N",S),LNX(SEL("STR"
,S))),PARTITION(3,2 * 2,SEL("N",S),LNX(SEL("STR",S))),SEL("STR",S))),NUL)),U0("T
F","T"))
NXT

     3     ----------
FUN        U1(S,"N",SEL("N",S) + 1)
NXT        4

     4     ----------
FUN        S
NXT        COND(LEQ(SEL("N",S),PLIM(3,LNX(SEL("STR",S)))),1,5)

     5     ----------
FUN        U0("TF","F")
NXT


   132     ----------
     0     ----------
FUN        U1(S,"N",1)
NXT        1

     1     ----------
FUN        S
NXT        COND(AND(P1K("(",3,1),AND(P3K("W",3,2),P1K("*)",3,3))),2,3)

     2     ----------
FUN        U(S,U(NUL,U(U0("W",SUBSTR(PARTITION(3,2 * 2 - 1,SEL("N",S),LNX(SEL("STR"
,S))),PARTITION(3,2 * 2,SEL("N",S),LNX(SEL("STR",S))),SEL("STR",S))),NUL)),U0("T
F","T"))
NXT

     3     ----------
FUN        U1(S,"N",SEL("N",S) + 1)
NXT        4

     4     ----------
FUN        S
NXT        COND(LEQ(SEL("N",S),PLIM(3,LNX(SEL("STR",S)))),1,5)

     5     ----------
FUN        U0("TF","F")
NXT


   142     ----------
     0     ----------
FUN        U1(S,"N",1)
NXT        1
```

```
      1     ----------
FUN       S
NXT       COND(P1K("(",1,1),2,3)

      2     ----------
FUN       U(S,NUL,U0("TF","T"))
NXT

      3     ----------
FUN       U1(S,"N",SEL("N",S) + 1)
NXT       4

      4     ----------
FUN       S
NXT       COND(LEQ(SEL("N",S),PLIM(1,LNX(SEL("STR",S)))),1,5)

      5     ----------
FUN       U0("TF","F")
NXT


152       ----------
      0     ----------
FUN       U1(S,"N",1)
NXT       1

      1     ----------
FUN       S
NXT       COND(P1K(")",1,1),2,3)

      2     ----------
FUN       U(S,NUL,U0("TF","T"))
NXT

      3     ----------
FUN       U1(S,"N",SEL("N",S) + 1)
NXT       4

      4     ----------
FUN       S
NXT       COND(LEQ(SEL("N",S),PLIM(1,LNX(SEL("STR",S)))),1,5)

      5     ----------
FUN       U0("TF","F")
NXT
```

APPLY FLOWCHART TO NULL STATE VECTOR S

FINAL STATE VECTOR
MANN,KAFKA,HESSE

TIME TO PROCESS SEMANTIC DESCRIPTION 1834 MILLISEC
TIME TO PARSE 3916 MILLISEC
TIME TO EVALUATE SEMANTIC ATTRIBUTES 70196 MILLISEC
'EXECUTION' TIME 4359970 MILLISEC
'STATEMENTS' EXECUTED 249873

SOURCE PROGRAM

```
L=-I- / -J- / -K-;
P=L / -(-A-)-;
A=P / P-*-A / P-+-A;
X=L / -(- / -)- / -*- / -+-;
Y=X;
LET T1=<<P-*-A # A;
         P-+-A # A;
         -(-A-)- # A;
         XY # -*-;
         A # .-YES-;
         -*- # .-NO->>;
T1<-I*(J+K)+J->;
END
```

```
LISTING OF ATTRIBUTES OF PROGRAM NODE

          DISPLAY OF EXPRESSION-E
0     ----------
FUN     APPLI(SEL(1,TRANSLIB),"I*(J+K)+1")
NXT


          DISPLAY OF VARIABLETABLE-V
L     ----------
    0     ----------
FUN     S
NXT     COND(EQU(SEL("L" "LNX",S), LNX(SEL("STR",S))),1,2)

    1     ----------
FUN     "F"
NXT

    2     ----------
FUN     U1(S,"L" "LNX", LNX(SEL("STR",S)))
NXT     3

    3     ----------
FUN     OR(SEL("TF",APPLI(SEL(213,CONCATLIB),S)),OR(SEL("TF",APPLI(SEL(214,CONCA
TLIB),S)),SEL("TF",APPLI(SEL(215,CONCATLIB),S))))
NXT


P     ----------
    0     ----------
FUN     S
NXT     COND(EQU(SEL("P" "LNX",S), LNX(SEL("STR",S))),1,2)

    1     ----------
FUN     "F"
NXT

    2     ----------
FUN     U1(S,"P" "LNX", LNX(SEL("STR",S)))
NXT     3

    3     ----------
FUN     OR(SEL("TF",APPLI(SEL(216,CONCATLIB),S)),SEL("TF",APPLI(SEL(218,CONCATLI
B),S)))
NXT


A     ----------
    0     ----------
FUN     S
NXT     COND(EQU(SEL("A" "LNX",S), LNX(SEL("STR",S))),1,2)

    1     ----------
FUN     "F"
NXT

    2     ----------
```

```
FUN     U1(S,"A" "LNX", LNX(SEL("STR",S)))
NXT     3

    3   ----------
FUN     OR(SEL("TF",APPLI(SEL(220,CONCATLIB),S)),OR(SEL("TF",APPLI(SEL(222,CONCA
TLIB),S)),SEL("TF",APPLI(SEL(226,CONCATLIB),S))))
NXT


X     ----------
    0   ----------
FUN     S
NXT     COND(EQU(SEL("X" "LNX",S), LNX(SEL("STR",S))),1,2)

    1   ----------
FUN     "F"
NXT

    2   ----------
FUN     U1(S,"X" "LNX", LNX(SEL("STR",S)))
NXT     3

    3   ----------
FUN     OR(SEL("TF",APPLI(SEL(230,CONCATLIB),S)),OR(SEL("TF",APPLI(SEL(232,CONCA
TLIB),S)),OR(SEL("TF",APPLI(SEL(233,CONCATLIB),S)),OR(SEL("TF",APPLI(SEL(234,CON
CATLIB),S)),SEL("TF",APPLI(SEL(235,CONCATLIB),S))))))
NXT


Y     ----------
    0   ----------
FUN     S
NXT     COND(EQU(SEL("Y" "LNX",S), LNX(SEL("STR",S))),1,2)

    1   ----------
FUN     "F"
NXT

    2   ----------
FUN     U1(S,"Y" "LNX", LNX(SEL("STR",S)))
NXT     3

    3   ----------
FUN     SEL("TF",APPLI(SEL(236,CONCATLIB),S))
NXT



          DISPLAY OF TRANSTABLE-T
1     ----------
    0   ----------
FUN     S
NXT     238

  238   ----------
FUN     U(UO("P1",1),UO("P2",1),UO("STR",S))
NXT     239

  239   ----------
```

```
FUN     S
NXT     COND(APPLI('SEL("TF",APPLI(SEL(246,CONCATLIB),S))',U1(S,"STR",SUBSTR(SEL
("P1",S),SEL("P2",S),SEL("STR",S)))),240,241)

    240     ----------
FUN     SUBSTR(1,SEL("P1",S)    ,SEL("STR",S)) APPLI('SEL("A",S)',APPLI("APPLI(S
EL(246,CONCATLIB),S)",U1(S,"STR",SUBSTR(SEL("P1",S),SEL("P2",S),SEL("STR",S)))))
 SUBSTR(SEL("P2",S),LNX(SEL("STR",S)) + 1,SEL("STR",S))
NXT     0

    241     ----------
FUN     U1(S,"P2",SEL("P2",S) + 1)
NXT     COND(LEQ(SEL("P2",S) - 1,LNX(SEL("STR",S))),239,242)

    242     ----------
FUN     U(S,U0("P1",SEL("P1",S) + 1),U0("P2",SEL("P1",S) + 2))
NXT     COND(LEQ(SEL("P1",S),LNX(SEL("STR",S))),239,243)

    243     ----------
FUN     SEL("STR",S)
NXT     250

    250     ----------
FUN     U(U0("P1",1),U0("P2",1),U0("STR",S))
NXT     251

    251     ----------
FUN     S
NXT     COND(APPLI('SEL("TF",APPLI(SEL(258,CONCATLIB),S))',U1(S,"STR",SUBSTR(SEL
("P1",S),SEL("P2",S),SEL("STR",S)))),252,253)

    252     ----------
FUN     SUBSTR(1,SEL("P1",S)    ,SEL("STR",S)) APPLI('SEL("A",S)',APPLI("APPLI(S
EL(258,CONCATLIB),S)",U1(S,"STR",SUBSTR(SEL("P1",S),SEL("P2",S),SEL("STR",S)))))
 SUBSTR(SEL("P2",S),LNX(SEL("STR",S)) + 1,SEL("STR",S))
NXT     0

    253     ----------
FUN     U1(S,"P2",SEL("P2",S) + 1)
NXT     COND(LEQ(SEL("P2",S) - 1,LNX(SEL("STR",S))),251,254)

    254     ----------
FUN     U(S,U0("P1",SEL("P1",S) + 1),U0("P2",SEL("P1",S) + 2))
NXT     COND(LEQ(SEL("P1",S),LNX(SEL("STR",S))),251,255)

    255     ----------
FUN     SEL("STR",S)
NXT     262

    262     ----------
FUN     U(U0("P1",1),U0("P2",1),U0("STR",S))
NXT     263

    263     ----------
FUN     S
NXT     COND(APPLI('SEL("TF",APPLI(SEL(270,CONCATLIB),S))',U1(S,"STR",SUBSTR(SEL
("P1",S),SEL("P2",S),SEL("STR",S)))),264,265)

    264     ----------
FUN     SUBSTR(1,SEL("P1",S)    ,SEL("STR",S)) APPLI('SEL("A",S)',APPLI("APPLI(S
```

```
EL(270,CONCATLIB),S)",U1(S,"STR",SUBSTR(SEL("P1",S),SEL("P2",S),SEL("STR",S)))))
 SUBSTR(SEL("P2",S),LNX(SEL("STR",S)) + 1,SEL("STR",S))
NXT      0

     265      ----------
FUN      U1(S,"P2",SEL("P2",S) + 1)
NXT      COND(LEQ(SEL("P2",S) - 1,LNX(SEL("STR",S))),263,266)

     266      ----------
FUN      U(S,U0("P1",SEL("P1",S) + 1),U0("P2",SEL("P1",S) + 2))
NXT      COND(LEQ(SEL("P1",S),LNX(SEL("STR",S))),263,267)

     267      ----------
FUN      SEL("STR",S)
NXT      272

     272      ----------
FUN      U(U0("P1",1),U0("P2",1),U0("STR",S))
NXT      273

     273      ----------
FUN      S
NXT      COND(APPLI('SEL("TF",APPLI(SEL(280,CONCATLIB),S))',U1(S,"STR",SUBSTR(SEL
("P1",S),SEL("P2",S),SEL("STR",S)))),274,275)

     274      ----------
FUN      SUBSTR(1,SEL("P1",S)     ,SEL("STR",S)) APPLI('"*"',APPLI("APPLI(SEL(280,
CONCATLIB),S)",U1(S,"STR",SUBSTR(SEL("P1",S),SEL("P2",S),SEL("STR",S))))) SUBSTR
(SEL("P2",S),LNX(SEL("STR",S)) + 1,SEL("STR",S))
NXT      0

     275      ----------
FUN      U1(S,"P2",SEL("P2",S) + 1)
NXT      COND(LEQ(SEL("P2",S) - 1,LNX(SEL("STR",S))),273,276)

     276      ----------
FUN      U(S,U0("P1",SEL("P1",S) + 1),U0("P2",SEL("P1",S) + 2))
NXT      COND(LEQ(SEL("P1",S),LNX(SEL("STR",S))),273,277)

     277      ----------
FUN      SEL("STR",S)
NXT      284

     284      ----------
FUN      U(U0("P1",1),U0("P2",1),U0("STR",S))
NXT      285

     285      ----------
FUN      S
NXT      COND(APPLI('SEL("TF",APPLI(SEL(292,CONCATLIB),S))',U1(S,"STR",SUBSTR(SEL
("P1",S),SEL("P2",S),SEL("STR",S)))),286,287)

     286      ----------
FUN      SUBSTR(1,SEL("P1",S)     ,SEL("STR",S)) APPLI('"YES"',APPLI("APPLI(SEL(29
2,CONCATLIB),S)",U1(S,"STR",SUBSTR(SEL("P1",S),SEL("P2",S),SEL("STR",S))))) SUBS
TR(SEL("P2",S),LNX(SEL("STR",S)) + 1,SEL("STR",S))
NXT

     287      ----------
FUN      U1(S,"P2",SEL("P2",S) + 1)
```

```
NXT     COND(LEQ(SEL("P2",S) - 1,LNX(SEL("STR",S))),285,288)

    288     ----------
FUN     U(S,U0("P1",SEL("P1",S) + 1),U0("P2",SEL("P1",S) + 2))
NXT     COND(LEQ(SEL("P1",S),LNX(SEL("STR",S))),285,289)

    289     ----------
FUN     SEL("STR",S)
NXT     294

    294     ----------
FUN     U(U0("P1",1),U0("P2",1),U0("STR",S))
NXT     295

    295     ----------
FUN     S
NXT     COND(APPLI('SEL("TF",APPLI(SEL(303,CONCATLIB),S))',U1(S,"STR",SUBSTR(SEL
("P1",S),SEL("P2",S),SEL("STR",S)))),296,297)

    296     ----------
FUN     SUBSTR(1,SEL("P1",S)     ,SEL("STR",S)) APPLI('"NO"',APPLI("APPLI(SEL(303
,CONCATLIB),S)",U1(S,"STR",SUBSTR(SEL("P1",S),SEL("P2",S),SEL("STR",S))))) SUBST
R(SEL("P2",S),LNX(SEL("STR",S)) + 1,SEL("STR",S))
NXT

    297     ----------
FUN     U1(S,"P2",SEL("P2",S) + 1)
NXT     COND(LEQ(SEL("P2",S) - 1,LNX(SEL("STR",S))),295,298)

    298     ----------
FUN     U(S,U0("P1",SEL("P1",S) + 1),U0("P2",SEL("P1",S) + 2))
NXT     COND(LEQ(SEL("P1",S),LNX(SEL("STR",S))),295,299)

    299     ----------
FUN     SEL("STR",S)
NXT



          DISPLAY OF CONCATLIB-C
213     ----------
    0     ----------
FUN     U1(S,"N",1)
NXT     1

    1     ----------
FUN     S
NXT     COND(P1K("I",1,1),2,3)

    2     ----------
FUN     U(S,NUL,U0("TF","T"))
NXT

    3     ----------
FUN     U1(S,"N",SEL("N",S) + 1)
NXT     4

    4     ----------
FUN     S
```

```
NXT      COND(LEQ(SEL("N",S),PLIM(1,LNX(SEL("STR",S)))),1,5)

     5   ----------
FUN      UO("TF","F")
NXT


214      ----------
     0   ----------
FUN      U1(S,"N",1)
NXT      1

     1   ----------
FUN      S
NXT      COND(P1K("J",1,1),2,3)

     2   ----------
FUN      U(S,NUL,UO("TF","T"))
NXT

     3   ----------
FUN      U1(S,"N",SEL("N",S) + 1)
NXT      4

     4   ----------
FUN      S
NXT      COND(LEQ(SEL("N",S),PLIM(1,LNX(SEL("STR",S)))),1,5)

     5   ----------
FUN      UO("TF","F")
NXT


215      ----------
     0   ----------
FUN      U1(S,"N",1)
NXT      1

     1   ----------
FUN      S
NXT      COND(P1K("K",1,1),2,3)

     2   ----------
FUN      U(S,NUL,UO("TF","T"))
NXT

     3   ----------
FUN      U1(S,"N",SEL("N",S) + 1)
NXT      4

     4   ----------
FUN      S
NXT      COND(LEQ(SEL("N",S),PLIM(1,LNX(SEL("STR",S)))),1,5)

     5   ----------
FUN      UO("TF","F")
NXT


216      ----------
```

```
     0      ----------
FUN      U1(S,"N",1)
NXT      1

     1      ----------
FUN      S
NXT      COND(P3K("L",1,1),2,3)

     2      ----------
FUN      U(S,UO("L",SUBSTR(PARTITION(1,2 * 1 - 1,SEL("N",S),LNX(SEL("STR",S))),PA
RTITION(1,2 * 1,SEL("N",S),LNX(SEL("STR",S))),SEL("STR",S))),UO("TF","T"))
NXT

     3      ----------
FUN      U1(S,"N",SEL("N",S) + 1)
NXT      4

     4      ----------
FUN      S
NXT      COND(LEQ(SEL("N",S),PLIM(1,LNX(SEL("STR",S)))),1,5)

     5      ----------
FUN      UO("TF","F")
NXT


218      ----------
     0      ----------
FUN      U1(S,"N",1)
NXT      1

     1      ----------
FUN      S
NXT      COND(AND(P1K("(",3,1),AND(P3K("A",3,2),P1K(")",3,3))),2,3)

     2      ----------
FUN      U(S,U(NUL,U(UO("A",SUBSTR(PARTITION(3,2 * 2 - 1,SEL("N",S),LNX(SEL("STR"
,S))),PARTITION(3,2 * 2,SEL("N",S),LNX(SEL("STR",S))),SEL("STR",S))),NUL)),UO("T
F","T"))
NXT

     3      ----------
FUN      U1(S,"N",SEL("N",S) + 1)
NXT      4

     4      ----------
FUN      S
NXT      COND(LEQ(SEL("N",S),PLIM(3,LNX(SEL("STR",S)))),1,5)

     5      ----------
FUN      UO("TF","F")
NXT


220      ----------
     0      ----------
FUN      U1(S,"N",1)
NXT      1

     1      ----------
```

```
FUN     S
NXT     COND(P3K("P",1,1),2,3)

    2   ----------
FUN     U(S,UO("P",SUBSTR(PARTITION(1,2 * 1 - 1,SEL("N",S),LNX(SEL("STR",S))),PA
RTITION(1,2 * 1,SEL("N",S),LNX(SEL("STR",S))),SEL("STR",S))),UO("TF","T"))
NXT

    3   ----------
FUN     U1(S,"N",SEL("N",S) + 1)
NXT     4

    4   ----------
FUN     S
NXT     COND(LEQ(SEL("N",S),PLIM(1,LNX(SEL("STR",S)))),1,5)

    5   ----------
FUN     UO("TF","F")
NXT


222     ----------
    0   ----------
FUN     U1(S,"N",1)
NXT     1

    1   ----------
FUN     S
NXT     COND(AND(P3K("P",3,1),AND(P1K("*",3,2),P3K("A",3,3))),2,3)

    2   ----------
FUN     U(S,U(UO("P",SUBSTR(PARTITION(3,2 * 1 - 1,SEL("N",S),LNX(SEL("STR",S))),
PARTITION(3,2 * 1,SEL("N",S),LNX(SEL("STR",S))),SEL("STR",S))),U(NUL,UO("A",SUBS
TR(PARTITION(3,2 * 3 - 1,SEL("N",S),LNX(SEL("STR",S))),PARTITION(3,2 * 3,SEL("N"
,S),LNX(SEL("STR",S))),SEL("STR",S)))),UO("TF","T"))
NXT

    3   ----------
FUN     U1(S,"N",SEL("N",S) + 1)
NXT     4

    4   ----------
FUN     S
NXT     COND(LEQ(SEL("N",S),PLIM(3,LNX(SEL("STR",S)))),1,5)

    5   ----------
FUN     UO("TF","F")
NXT


226     ----------
    0   ----------
FUN     U1(S,"N",1)
NXT     1

    1   ----------
FUN     S
NXT     COND(AND(P3K("P",3,1),AND(P1K("+",3,2),P3K("A",3,3))),2,3)

    2   ----------
```

```
FUN     U(S,U(UO("P",SUBSTR(PARTITION(3,2 * 1 - 1,SEL("N",S),LNX(SEL("STR",S))),
PARTITION(3,2 * 1,SEL("N",S),LNX(SEL("STR",S))),SEL("STR",S))),U(NUL,UO("A",SUBS
TR(PARTITION(3,2 * 3 - 1,SEL("N",S),LNX(SEL("STR",S))),PARTITION(3,2 * 3,SEL("N"
,S),LNX(SEL("STR",S))),SEL("STR",S)))),UO("TF","T"))
NXT

     3    ----------
FUN      U1(S,"N",SEL("N",S) + 1)
NXT      4

     4    ----------
FUN      S
NXT      COND(LEQ(SEL("N",S),PLIM(3,LNX(SEL("STR",S)))),1,5)

     5    ----------
FUN      UO("TF","F")
NXT


230      ----------
     0    ----------
FUN      U1(S,"N",1)
NXT      1

     1    ----------
FUN      S
NXT      COND(P3K("L",1,1),2,3)

     2    ----------
FUN      U(S,UO("L",SUBSTR(PARTITION(1,2 * 1 - 1,SEL("N",S),LNX(SEL("STR",S))),PA
RTITION(1,2 * 1,SEL("N",S),LNX(SEL("STR",S))),SEL("STR",S))),UO("TF","T"))
NXT

     3    ----------
FUN      U1(S,"N",SEL("N",S) + 1)
NXT      4

     4    ----------
FUN      S
NXT      COND(LEQ(SEL("N",S),PLIM(1,LNX(SEL("STR",S)))),1,5)

     5    ----------
FUN      UO("TF","F")
NXT


232      ----------
     0    ----------
FUN      U1(S,"N",1)
NXT      1

     1    ----------
FUN      S
NXT      COND(P1K("(",1,1),2,3)

     2    ----------
FUN      U(S,NUL,UO("TF","T"))
NXT

     3    ----------
```

```
FUN     U1(S,"N",SEL("N",S) + 1)
NXT     4

    4   ----------
FUN     S
NXT     COND(LEQ(SEL("N",S),PLIM(1,LNX(SEL("STR",S)))),1,5)

    5   ----------
FUN     U0("TF","F")
NXT


233     ----------
    0   ----------
FUN     U1(S,"N",1)
NXT     1

    1   ----------
FUN     S
NXT     COND(P1K(")",1,1),2,3)

    2   ----------
FUN     U(S,NUL,U0("TF","T"))
NXT

    3   ----------
FUN     U1(S,"N",SEL("N",S) + 1)
NXT     4

    4   ----------
FUN     S
NXT     COND(LEQ(SEL("N",S),PLIM(1,LNX(SEL("STR",S)))),1,5)

    5   ----------
FUN     U0("TF","F")
NXT


234     ----------
    0   ----------
FUN     U1(S,"N",1)
NXT     1

    1   ----------
FUN     S
NXT     COND(P1K("*",1,1),2,3)

    2   ----------
FUN     U(S,NUL,U0("TF","T"))
NXT

    3   ----------
FUN     U1(S,"N",SEL("N",S) + 1)
NXT     4

    4   ----------
FUN     S
NXT     COND(LEQ(SEL("N",S),PLIM(1,LNX(SEL("STR",S)))),1,5)

    5   ----------
```

```
FUN     U0("TF","F")
NXT

235     ----------
     0      ----------
FUN     U1(S,"N",1)
NXT     1

     1      ----------
FUN     S
NXT     COND(P1K("+",1,1),2,3)

     2      ----------
FUN     U(S,NUL,U0("TF","T"))
NXT

     3      ----------
FUN     U1(S,"N",SEL("N",S) + 1)
NXT     4

     4      ----------
FUN     S
NXT     COND(LEQ(SEL("N",S),PLIM(1,LNX(SEL("STR",S)))),1,5)

     5      ----------
FUN     U0("TF","F")
NXT

236     ----------
     0      ----------
FUN     U1(S,"N",1)
NXT     1

     1      ----------
FUN     S
NXT     COND(P3K("X",1,1),2,3)

     2      ----------
FUN     U(S,U0("X",SUBSTR(PARTITION(1,2 * 1 - 1,SEL("N",S),LNX(SEL("STR",S))),PA
RTITION(1,2 * 1,SEL("N",S),LNX(SEL("STR",S))),SEL("STR",S))),U0("TF","T"))
NXT

     3      ----------
FUN     U1(S,"N",SEL("N",S) + 1)
NXT     4

     4      ----------
FUN     S
NXT     COND(LEQ(SEL("N",S),PLIM(1,LNX(SEL("STR",S)))),1,5)

     5      ----------
FUN     U0("TF","F")
NXT

246     ----------
     0      ----------
FUN     U1(S,"N",1)
```

```
NXT     1

     1     ----------
FUN     S
NXT     COND(AND(P3K("P",3,1),AND(P1K("*",3,2),P3K("A",3,3))),2,3)

     2     ----------
FUN     U(S,U(UO("P",SUBSTR(PARTITION(3,2 * 1 - 1,SEL("N",S),LNX(SEL("STR",S))),
PARTITION(3,2 * 1,SEL("N",S),LNX(SEL("STR",S))),SEL("STR",S))),U(NUL,UO("A",SUBS
TR(PARTITION(3,2 * 3 - 1,SEL("N",S),LNX(SEL("STR",S))),PARTITION(3,2 * 3,SEL("N"
,S),LNX(SEL("STR",S))),SEL("STR",S)))),UO("TF","T"))
NXT

     3     ----------
FUN     U1(S,"N",SEL("N",S) + 1)
NXT     4

     4     ----------
FUN     S
NXT     COND(LEQ(SEL("N",S),PLIM(3,LNX(SEL("STR",S)))),1,5)

     5     ----------
FUN     UO("TF","F")
NXT


258     ----------
     0     ----------
FUN     U1(S,"N",1)
NXT     1

     1     ----------
FUN     S
NXT     COND(AND(P3K("P",3,1),AND(P1K("+",3,2),P3K("A",3,3))),2,3)

     2     ----------
FUN     U(S,U(UO("P",SUBSTR(PARTITION(3,2 * 1 - 1,SEL("N",S),LNX(SEL("STR",S))),
PARTITION(3,2 * 1,SEL("N",S),LNX(SEL("STR",S))),SEL("STR",S))),U(NUL,UO("A",SUBS
TR(PARTITION(3,2 * 3 - 1,SEL("N",S),LNX(SEL("STR",S))),PARTITION(3,2 * 3,SEL("N"
,S),LNX(SEL("STR",S))),SEL("STR",S)))),UO("TF","T"))
NXT

     3     ----------
FUN     U1(S,"N",SEL("N",S) + 1)
NXT     4

     4     ----------
FUN     S
NXT     COND(LEQ(SEL("N",S),PLIM(3,LNX(SEL("STR",S)))),1,5)

     5     ----------
FUN     UO("TF","F")
NXT


270     ----------
     0     ----------
FUN     U1(S,"N",1)
NXT     1
```

```
     1     ----------
FUN        S
NXT        COND(AND(P1K("(",3,1),AND(P3K("A",3,2),P1K(")",3,3))),2,3)

     2     ----------
FUN        U(S,U(NUL,U(UO("A",SUBSTR(PARTITION(3,2 * 2 - 1,SEL("N",S),LNX(SEL("STR"
,S))),PARTITION(3,2 * 2,SEL("N",S),LNX(SEL("STR",S))),SEL("STR",S))),NUL)),UO("T
F","T"))
NXT

     3     ----------
FUN        U1(S,"N",SEL("N",S) + 1)
NXT        4

     4     ----------
FUN        S
NXT        COND(LEQ(SEL("N",S),PLIM(3,LNX(SEL("STR",S)))),1,5)

     5     ----------
FUN        UO("TF","F")
NXT


280  ----------
     0     ----------
FUN        U1(S,"N",1)
NXT        1

     1     ----------
FUN        S
NXT        COND(AND(P3K("X",2,1),P3K("Y",2,2)),2,3)

     2     ----------
FUN        U(S,U(UO("X",SUBSTR(PARTITION(2,2 * 1 - 1,SEL("N",S),LNX(SEL("STR",S))),
PARTITION(2,2 * 1,SEL("N",S),LNX(SEL("STR",S))),SEL("STR",S))),UO("Y",SUBSTR(PAR
TITION(2,2 * 2 - 1,SEL("N",S),LNX(SEL("STR",S))),PARTITION(2,2 * 2,SEL("N",S),LN
X(SEL("STR",S))),SEL("STR",S)))),UO("TF","T"))
NXT

     3     ----------
FUN        U1(S,"N",SEL("N",S) + 1)
NXT        4

     4     ----------
FUN        S
NXT        COND(LEQ(SEL("N",S),PLIM(2,LNX(SEL("STR",S)))),1,5)

     5     ----------
FUN        UO("TF","F")
NXT


292  ----------
     0     ----------
FUN        U1(S,"N",1)
NXT        1

     1     ----------
FUN        S
NXT        COND(P3K("A",1,1),2,3)
```

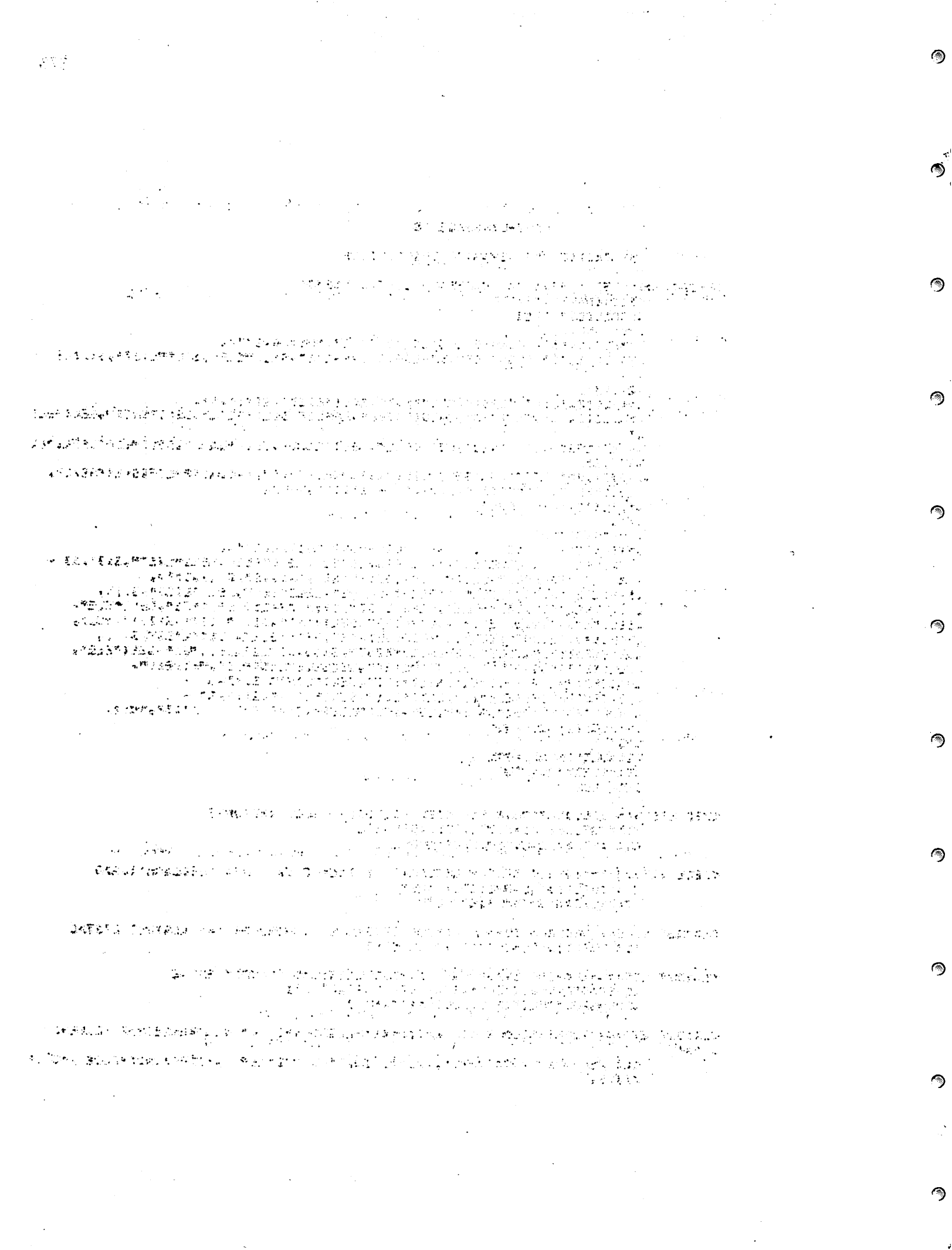
```
     2     ----------
FUN       U(S,UO("A",SUBSTR(PARTITION(1,2 * 1 - 1,SEL("N",S),LNX(SEL("STR",S))),PA
RTITION(1,2 * 1,SEL("N",S),LNX(SEL("STR",S))),SEL("STR",S))),UO("TF","T"))
NXT

     3     ----------
FUN       U1(S,"N",SEL("N",S) + 1)
NXT       4

     4     ----------
FUN       S
NXT       COND(LEQ(SEL("N",S),PLIM(1,LNX(SEL("STR",S)))),1,5)

     5     ----------
FUN       UO("TF","F")
NXT


303       ----------
     0     ----------
FUN       U1(S,"N",1)
NXT       1

     1     ----------
FUN       S
NXT       COND(PIK("*",1,1),2,3)

     2     ----------
FUN       U(S,NUL,UO("TF","T"))
NXT

     3     ----------
FUN       U1(S,"N",SEL("N",S) + 1)
NXT       4

     4     ----------
FUN       S
NXT       COND(LEQ(SEL("N",S),PLIM(1,LNX(SEL("STR",S)))),1,5)

     5     ----------
FUN       UO("TF","F")
NXT
```

```
APPLY FLOWCHART TO NULL STATE VECTOR S


FINAL STATE VECTOR
YES

TIME TO PROCESS SEMANTIC DESCRIPTION 1834 MILLISEC
TIME TO PARSE 4917 MILLISEC
TIME TO EVALUATE SEMANTIC ATTRIBUTES 98227 MILLISEC
'EXECUTION' TIME 1058372 MILLISEC
'STATEMENTS' EXECUTED 53819
```

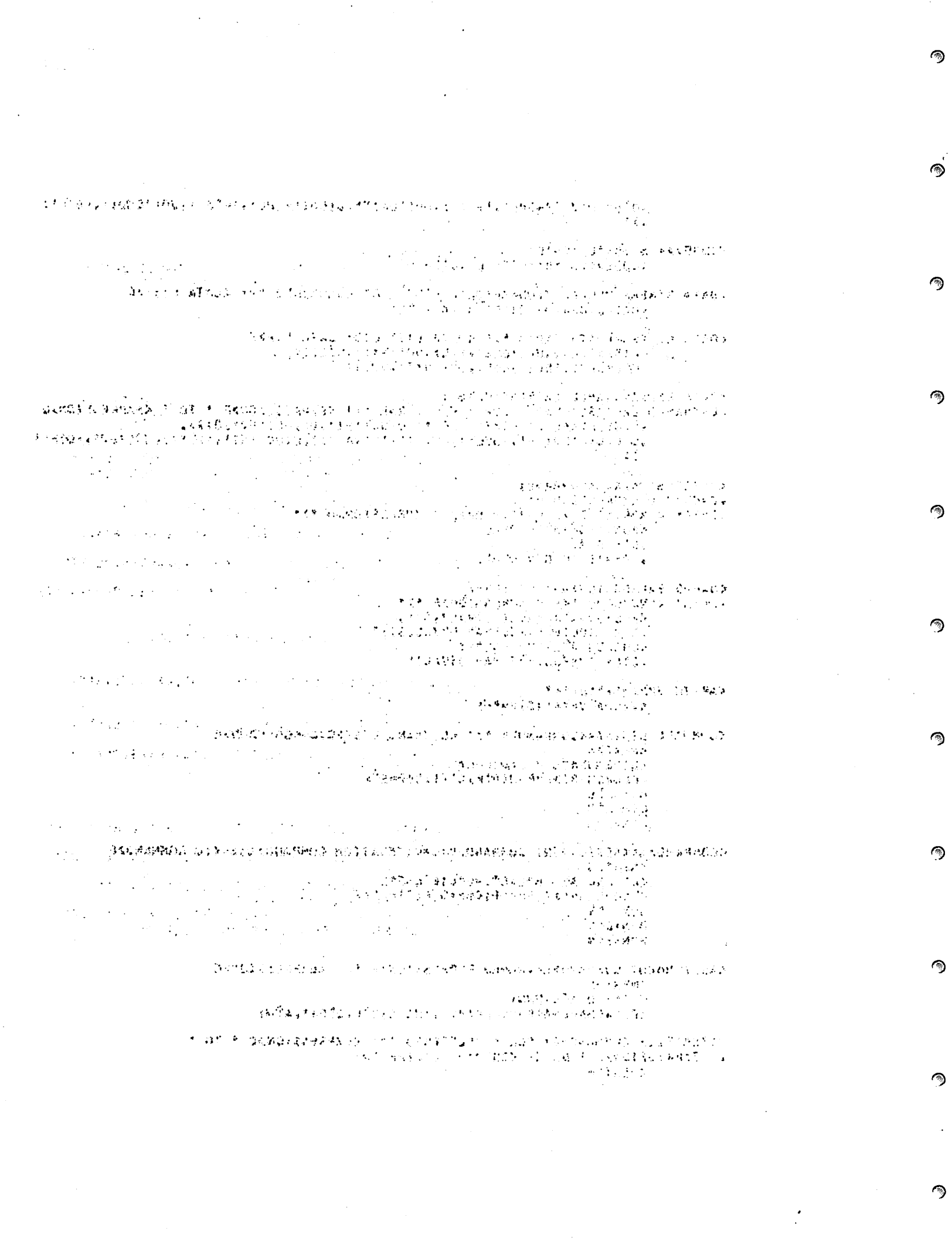
MINI-LANGUAGE 10

SYNTACTIC AND SEMANTIC DESCRIPTION

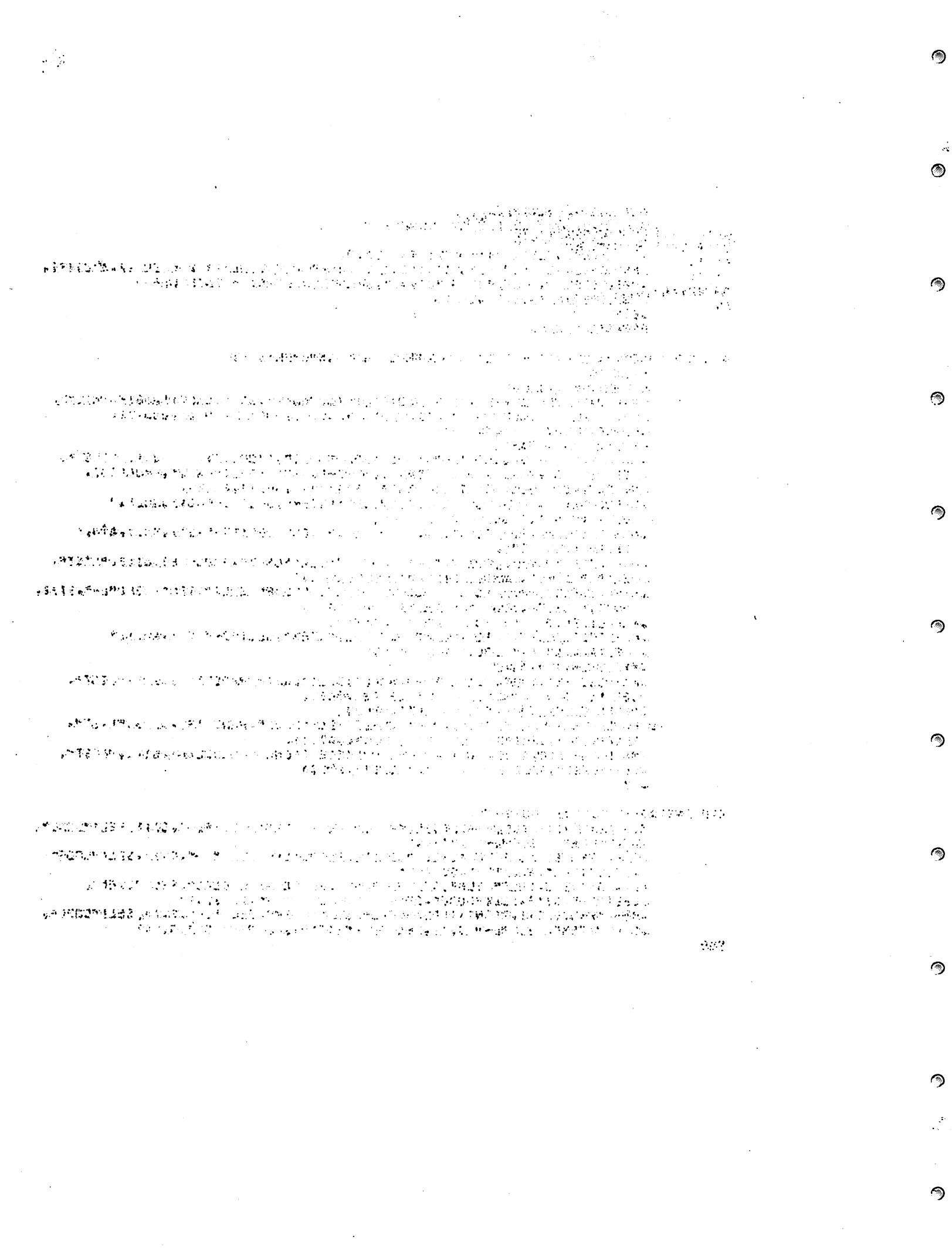
```
<PROGRAM>A=<DEC LIST>B ';' <COMMAND LIST>C ';END'
        <FLOWCHART>A↑F=C↑F
        <PROCLIB>A↑P=U(
        .UO('GNLI',U(
        .MKFLC(0,'S','COND(EQU(SEL("LIST",S),NUL),4,1)'),
        .MKFLC(1,'S','COND(LTN(APPLIF(SEL("REPL",SEL("ELE",SEL("LIST",S))),S),1
        .),
        .2,3)'),
        .MKFLC(2,'U1(S,"LIST",SEL("NXT",SEL("LIST",S)))',0),
        .MKFLC(3,'U(UO("ITEM",U(UO("REPL",APPLIF(SEL("REPL",SEL("UNIT",SEL("ELE
        .",
        .SEL("LIST",S)))),S)),UO("CODE",SEL("CODE",SEL("UNIT",SEL("ELE",SEL("LI
        .ST",S)
        .))))),UO("LIST",U1(SEL("LIST",S),"REPL?ELE",MKCON(APPLIF(SEL("REPL",
        .SEL("ELE",SEL("LIST",S))),S) - 1)))',NUL),
        .MKFLC(4,'NUL',NUL)
        .)),
        .UO('GNDI',U(
        .MKFLC(0,'S','COND(EQU(SEL("LIST",S),NUL),4,1)'),
        .MKFLC(1,'S','COND(EQU(APPLIF(SEL("E2",SEL("ELE",SEL("LIST",S))),S) -
        . APPLIF(SEL("E1",SEL("ELE",SEL("LIST",S))),S),0),2,3)'),
        .MKFLC(2,'U(UO("ITEM",APPLIF(SEL("B",SEL("ELE",SEL("LIST",S))),
        .U1(S,SEL("I",SEL("ELE",SEL("LIST",S))),APPLIF(SEL("E1",SEL("ELE",
        .SEL("LIST",S))),S)))),UO("LIST",SEL("NXT",SEL("LIST",S))))',NUL),
        .MKFLC(3,'U(UO("ITEM",APPLIF(SEL("B",SEL("ELE",SEL("LIST",S))),
        .U1(S,SEL("I",SEL("ELE",SEL("LIST",S))),APPLIF(SEL("E1",SEL("ELE",
        .SEL("LIST",S))),S)))),UO("LIST",U1(SEL("LIST",S),"E1?ELE",
        .MKCON(APPLIF(SEL("E1",SEL("ELE",SEL("LIST",S))),S) +
        . INCR(APPLIF(SEL("E2",SEL("ELE",SEL("LIST",S))),S) -
        . APPLIF(SEL("E1",SEL("ELE",SEL("LIST",S))),S))    ))))',NUL),
        .MKFLC(4,'NUL',NUL)
        .)))
        <LAYOUTLIB>A↑L=B↑L
        <DATALIB>A↑D=B↑D
        C↑N='NUL'

<DEC LIST>A=<DECLARATION>B ';' <DEC LIST>C;D=<DECLARATION>E
        <LAYOUTLIB>A↑L=U(B↑L,C↑L);D↑L=E↑L
        <DATALIB>A↑D=U(B↑D,C↑D);D↑D=E↑D

<DECLARATION>A=<LAYOUT STREAM DECLARATION>B;C=<DATA STREAM DECLARATION>D
        <LAYOUTLIB>A↑L=B↑L;C↑L='NUL'
        <DATALIB>A↑D='NUL';C↑D=D↑D

<LAYOUT STREAM DECLARATION>A='LAYOUT STREAM L' <$NUMBR>B '=' <LAYOUT LIST>C
        <LAYOUTLIB>A↑L=UO('L' B↑VAL,C↑L)

<LAYOUT LIST>A=<LAYOUT EXP>B ',' <LAYOUT LIST>C;D=<LAYOUT EXP>E
        <LIST>A↑L=U(UO('ELE',B↑L),UO('NXT',C↑L));
        .D↑L=U(UO('ELE',E↑L),UO('NXT',NUL))

<LAYOUT EXP>A=<EXPRESSION>B <CODE>C;D=<EXPRESSION>E '<' <EXPRESSION>F <CODE>G
. '>'
        <LISTEL>A↑L=U(UO('REPL',1),UO('UNIT',U(UO('REPL','B↑F'),UO('CODE','C↑C'
        .))));
```



```
        .D↑L= U(JO('REPL','E↑F'),UO('UNIT',U(UO('REPL','F↑F'),UO('CODE','G↑C'))
        .))

<CODE>A='B';B='L';C='D'
        <CODE>A↑C='B';B↑C='L';C↑C='D'

<DATA STREAM DECLARATION>A='DATA STREAM D' <$NUMBR>B '=' <DATA LIST>C
        <DATALIB>A↑D=UO('D' B↑VAL,C↑L)

<DATA LIST>A=<DATA EXP>B ',' <DATA LIST>C;D=<DATA EXP>E
        <LIST>A L=U(UO('ELE',B↑L),UO('NXT',C↑L));
        .D↑L=U(UO('ELE',E↑L),UO('NXT',NUL))

<DATA EXP>A=<NAMED EXPRESSION>B ;
.C=<NAMED EXPRESSION>D ' FOR ' <$LETTR>E '=' <EXPRESSION>F ' TO ' <EXPRESSION>G
        <LISTEL>A↑L=U(UO('B','B↑L'),U(UO('E1',0),UO('E2',0)));
        .C↑L=U(U(UO('B','D↑L'),UO('I','E↑VAL')),U(UO('E1','F↑F'),UO('E2','G↑F'))
        .))

<EXPRESSION>A=<$NUMBR>B;
.C=<NAMED EXPRESSION>D;
.E='(' <EXPRESSION>F <ARITH OP>G <EXPRESSION>H ')'
        <FUNCTION>A↑F=B↑VAL;
        .C↑F=D↑R;
        .E↑F='(F↑F G↑V H↑F)'

<NAMED EXPRESSION>A=<$LETTR>B;
.C='A' <$NUMBR>D '<' <EXPRESSION>E '>'
        <RHSFUN>A↑R='SEL("B↑VAL",S)';
        .C R='SEL(E↑F,SEL("A" D↑VAL,S))'
        <LHSFUN>A↑L='"B↑VAL"';
        .C↑L='COMPOS(E↑F,"A" D↑VAL)'

<ARITH OP>A='+';B='-'
        <VAL>A↑V='+';B↑V='-'

<COMMAND LIST>A=<COMMAND>B ';' <COMMAND LIST>C;D=<COMMAND>E
        <NEXT>N
        <LINENO>A↑L=B↑L;D↑L=E↑L
        <FLOWCHART>A↑F=U(B↑F,C↑F);D↑F=E↑F
        B↑N=C↑L
        C↑N=A↑N
        E↑N=D↑N

<COMMAND>A=<ASSIGNMENT COMMAND>B;C=<ITERATION COMMAND>D;E=<IO COMMAND>F
        <NEXT>N
        <LINENO>A↑L=B↑L;C↑L=D↑L;E↑L=F↑L
        <FLOWCHART>A↑F=B↑F;C↑F=D↑F;E↑F=F↑F
        B↑N=A↑N
        D↑N=C↑N
        F↑N=E↑N

<ASSIGNMENT COMMAND>A=<NAMED EXPRESSION>B '=' <EXPRESSION>C
        <NEXT>N
        <LINENO>A↑L=UNO()
        <FLOWCHART>A↑F=MKFLC(A↑L,'U1(S,B↑L,C↑F)',A↑N)

<ITERATION COMMAND>A='FOR ' <$LETTR>B '=' <EXPRESSION>C ' TO '
. <EXPRESSION>D ' DO <' <COMMAND LIST>E '>'
        <NEXT>N
```



```
            <LINENO>A↑L=UNO()
            <U>A↑U=U(UO(1,UNO()),UO(2,UNO()))
            <FLOWCHART>A↑F=U(
            .MKFLC(A↑L,'U1(S,"B↑VAL",C↑F)',E↑L),
            .MKFLC(SEL(1,A↑U),'S','COND(EQQ(SEL("B↑VAL",S),D↑F),A↑N,SEL(2,A↑U))'),
            .MKFLC(SEL(2,A↑U),'U1(S,"B↑VAL",SEL("B↑VAL",S) + INCR(D↑F -
            . SEL("B↑VAL",S)))',E↑L),
            .E↑F)
            E↑N=SEL(1,A↑U)

<IO COMMAND>A=<IO INST>B '(L' <$NUMBR>C ',D' <$NUMBR>D ')'
            <NEXT>N
            <LINENO>A↑L=UNO()
            <U>A↑U=U(JO(1,UNO()),UO(2,UNO()),UO(3,UNO()),UO(4,UNO()),UO(5,UNO()),
            .UO(6,UNO()),UO(7,UNO()),UO(8,UNO()),UO(9,UNO()),UO(10,UNO()))
            <FLOWCHART>A↑F=U(A↑X,A↑Y)
            <XFLOWCHART>A↑X=U(
            .MKFLC(A↑L,'U(S,U(UO("L-",APPLI(SEL("GNLI",PROCLIB),    U1(S,"LIST",
            .SEL("L"  C↑VAL,LAYOUTLIB)))) ,UO("D-",APPLI(SEL("GNDI",PROCLIB),
            .U1(S,"LIST",SEL("D" D VAL,DATALIB))))))',SEL(1,A U)),
            .MKFLC(SEL(1,A↑U),'S','COND(EQU(SEL("ITEM",SEL("L-",S)),NUL),'
           . SEL(2,A↑U) ',' SEL(4,A↑U) ')'),
            .MKFLC(SEL(2,A↑U),'S','COND(EQU(SEL("ITEM",SEL("D-",S)),NUL),A↑N,'
            . SEL(3,A↑U) ')'),
            .MKFLC(SEL(3,A↑U),'U1(S,"L-"  ,APPLI(SEL("GNLI",PROCLIB),U1(S,"LIST",
            .SEL("L" C↑VAL,LAYOUTLIB))))',SEL(1,A↑U)),
            .MKFLC(SEL(4,A↑U),'S',    'COND(EQU(SEL("CODE",SEL("ITEM",SEL("L-",S))),
             ."D"),' SEL(6,A↑U) ',' SEL(5,A↑U) ')'),
            .MKFLC(SEL(5,A↑U),'B A',SEL(10,A↑U)),
            .MKFLC(SEL(6,A↑U),'S','COND(EQU(SEL("ITEM",SEL("D-",S)),NUL),'
            . SEL(7,A↑U) ',' SEL(8,A↑U) ')'))
            <YYFLOWCHART>A↑Y=U(
            .MKFLC(SEL(7,A↑U),'U1(S,"D-",APPLI(SEL("GNDI",PROCLIB),U1(S,"LIST",
            .SEL("D" D↑VAL,DATALIB))))',SEL(8,A↑U)),
            .MKFLC(SEL(8,A↑U),'B↑B',SEL(9,A↑U)),
           .U(MKFLC(SEL(9,A↑U),'U1(S,"D-",APPLI(SEL("GNDI",PROCLIB),U1(S,"LIST",
            .SEL("LIST",SEL("D-",S)))))',SEL(10,A↑U)),
            .MKFLC(SEL(10,A↑U),'U1(S,"L-",APPLI(SEL("GNLI",PROCLIB),U1(S,"LIST",
            .SEL("LIST",SEL("L-",S)))))',SEL(1,A↑U))
            .))

<IO INST>A='INPUT';B='OUTPUT'
            <A>A↑A='PROJECT(S,READX(SEL("REPL",SEL("ITEM",SEL("L-",S))),SEL("CODE",
            .SEL("ITEM",SEL("L-",S)))))';
            .B↑A='PROJECT(S,PRINTX(SEL("REPL",SEL("ITEM",SEL("L-",S))),SEL("CODE",
             .SEL("ITEM",SEL("L-",S)))))'
            <B>A↑B='U1(S,SEL("ITEM",SEL("D-",S)),READX(SEL("REPL",SEL("ITEM",
            .SEL("L-",S))),SEL("CODE",SEL("ITEM",SEL("L-",S)))))';
            .B↑B='PROJECT(S,PRINTX(SEL("REPL",SEL("ITEM",SEL("L-",S))),SEL("CODE",
            .SEL("ITEM",SEL("L-",S))),SEL(SEL("ITEM",SEL("D-",S)),S)))'
END
```



SOURCE PROGRAM

```
LAYOUT STREAM L1=4D,4D,4D;
DATA STREAM D1=A,B;
A=11;
B=22;
OUTPUT(L1,D1);
END
```

LISTING OF ATTRIBUTES OF PROGRAM NODE

DISPLAY OF FLOWCHART-F

```
16      ----------
FUN     U1(S,"A",11)
NXT     17

17      ----------
FUN     U1(S,"B",22)
NXT     18

18      ----------
FUN     U(S,U(UO("L-",APPLI(SEL("GNLI",PROCLIB),    U1(S,"LIST",SEL("L"  1,LAYOU
TLIB)))) ,UO("D-",APPLI(SEL("GNDI",PROCLIB),U1(S,"LIST",SEL("D" 1,DATALIB))))))
NXT     19

19      ----------
FUN     S
NXT     COND(EQU(SEL("ITEM",SEL("L-",S)),NUL),20,22)

20      ----------
FUN     S
NXT     COND(EQU(SEL("ITEM",SEL("D-",S)),NUL),NUL,21)

21      ----------
FUN     U1(S,"L-"  ,APPLI(SEL("GNLI",PROCLIB),U1(S,"LIST",SEL("L" 1,LAYOUTLIB)))
)
NXT     19

22      ----------
FUN     S
NXT     COND(EQU(SEL("CODE",SEL("ITEM",SEL("L-",S))),"D"),24,23)

23      ----------
FUN     PROJECT(S,PRINTX(SEL("REPL",SEL("ITEM",SEL("L-",S))),SEL("CODE",SEL("ITE
M",SEL("L-",S)))))
NXT     28

24      ----------
FUN     S
NXT     COND(EQU(SEL("ITEM",SEL("D-",S)),NUL),25,26)

25      ----------
FUN     U1(S,"D-",APPLI(SEL("GNDI",PROCLIB),U1(S,"LIST",SEL("D" 1,DATALIB))))
NXT     26

26      ----------
FUN     PROJECT(S,PRINTX(SEL("REPL",SEL("ITEM",SEL("L-",S))),SEL("CODE",SEL("ITE
M",SEL("L-",S))),SEL(SEL("ITEM",SEL("D-",S)),S)))
NXT     27

27      ----------
FUN     U1(S,"D-",APPLI(SEL("GNDI",PROCLIB),U1(S,"LIST",SEL("LIST",SEL("D-",S)))
))
NXT     28

28      ----------
FUN     U1(S,"L-",APPLI(SEL("GNLI",PROCLIB),U1(S,"LIST",SEL("LIST",SEL("L-",S)))
```

```
))
NXT     19


          DISPLAY OF PROCLIB-P
GNLI      ----------
     0    ----------
FUN       S
NXT       COND(EQU(SEL("LIST",S),NUL),4,1)

     1    ----------
FUN       S
NXT       COND(LTN(APPLIF(SEL("REPL",SEL("ELE",SEL("LIST",S))),S),1),2,3)

     2    ----------
FUN       U1(S,"LIST",SEL("NXT",SEL("LIST",S)))
NXT       0

     3    ----------
FUN       U(UO("ITEM",U(UO("REPL",APPLIF(SEL("REPL",SEL("UNIT",SEL("ELE",SEL("LIST
",S)))),S)),UO("CODE",SEL("CODE",SEL("UNIT",SEL("ELE",SEL("LIST",S)))))),UO("LI
ST",U1(SEL("LIST",S),"REPL?ELE",MKCON(APPLIF(SEL("REPL",SEL("ELE",SEL("LIST",S))
),S) - 1))))
NXT

     4    ----------
FUN       NUL
NXT


GNDI      ----------
     0    ----------
FUN       S
NXT       COND(EQU(SEL("LIST",S),NUL),4,1)

     1    ----------
FUN       S
NXT       COND(EQU(APPLIF(SEL("E2",SEL("ELE",SEL("LIST",S))),S) - APPLIF(SEL("E1",
SEL("ELE",SEL("LIST",S))),S),0),2,3)

     2    ----------
FUN       U(UO("ITEM",APPLIF(SEL("B",SEL("ELE",SEL("LIST",S))),U1(S,SEL("I",SEL("E
LE",SEL("LIST",S))),APPLIF(SEL("E1",SEL("ELE",SEL("LIST",S))),S)))),UO("LIST",SE
L("NXT",SEL("LIST",S))))
NXT

     3    ----------
FUN       U(UO("ITEM",APPLIF(SEL("B",SEL("ELE",SEL("LIST",S))),U1(S,SEL("I",SEL("E
LE",SEL("LIST",S))),APPLIF(SEL("E1",SEL("ELE",SEL("LIST",S))),S)))),UO("LIST",U1
(SEL("LIST",S),"E1?ELE",MKCON(APPLIF(SEL("E1",SEL("ELE",SEL("LIST",S))),S) + INC
R(APPLIF(SEL("E2",SEL("ELE",SEL("LIST",S))),S) - APPLIF(SEL("E1",SEL("ELE",SEL("
LIST",S))),S))     ))))
NXT

     4    ----------
FUN       NUL
NXT
```

```
          DISPLAY OF LAYOUTLIB-L
L1     ----------
     ELE      ----------
REPL     1
          UNIT      ----------
REPL     4
CODE     D

     NXT      ----------
          ELE      ----------
REPL     1
               UNIT      ----------
REPL     4
CODE     D

          NXT      ----------
               ELE      ----------
REPL     1
                    UNIT      ----------
REPL     4
CODE     D

NXT
```

```
          DISPLAY OF DATALIB-D
D1     ----------
     ELE      ----------
B     "A"
E1     0
E2     0

     NXT      ----------
          ELE      ----------
B     "B"
E1     0
E2     0

NXT
```

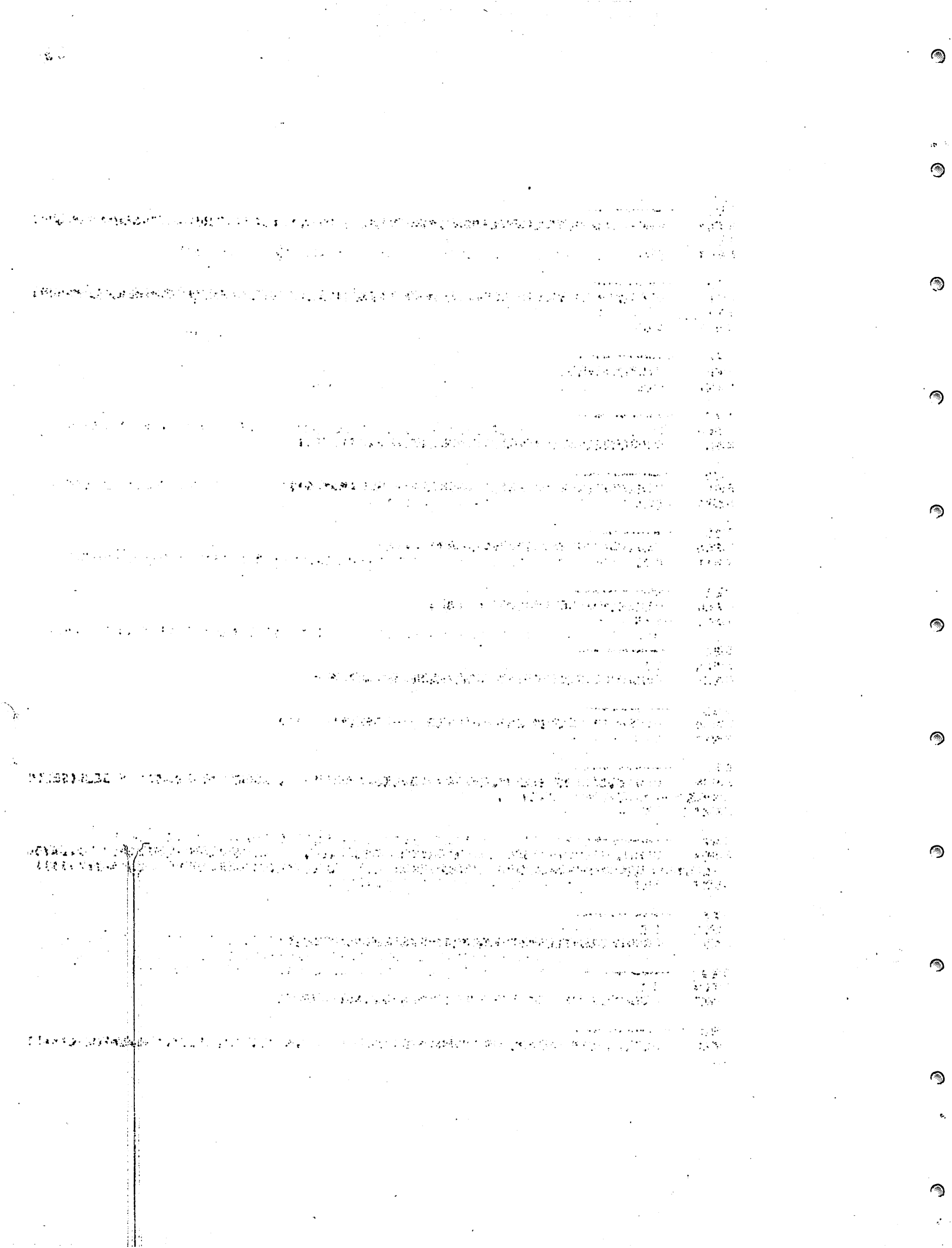
```
                APPLY FLOWCHART TO NULL STATE VECTOR S

  11  22  11  22  11  22

                FINAL STATE VECTOR
A      11
B      22
D-
L-

TIME TO PROCESS SEMANTIC DESCRIPTION 1714 MILLISEC
TIME TO PARSE 1155 MILLISEC
TIME TO EVALUATE SEMANTIC ATTRIBUTES 6995 MILLISEC
'EXECUTION' TIME 2311 MILLISEC
'STATEMENTS' EXECUTED 110
```

SOURCE PROGRAM

```
LAYOUT STREAM L1=3D,0L;
LAYOUT STREAM L1=(48-(J+J))B,J<4D>,0L;
DATA STREAM D1=A1<I> FOR I=1 TO J;
FOR J=1 TO 2 DO <A1<J>=1;OUTPUT(L1,D1)>;
FOR J=3 TO 5 DO <A1<J>=1;
   FOR K=(J-1) TO 2 DO <A1<K>=(A1<K>+A1<(K-1)>)>;
   OUTPUT(L1,D1)>;
END
```

```
LISTING OF ATTRIBUTES OF PROGRAM NODE


          DISPLAY OF FLOWCHART-F
57      ----------
FUN      U1(S,"J",1)
NXT      61

58      ----------
FUN      S
NXT      COND(EQQ(SEL("J",S),2),76,SEL(2,ATT-60))

59      ----------
FUN      U1(S,"J",SEL("J",S) + INCR(2 - SEL("J",S)))
NXT      61

61      ----------
FUN      U1(S,COMPOS(SEL("J",S),"A" 1),1)
NXT      62

62      ----------
FUN      U(S,U(UO("L-",APPLI(SEL("GNLI",PROCLIB),    U1(S,"LIST",SEL("L"  1,LAYOU
TLIB)))) ,UO("D-",APPLI(SEL("GNDI",PROCLIB),U1(S,"LIST",SEL("D" 1,DATALIB))))))
NXT      63

63      ----------
FUN      S
NXT      COND(EQU(SEL("ITEM",SEL("L-",S)),NUL),64,66)

64      ----------
FUN      S
NXT      COND(EQU(SEL("ITEM",SEL("D-",S)),NUL),58,65)

65      ----------
FUN      U1(S,"L-"   ,APPLI(SEL("GNLI",PROCLIB),U1(S,"LIST",SEL("L" 1,LAYOUTLIB)))
)
NXT      63

66      ----------
FUN      S
NXT      COND(EQU(SEL("CODE",SEL("ITEM",SEL("L-",S))),"D"),68,67)

67      ----------
FUN      PROJECT(S,PRINTX(SEL("REPL",SEL("ITEM",SEL("L-",S))),SEL("CODE",SEL("ITE
M",SEL("L-",S)))))
NXT      72

68      ----------
FUN      S
NXT      COND(EQU(SEL("ITEM",SEL("D-",S)),NUL),69,70)

69      ----------
FUN      U1(S,"D-",APPLI(SEL("GNDI",PROCLIB),U1(S,"LIST",SEL("D" 1,DATALIB))))
NXT      70

70      ----------
FUN      PROJECT(S,PRINTX(SEL("REPL",SEL("ITEM",SEL("L-",S))),SEL("CODE",SEL("ITE
M",SEL("L-",S))),SEL(SEL("ITEM",SEL("D-",S)),S)))
NXT      71
```

```
71      ----------
FUN      U1(S,"D-",APPLI(SEL("GNDI",PROCLIB),U1(S,"LIST",SEL("LIST",SEL("D-",S)))
))
NXT      72

72      ----------
FUN      U1(S,"L-",APPLI(SEL("GNLI",PROCLIB),U1(S,"LIST",SEL("LIST",SEL("L-",S)))
))
NXT      63

76      ----------
FUN      U1(S,"J",3)
NXT      80

77      ----------
FUN      S
NXT      COND(EQQ(SEL("J",S),5),NUL,SEL(2,ATT.79))

78      ----------
FUN      U1(S,"J",SEL("J",S) + INCR(5 - SEL("J",S)))
NXT      80

80      ----------
FUN      U1(S,COMPOS(SEL("J",S),"A" 1),1)
NXT      81

81      ----------
FUN      U1(S,"K",(SEL("J",S) - 1))
NXT      85

82      ----------
FUN      S
NXT      COND(EQQ(SEL("K",S),2),87,SEL(2,ATT.84))

83      ----------
FUN      U1(S,"K",SEL("K",S) + INCR(2 - SEL("K",S)))
NXT      85

85      ----------
FUN      U1(S,COMPOS(SEL("K",S),"A" 1),(SEL(SEL("K",S),SEL("A" 1,S)) + SEL((SEL("
K",S) - 1),SEL("A" 1,S))))
NXT      82

87      ----------
FUN      U(S,U(UO("L-",APPLI(SEL("GNLI",PROCLIB),    U1(S,"LIST",SEL("L" 1,LAYOU
TLIB)))) ,UO("D-",APPLI(SEL("GNDI",PROCLIB),U1(S,"LIST",SEL("D" 1,DATALIB)))))
NXT      88

88      ----------
FUN      S
NXT      COND(EQU(SEL("ITEM",SEL("L-",S)),NUL),89,91)

89      ----------
FUN      S
NXT      COND(EQU(SEL("ITEM",SEL("D-",S)),NUL),77,90)

90      ----------
FUN      U1(S,"L-"  ,APPLI(SEL("GNLI",PROCLIB),U1(S,"LIST",SEL("L" 1,LAYOUTLIB)))
)
```

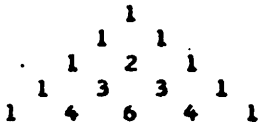


```
NXT      88

91       ----------
FUN      S
NXT      COND(EQU(SEL("CODE",SEL("ITEM",SEL("L-",S))),"D"),93,92)

92       ----------
FUN      PROJECT(S,PRINTX(SEL("REPL",SEL("ITEM",SEL("L-",S))),SEL("CODE",SEL("ITE
M",SEL("L-",S)))))
NXT      97

93       ----------
FUN      S
NXT      COND(EQU(SEL("ITEM",SEL("D-",S)),NUL),94,95)

94       ----------
FUN      U1(S,"D-",APPLI(SEL("GNDI",PROCLIB),U1(S,"LIST",SEL("D" 1,DATALIB))))
NXT      95

95       ----------
FUN      PROJECT(S,PRINTX(SEL("REPL",SEL("ITEM",SEL("L-",S))),SEL("CODE",SEL("ITE
M",SEL("L-",S))),SEL(SEL("ITEM",SEL("D-",S)),S)))
NXT      96

96       ----------
FUN      U1(S,"D-",APPLI(SEL("GNDI",PROCLIB),U1(S,"LIST",SEL("LIST",SEL("D-",S)))
))
NXT      97

97       ----------
FUN      U1(S,"L-",APPLI(SEL("GNLI",PROCLIB),U1(S,"LIST",SEL("LIST",SEL("L-",S)))
))
NXT      88


           DISPLAY OF PROCLIB-P
GNLI      ----------
     0       ----------
FUN      S
NXT      COND(EQU(SEL("LIST",S),NUL),4,1)

     1       ----------
FUN      S
NXT      COND(LTN(APPLIF(SEL("REPL",SEL("ELE",SEL("LIST",S))),S),1),2,3)

     2       ----------
FUN      U1(S,"LIST",SEL("NXT",SEL("LIST",S)))
NXT      0

     3       ----------
FUN      U(UO("ITEM",U(UO("REPL",APPLIF(SEL("REPL",SEL("UNIT",SEL("ELE",SEL("LIST
",S))))),S)),UO("CODE",SEL("CODE",SEL("UNIT",SEL("ELE",SEL("LIST",S)))))),UO("LI
ST",U1(SEL("LIST",S),"REPL?ELE",MKCON(APPLIF(SEL("REPL",SEL("ELE",SEL("LIST",S))
),S) - 1))))
NXT

     4       ----------
FUN      NUL
NXT
```

```
GNDI      ----------
     0      ----------
FUN       S
NXT       COND(EQU(SEL("LIST",S),NUL),4,1)

     1      ----------
FUN       S
NXT       COND(EQU(APPLIF(SEL("E2",SEL("ELE",SEL("LIST",S))),S) - APPLIF(SEL("E1",
SEL("ELE",SEL("LIST",S))),S),0),2,3)

     2      ----------
FUN       U(UO("ITEM",APPLIF(SEL("B",SEL("ELE",SEL("LIST",S))),U1(S,SEL("I",SEL("E
LE",SEL("LIST",S))),APPLIF(SEL("E1",SEL("ELE",SEL("LIST",S))),S))),UO("LIST",SE
L("NXT",SEL("LIST",S))))
NXT

     3      ----------
FUN       U(UO("ITEM",APPLIF(SEL("B",SEL("ELE",SEL("LIST",S))),U1(S,SEL("I",SEL("E
LE",SEL("LIST",S))),APPLIF(SEL("E1",SEL("ELE",SEL("LIST",S))),S))),UO("LIST",U1
(SEL("LIST",S),"E1?ELE",MKCON(APPLIF(SEL("E1",SEL("ELE",SEL("LIST",S))),S) + INC
R(APPLIF(SEL("E2",SEL("ELE",SEL("LIST",S))),S) - APPLIF(SEL("E1",SEL("ELE",SEL("
LIST",S))),S))     ))))
NXT

     4      ----------
FUN       NUL
NXT



          DISPLAY OF LAYOUTLIB-L
L1     ----------
     ELE       ----------
REPL      1
           UNIT      ----------
REPL      (48 - (SEL("J",S) + SEL("J",S)))
CODE      B


     NXT       ----------
           ELE       ----------
REPL       SEL("J",S)
                UNIT      ----------
REPL       4
CODE       D


           NXT       ----------
                ELE       ----------
REPL       1
                     UNIT      ----------
REPL       0
CODE       L


NXT
```

```
          DISPLAY OF DATALIB-D
D1      ----------
      ELE      ----------
B      COMPOS(SEL("I",S),"A" 1)
I       I
E1       1
E2      SEL("J",S)

NXT
```

APPLY FLOWCHART TO NULL STATE VECTOR S



FINAL STATE VECTOR

J	5
K	2
A1	————
1	1
5	1
4	4
3	6
2	4

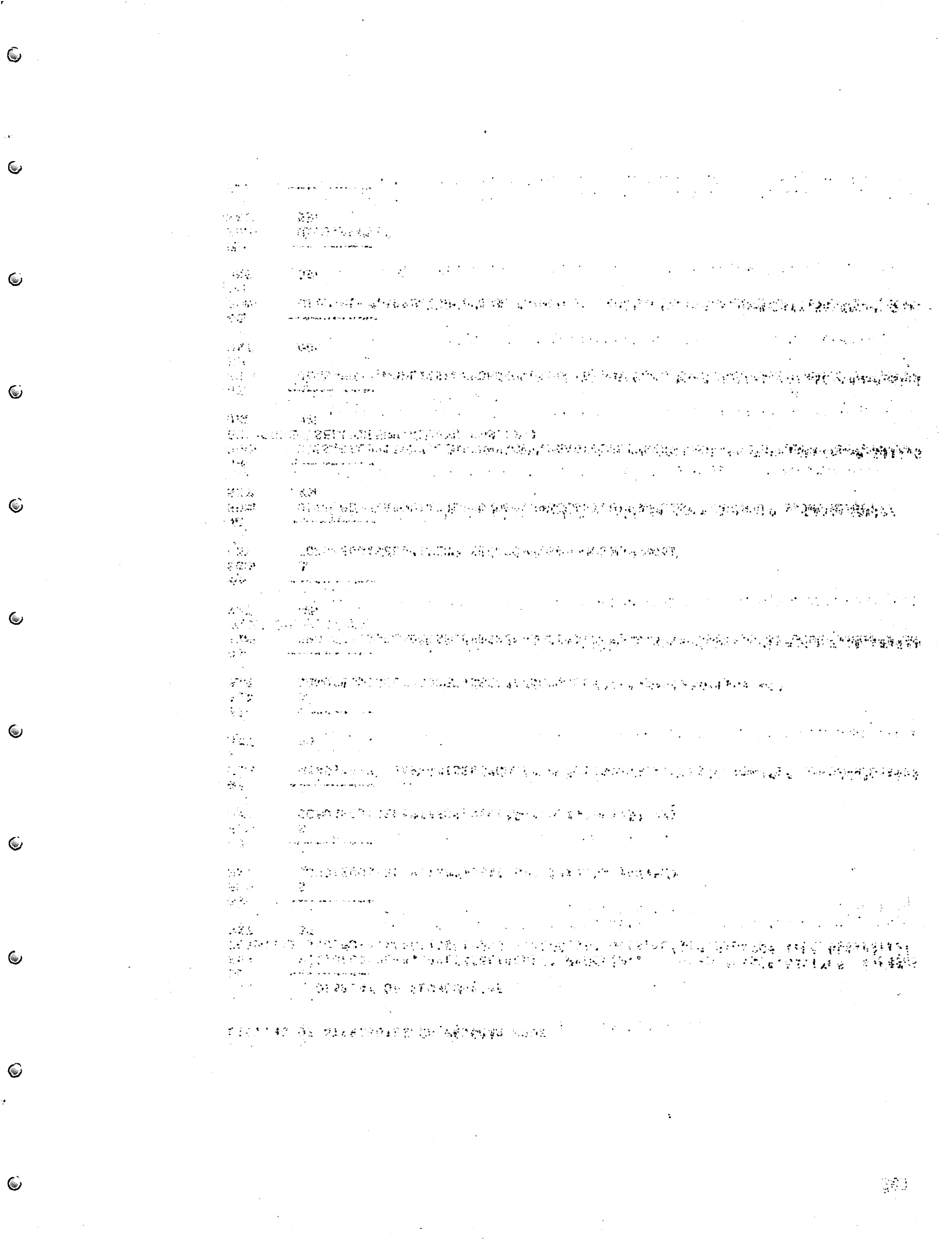
D-
L-

TIME TO PROCESS SEMANTIC DESCRIPTION 1714 MILLISEC
TIME TO PARSE 7164 MILLISEC
TIME TO EVALUATE SEMANTIC ATTRIBUTES 20127 MILLISEC
'EXECUTION' TIME 11580 MILLISEC
'STATEMENTS' EXECUTED 363

SOURCE PROGRAM

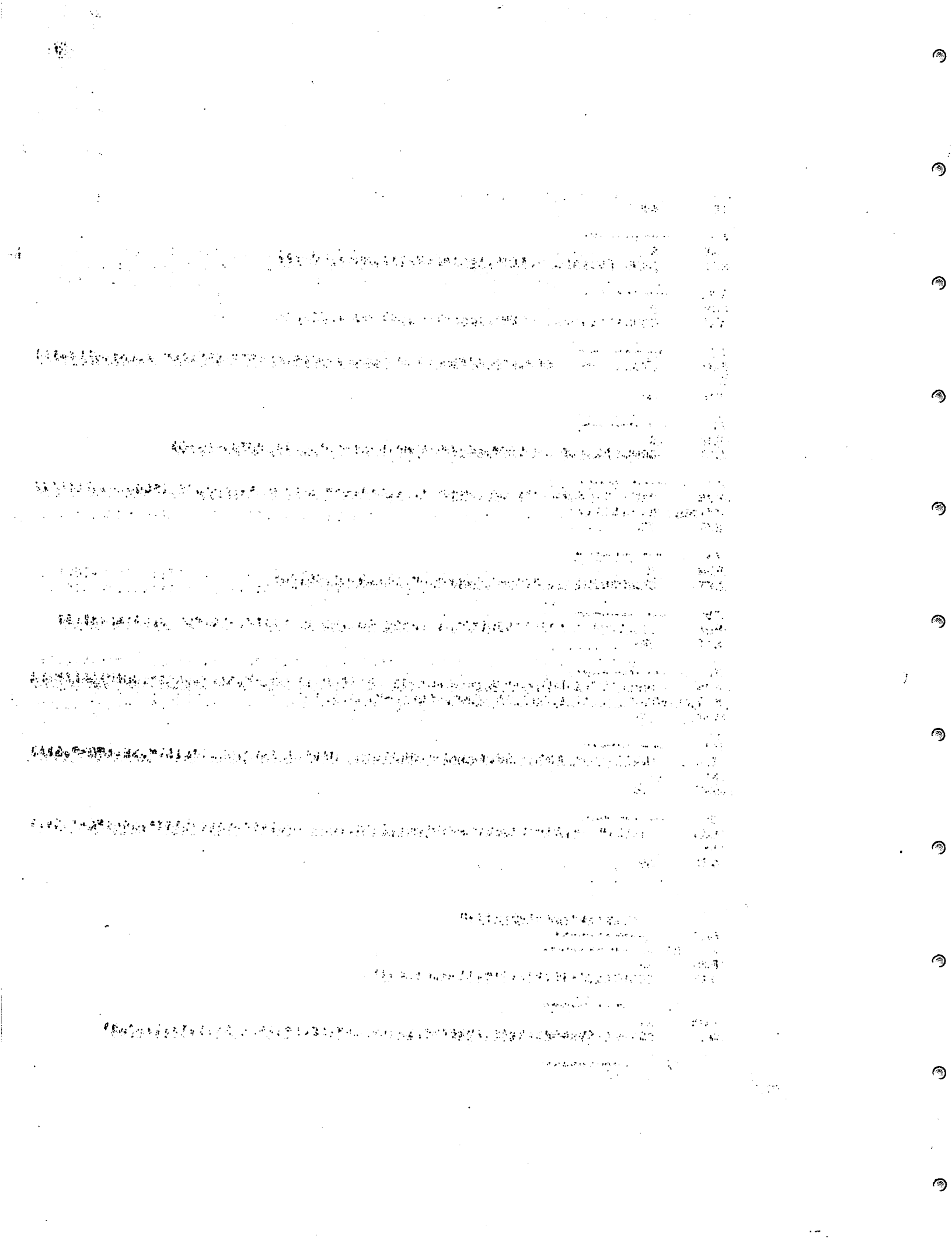
```
LAYOUT STREAM L1=3D,0L;
DATA STREAM D1=N;
LAYOUT STREAM L2=1B,3D,1B,3D,0L;
DATA STREAM D2=A,B;
DATA STREAM D3=B,A;
INPUT(L1,D1);
FOR X=1 TO N DO <INPUT(L2,D2);OUTPUT(L2,D3)>;
END
```

```
LISTING OF ATTRIBUTES OF PROGRAM NODE


          DISPLAY OF FLOWCHART-F
38     ----------
FUN     U(S,U(UO("L-",APPLI(SEL("GNLI",PROCLIB),    U1(S,"LIST",SEL("L"  1,LAYOU
TLIB)))) ,UO("D-",APPLI(SEL("GNDI",PROCLIB),U1(S,"LIST",SEL("D" 1,DATALIB))))))
NXT     39

39     ----------
FUN     S
NXT     COND(EQU(SEL("ITEM",SEL("L-",S)),NUL),40,42)

40     ----------
FUN     S
NXT     COND(EQU(SEL("ITEM",SEL("D-",S)),NUL),51,41)

41     ----------
FUN     U1(S,"L-"  ,APPLI(SEL("GNLI",PROCLIB),U1(S,"LIST",SEL("L" 1,LAYOUTLIB)))
)
NXT     39

42     ----------
FUN     S
NXT     COND(EQU(SEL("CODE",SEL("ITEM",SEL("L-",S))),"D"),44,43)

43     ----------
FUN     PROJECT(S,READX(SEL("REPL",SEL("ITEM",SEL("L-",S))),SEL("CODE",SEL("ITEM
",SEL("L-",S)))))
NXT     48

44     ----------
FUN     S
NXT     COND(EQU(SEL("ITEM",SEL("D-",S)),NUL),45,46)

45     ----------
FUN     U1(S,"D-",APPLI(SEL("GNDI",PROCLIB),U1(S,"LIST",SEL("D" 1,DATALIB))))
NXT     46

46     ----------
FUN     U1(S,SEL("ITEM",SEL("D-",S)),READX(SEL("REPL",SEL("ITEM",SEL("L-",S))),S
EL("CODE",SEL("ITEM",SEL("L-",S)))))
NXT     47

47     ----------
FUN     U1(S,"D-",APPLI(SEL("GNDI",PROCLIB),U1(S,"LIST",SEL("LIST",SEL("D-",S)))
))
NXT     48

48     ----------
FUN     U1(S,"L-",APPLI(SEL("GNLI",PROCLIB),U1(S,"LIST",SEL("LIST",SEL("L-",S)))
))
NXT     39

51     ----------
FUN     U1(S,"X",1)
NXT     55

52     ----------
```



```
FUN     S
NXT     COND(EQQ(SEL("X",S),SEL("N",S)),NUL,SEL(2,ATT.541))

53      ----------
FUN     U1(S,"X",SEL("X",S) + INCR(SEL("N",S) - SEL("X",S)))
NXT     55

55      ----------
FUN     U(S,U(UO("L-",APPLI(SEL("GNLI",PROCLIB),    U1(S,"LIST",SEL("L" 2,LAYOU
TLIB)))) ,UO("D-",APPLI(SEL("GNDI",PROCLIB),U1(S,"LIST",SEL("D" 2,DATALIB))))))
NXT     56

56      ----------
FUN     S
NXT     COND(EQU(SEL("ITEM",SEL("L-",S)),NUL),57,59)

57      ----------
FUN     S
NXT     COND(EQU(SEL("ITEM",SEL("D-",S)),NUL),68,58)

58      ----------
FUN     U1(S,"L-"  ,APPLI(SEL("GNLI",PROCLIB),U1(S,"LIST",SEL("L" 2,LAYOUTLIB)))
)
NXT     56

59      ----------
FUN     S
NXT     COND(EQU(SEL("CODE",SEL("ITEM",SEL("L-",S))),"D"),61,60)

60      ----------
FUN     PROJECT(S,READX(SEL("REPL",SEL("ITEM",SEL("L-",S))),SEL("CODE",SEL("ITEM
",SEL("L-",S)))))
NXT     65

61      ----------
FUN     S
NXT     COND(EQU(SEL("ITEM",SEL("D-",S)),NUL),62,63)

62      ----------
FUN     U1(S,"D-",APPLI(SEL("GNDI",PROCLIB),U1(S,"LIST",SEL("D" 2,DATALIB))))
NXT     63

63      ----------
FUN     U1(S,SEL("ITEM",SEL("D-",S)),READX(SEL("REPL",SEL("ITEM",SEL("L-",S))),S
EL("CODE",SEL("ITEM",SEL("L-",S)))))
NXT     64

64      ----------
FUN     U1(S,"D-",APPLI(SEL("GNDI",PROCLIB),U1(S,"LIST",SEL("LIST",SEL("D-",S)))
))
NXT     65

65      ----------
FUN     U1(S,"L-",APPLI(SEL("GNLI",PROCLIB),U1(S,"LIST",SEL("LIST",SEL("L-",S)))
))
NXT     56

68      ----------
FUN     U(S,U(UO("L-",APPLI(SEL("GNLI",PROCLIB),    U1(S,"LIST",SEL("L" 2,LAYOU
TLIB)))) ,UO("D-",APPLI(SEL("GNDI",PROCLIB),U1(S,"LIST",SEL("D" 3,DATALIB))))))
```

```
NXT     69

69      ----------
FUN     S
NXT     COND(EQU(SEL("ITEM",SEL("L-",S)),NUL),70,72)

70      ----------
FUN     S
NXT     COND(EQU(SEL("ITEM",SEL("D-",S)),NUL),52,71)

71      ----------
FUN     U1(S,"L-"   ,APPLI(SEL("GNLI",PROCLIB),U1(S,"LIST",SEL("L" 2,LAYOUTLIB)))
)
NXT     69

72      ----------
FUN     S
NXT     COND(EQU(SEL("CODE",SEL("ITEM",SEL("L-",S))),"D"),74,73)

73      ----------
FUN     PROJECT(S,PRINTX(SEL("REPL",SEL("ITEM",SEL("L-",S))),SEL("CODE",SEL("ITE
M",SEL("L-",S)))))
NXT     78

74      ----------
FUN     S
NXT     COND(EQU(SEL("ITEM",SEL("D-",S)),NUL),75,76)

75      ----------
FUN     U1(S,"D-",APPLI(SEL("GNDI",PROCLIB),U1(S,"LIST",SEL("D" 3,DATALIB))))
NXT     76

76      ----------
FUN     PROJECT(S,PRINTX(SEL("REPL",SEL("ITEM",SEL("L-",S))),SEL("CODE",SEL("ITE
M",SEL("L-",S))),SEL(SEL("ITEM",SEL("D-",S)),S)))
NXT     77

77      ----------
FUN     U1(S,"D-",APPLI(SEL("GNDI",PROCLIB),U1(S,"LIST",SEL("LIST",SEL("D-",S)))
))
NXT     78

78      ----------
FUN     U1(S,"L-",APPLI(SEL("GNLI",PROCLIB),U1(S,"LIST",SEL("LIST",SEL("L-",S)))
))
NXT     69


          DISPLAY OF PROCLIB-P
GNLI     ----------
     0     ----------
FUN     S
NXT     COND(EQU(SEL("LIST",S),NUL),4,1)

     1     ----------
FUN     S
NXT     COND(LTN(APPLIF(SEL("REPL",SEL("ELE",SEL("LIST",S))),S),1),2,3)

     2     ----------
```



```
FUN     U1(S,"LIST",SEL("NXT",SEL("LIST",S)))
NXT     0

    3   ----------
FUN     U(UO("ITEM",U(UO("REPL",APPLIF(SEL("REPL",SEL("UNIT",SEL("ELE",SEL("LIST
",S)))),S)),UO("CODE",SEL("CODE",SEL("UNIT",SEL("ELE",SEL("LIST",S)))))),UO("LI
ST",U1(SEL("LIST",S),"REPL?ELE",MKCON(APPLIF(SEL("REPL",SEL("ELE",SEL("LIST",S))
),S) - 1))))
NXT

    4   ----------
FUN     NUL
NXT


GND1    ----------
    0   ----------
FUN     S
NXT     COND(EQU(SEL("LIST",S),NUL),4,1)

    1   ----------
FUN     S
NXT     COND(EQU(APPLIF(SEL("E2",SEL("ELE",SEL("LIST",S))),S) - APPLIF(SEL("E1",
SEL("ELE",SEL("LIST",S))),S),0),2,3)

    2   ----------
FUN     U(UO("ITEM",APPLIF(SEL("B",SEL("ELE",SEL("LIST",S))),U1(S,SEL("I",SEL("E
LE",SEL("LIST",S))),APPLIF(SEL("E1",SEL("ELE",SEL("LIST",S))),S)))),UO("LIST",SE
L("NXT",SEL("LIST",S))))
NXT

    3   ----------
FUN     U(UO("ITEM",APPLIF(SEL("B",SEL("ELE",SEL("LIST",S))),U1(S,SEL("I",SEL("E
LE",SEL("LIST",S))),APPLIF(SEL("E1",SEL("ELE",SEL("LIST",S))),S)))),UO("LIST",U1
(SEL("LIST",S),"E1?ELE",MKCON(APPLIF(SEL("E1",SEL("ELE",SEL("LIST",S))),S) + INC
R(APPLIF(SEL("E2",SEL("ELE",SEL("LIST",S))),S) - APPLIF(SEL("E1",SEL("ELE",SEL("
LIST",S))),S))    ))))
NXT

    4   ----------
FUN     NUL
NXT



          DISPLAY OF LAYOUTLIB-L
L1     ----------
    ELE     ----------
REPL     1
          UNIT     ----------
REPL     3
CODE     D


    NXT     ----------
          ELE     ----------
REPL     1
                UNIT     ----------
REPL     0
```

```
CODE      L

NXT

L2      ----------
     ELE      ----------
REPL      1
          UNIT      ----------
REPL      1
CODE      B

     NXT      ----------
          ELE      ----------
REPL      1
               UNIT      ----------
REPL      3
CODE      D

          NXT      ----------
               ELE      ----------
REPL      1
                    UNIT      ----------
REPL      1
CODE      B

               NXT      ----------
                    ELE      ----------
REPL      1
                         UNIT      ----------
REPL      3
CODE      D

                    NXT      ----------
                         ELE      ----------
REPL      1
                              UNIT      ----------
REPL      0
CODE      L

NXT
```

```
          DISPLAY OF DATALIB-D
D1      ----------
     ELE      ----------
B       "N"
E1        0
E2        0
```

```
NXT

D2      ----------
      ELE       ----------
B      "A"
E1      0
E2      0

      NXT       ----------
            ELE       ----------
B      "B"
E1      0
E2      0

NXT


D3      ----------
      ELE       ----------
B      "B"
E1      0
E2      0

      NXT       ----------
            ELE       ----------
B      "A"
E1      0
E2      0

NXT
```

```
          APPLY FLOWCHART TO NULL STATE VECTOR S

READ(3,D) RETURNS 3
READ(0,L)
READ(1,B)
READ(3,D) RETURNS 1
READ(1,B)
READ(3,D) RETURNS 2
READ(0,L)
   2  1
READ(1,B)
READ(3,D) RETURNS 11
READ(1,B)
READ(3,D) RETURNS 22
READ(0,L)
  22  11
READ(1,B)
READ(3,D) RETURNS 111
READ(1,B)
READ(3,D) RETURNS 222
READ(0,L)
 222 111


          FINAL STATE VECTOR
N      3
X      3
A      111
B      222
D-
L-

TIME TO PROCESS SEMANTIC DESCRIPTION 1572 MILLISEC
TIME TO PARSE 2382 MILLISEC
TIME TO EVALUATE SEMANTIC ATTRIBUTES 15548 MILLISEC
'EXECUTION' TIME 11505 MILLISEC
'STATEMENTS' EXECUTED 440
```

SOURCE PROGRAM

```
LAYOUT STREAM L1=(48-(J+J))B,J<4D>,0L;
DATA STREAM D1=A1<I> FOR I=K TO (K+(J-1));
DATA STREAM D2=A1<I> FOR I=K TO (K-(J-1));
K=1;
FOR J=1 TO 4 DO <INPUT(L1,D1);K=(K+J)>;
K=(K-1);
FOR J=4 TO 1 DO <OUTPUT(L1,D2);K=(K-J)>;
END
```

```
LISTING OF ATTRIBUTES OF PROGRAM NODE


          DISPLAY OF FLOWCHART-F
107       ----------
FUN       U1(S,"K",1)
NXT       108

108       ----------
FUN       U1(S,"J",1)
NXT       112

109       ----------
FUN       S
NXT       COND(EQQ(SEL("J",S),4),127,SEL(2,ATT.111))

110       ----------
FUN       U1(S,"J",SEL("J",S) + INCR(4 - SEL("J",S)))
NXT       112

112       ----------
FUN       U(S,U(UO("L-",APPLI(SEL("GNLI",PROCLIB),    U1(S,"LIST",SEL("L"  1,LAYOU
TLIB)))) ,UO("D-",APPLI(SEL("GNDI",PROCLIB),U1(S,"LIST",SEL("D" 1,DATALIB))))))
NXT       113

113       ----------
FUN       S
NXT       COND(EQU(SEL("ITEM",SEL("L-",S)),NUL),114,116)

114       ----------
FUN       S
NXT       COND(EQU(SEL("ITEM",SEL("D-",S)),NUL),125,115)

115       ----------
FUN       U1(S,"L-"  ,APPLI(SEL("GNLI",PROCLIB),U1(S,"LIST",SEL("L" 1,LAYOUTLIB)))
)
NXT       113

116       ----------
FUN       S
NXT       COND(EQU(SEL("CODE",SEL("ITEM",SEL("L-",S))),"D"),118,117)

117       ----------
FUN       PROJECT(S,READX(SEL("REPL",SEL("ITEM",SEL("L-",S))),SEL("CODE",SEL("ITEM
",SEL("L-",S)))))
NXT       122

118       ----------
FUN       S
NXT       COND(EQU(SEL("ITEM",SEL("D-",S)),NUL),119,120)

119       ----------
FUN       U1(S,"D-",APPLI(SEL("GNDI",PROCLIB),U1(S,"LIST",SEL("D" 1,DATALIB))))
NXT       120

120       ----------
FUN       U1(S,SEL("ITEM",SEL("D-",S)),READX(SEL("REPL",SEL("ITEM",SEL("L-",S))),S
EL("CODE",SEL("ITEM",SEL("L-",S)))))
NXT       121
```

```
121     ----------
FUN     U1(S,"D-",APPLI(SEL("GNDI",PROCLIB),U1(S,"LIST",SEL("LIST",SEL("D-",S)))
))
NXT     122

122     ----------
FUN     U1(S,"L-",APPLI(SEL("GNLI",PROCLIB),U1(S,"LIST",SEL("LIST",SEL("L-",S)))
))
NXT     113

125     ----------
FUN     U1(S,"K",(SEL("K",S) + SEL("J",S)))
NXT     109

127     ----------
FUN     U1(S,"K",(SEL("K",S) - 1))
NXT     128

128     ----------
FUN     U1(S,"J",4)
NXT     132

129     ----------
FUN     S
NXT     COND(EQQ(SEL("J",S),1),NUL,SEL(2,ATT.131))

130     ----------
FUN     U1(S,"J",SEL("J",S) + INCR(1 - SEL("J",S)))
NXT     132

132     ----------
FUN     U(S,U(UO("L-",APPLI(SEL("GNLI",PROCLIB),    U1(S,"LIST",SEL("L" 1,LAYOU
TLIB)))) ,UO("D-",APPLI(SEL("GNDI",PROCLIB),U1(S,"LIST",SEL("D" 2,DATALIB))))))
NXT     133

133     ----------
FUN     S
NXT     COND(EQU(SEL("ITEM",SEL("L-",S)),NUL),134,136)

134     ----------
FUN     S
NXT     COND(EQU(SEL("ITEM",SEL("D-",S)),NUL),145,135)

135     ----------
FUN     U1(S,"L-"  ,APPLI(SEL("GNLI",PROCLIB),U1(S,"LIST",SEL("L" 1,LAYOUTLIB)))
)
NXT     133

136     ----------
FUN     S
NXT     COND(EQU(SEL("CODE",SEL("ITEM",SEL("L-",S))),"D"),138,137)

137     ----------
FUN     PROJECT(S,PRINTX(SEL("REPL",SEL("ITEM",SEL("L-",S))),SEL("CODE",SEL("ITE
M",SEL("L-",S)))))
NXT     142

138     ----------
FUN     S
```

```
NXT     COND(EQU(SEL("ITEM",SEL("D-",S)),NUL),139,140)

139     ----------
FUN     U1(S,"D-",APPLI(SEL("GNDI",PROCLIB),U1(S,"LIST",SEL("D" 2,DATALIB))))
NXT     140

140     ----------
FUN     PROJECT(S,PRINTX(SEL("REPL",SEL("ITEM",SEL("L-",S))),SEL("CODE",SEL("ITE
M",SEL("L-",S))),SEL(SEL("ITEM",SEL("D-",S)),S)))
NXT     141

141     ----------
FUN     U1(S,"D-",APPLI(SEL("GNDI",PROCLIB),U1(S,"LIST",SEL("LIST",SEL("D-",S)))
))
NXT     142

142     ----------
FUN     U1(S,"L-",APPLI(SEL("GNLI",PROCLIB),U1(S,"LIST",SEL("LIST",SEL("L-",S)))
))
NXT     133

145     ----------
FUN     U1(S,"K",(SEL("K",S) - SEL("J",S)))
NXT     129



          DISPLAY OF PROCLIB-P
GNLI     ----------
     0      ----------
FUN     S
NXT     COND(EQU(SEL("LIST",S),NUL),4,1)

     1      ----------
FUN     S
NXT     COND(LTN(APPLIF(SEL("REPL",SEL("ELE",SEL("LIST",S))),S),1),2,3)

     2      ----------
FUN     U1(S,"LIST",SEL("NXT",SEL("LIST",S)))
NXT     0

     3      ----------
FUN     U(UO("ITEM",U(UO("REPL",APPLIF(SEL("REPL",SEL("UNIT",SEL("ELE",SEL("LIST
",S)))),S)),UO("CODE",SEL("CODE",SEL("UNIT",SEL("ELE",SEL("LIST",S)))))),UO("LI
ST",U1(SEL("LIST",S),"REPL?ELE",MKCON(APPLIF(SEL("REPL",SEL("ELE",SEL("LIST",S))
),S) - 1))))
NXT

     4      ----------
FUN     NUL
NXT


GNDI     ----------
     0      ----------
FUN     S
NXT     COND(EQU(SEL("LIST",S),NUL),4,1)

     1      ----------
FUN     S
```

```
NXT      COND(EQU(APPLIF(SEL("E2",SEL("ELE",SEL("LIST",S))),S) - APPLIF(SEL("E1",
SEL("ELE",SEL("LIST",S))),S),0),2,3)

     2     ----------
FUN      U(UO("ITEM",APPLIF(SEL("B",SEL("ELE",SEL("LIST",S))),U1(S,SEL("I",SEL("E
LE",SEL("LIST",S))),APPLIF(SEL("E1",SEL("ELE",SEL("LIST",S))),S))),UO("LIST",SE
L("NXT",SEL("LIST",S))))
NXT

     3     ----------
FUN      U(UO("ITEM",APPLIF(SEL("B",SEL("ELE",SEL("LIST",S))),U1(S,SEL("I",SEL("E
LE",SEL("LIST",S))),APPLIF(SEL("E1",SEL("ELE",SEL("LIST",S))),S))),UO("LIST",U1
(SEL("LIST",S),"E1?ELE",MKCON(APPLIF(SEL("E1",SEL("ELE",SEL("LIST",S))),S) + INC
R(APPLIF(SEL("E2",SEL("ELE",SEL("LIST",S))),S) - APPLIF(SEL("E1",SEL("ELE",SEL("
LIST",S))),S))    ))))
NXT

     4     ----------
FUN      NUL
NXT



          DISPLAY OF LAYOUTLIB-L
L1     ----------
     ELE     ----------
REPL     1
          UNIT     ----------
REPL     (48 - (SEL("J",S) + SEL("J",S)))
CODE     B


     NXT     ----------
          ELE     ----------
REPL     SEL("J",S)
               UNIT     ----------
REPL     4
CODE     D


          NXT     ----------
               ELE     ----------
REPL     1
                    UNIT     ----------
REPL     0
CODE     L


NXT



          DISPLAY OF DATALIB-D
D1     ----------
     ELE     ----------
B      COMPOS(SEL("I",S),"A" 1)
I      I
```

APPLY FLOWCHART TO NULL STATE VECTOR S

```
READ(46,B)
READ(4,D) RETURNS 1
READ(0,L)
READ(44,B)
READ(4,D) RETURNS 2
READ(4,D) RETURNS 3
READ(0,L)
READ(42,B)
READ(4,D) RETURNS 4
READ(4,D) RETURNS 5
READ(4,D) RETURNS 6
READ(0,L)
READ(40,B)
READ(4,D) RETURNS 7
READ(4,D) RETURNS 8
READ(4,D) RETURNS 9
READ(4,D) RETURNS 10
READ(0,L)
```

```
10   9   8   7
   6   5   4
     3   2
       1
```

FINAL STATE VECTOR

A1	
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	8
9	9
10	10
J	1
D-	
L-	
K	0

```
TIME TO PROCESS SEMANTIC DESCRIPTION 1572 MILLISEC
TIME TO PARSE 6998 MILLISEC
TIME TO EVALUATE SEMANTIC ATTRIBUTES 21779 MILLISEC
'EXECUTION' TIME 18897 MILLISEC
'STATEMENTS' EXECUTED 506
```

APPENDIX II

LISTING OF THE SEMANTICS-BASED INTERPRETER

```
*       A SEMANTICS-BASED INTERPRETER - WILLIAM D. FLANNERY
*
        TZ1 = TIME()
        UNIQUENUMBER = 0
        &STLIMIT = 40000000
        OUTPUT('OUTPUT',,'(24H                          ,80A1)')
        OUTPUT('NEWPAGE',6,'(1H1,1A1)')                                  00009900
        DATA('ATTLOC(NODE,INX)')                                         00000700
        DATA('PNODE(ALT,ATT,INH,SYN)')                                   00000800
        DATA('TREEL(KEY,LHS,PR,RHS,NVAL,NSON,VALS,SONS)')                00000900
        DATA('VNO(SLX,OBJ,LINK)')                                        00001000
*                                                                        00001100
        NULL = RPOS(0)                                                   00001200
        ALP1 = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'                              00001300
        NUM1 = '1234567890'                                              00001400
        CHARS = BREAK('''')                                              00001500
        SPACE = (SPAN(' ') | NUL)                                        00001600
        ALPHA = SPAN(ALP1)                                               00001700
        ALPHX = SPAN(ALP1 '()*+,')                                       00001800
        IDENT = ALPHA                                                    00001900
        NUMBR = SPAN(NUM1)                                               00002000
        ALNUM = SPAN(ALP1 NUM1)                                          00002100
        LETTR = ANY(ALP1)                                                00002200
        BLANKX = SPAN(' ')                                               00002300
        PSPAN = SPAN(ALP1 NUM1 '-_')                                     00002400
        ALPHA1 = SPAN(ALP1 ' ')                                          00002500
      NONTERM = (NUL | '$') ALPHA1                                       00002600
        BNONTERM = '<' NONTERM '>'                                       00002700
*ATTREFX MATCHES ATTRIBUTE REFERENCES                                    00002800
        ATTREFX = ANY(ALP1) ' ' ALPHA                                    00002900
        COPYPAT = '''' ATTREFX '''' | ATTREFX                            00003000
        ATTPAT = 'NUL' | 'ATT.' NUMBR                                    00003100
*SEMEQX MATCHES SEMANTIC EQUATIONS                                       00003200
        SEMEQX = BREAK(';')                                              00003300
*                                                                        00003400
        DEFINE('INCR(P1)')                                               00003500
        DEFINE('NEWPAGE(X)')                                             00003600
        DEFINE('DISPLAY(P,P2,LIST)')
        DEFINE('PRINTLONG(LINE,P2)X,Y,Z')                                00003800
        DEFINE('GETRULE()')                                              00003900
        DEFINE('PARSE(NON,P2,REST)SAV1,SAV3,VAL,PR,TERM')                00004000
        DEFINE('PUSH(ELEM)')                                             00004100
        DEFINE('POP()')                                                  00004200
        DEFINE('SEMPASS(N)LHS,PR,NXTSON')                                00004300
        DEFINE('FINDVAL(ATT,N)PROD,I,ATTRY,X')                           00004400
        DEFINE('SELECT(P1,P2,P3,P4)')                                    00004500
        DEFINE('UNO(X)')                                                 00004600
        DEFINE('UO(P1,P2)')                                              00004700
        DEFINE('UO(P1,P2)')                                              00004800
        DEFINE('U(P1,P2,P3,P4,P5,P6,P7,P8,P9,P10)')                      00004900
        DEFINE('U2(P1,P2)')                                              00005000
        DEFINE('U1(P1,P2,P3)X')                                          00005100
        DEFINE('COMPOS(P1,P2)')                                          00005200
        DEFINE('SEL(P1,P2)')                                             00005300
        DEFINE('ELEM(P1,P2)')                                            00005400
        DEFINE('MAKESUB(P1,P2,P3,P4)')                                   00005500
        DEFINE('COND(P1,P2,P3)')                                         00005600
```

```
*       A SEMANTICS-BASED INTERPRETER - WILLIAM D. FLANNERY
*
        TZ1 = TIME()
        UNIQUENUMBER = 0
        &STLIMIT = 40000000
        OUTPUT('OUTPUT',,'(24H                              ,80A1)')
        OUTPUT('NEWPAGE',6,'(1H1,1A1)')                                  00000900
        DATA('ATTLOC(NODE,INX)')                                         00000700
        DATA('PNODE(ALT,ATT,INH,SYN)')                                   00000800
        DATA('TREEL(KEY,LHS,PR,RHS,NVAL,NSON,VALS,SONS)')                00000900
        DATA('VNO(SLX,OBJ,LINK)')                                        00001000
*                                                                        00001100
        NULL = RPOS(0)                                                   00001200
        ALP1 = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'                              00001300
        NUM1 = '1234567890'                                              00001400
        CHARS = BREAK('''')                                              00001500
        SPACE = (SPAN(' ') | NUL)                                        00001600
        ALPHA = SPAN(ALP1)                                               00001700
        ALPHX = SPAN(ALP1 '()*+,')                                       00001800
        IDENT = ALPHA                                                    00001900
        NUMBR = SPAN(NUM1)                                               00002000
        ALNUM = SPAN(ALP1 NUM1)                                          00002100
        LETTR = ANY(ALP1)                                                00002200
        BLANKX = SPAN(' ')                                               00002300
        PSPAN = SPAN(ALP1 NUM1 '-_')                                     00002400
        ALPHA1 = SPAN(ALP1 ' ')                                          00002500
       NONTERM = (NUL | '$') ALPHA1                                      00002600
        BNONTERM = '<' NONTERM '>'                                       00002700
*ATTREFX MATCHES ATTRIBUTE REFERENCES                                    00002800
        ATTREFX = ANY(ALP1) ' ' ALPHA                                    00002900
        COPYPAT = '*' ATTREFX '*' | ATTREFX                              00003000
        ATTPAT = 'NUL' | 'ATT.' NUMBR                                    00003100
*SEMEQX MATCHES SEMANTIC EQUATIONS                                       00003200
        SEMEQX = BREAK(';')                                              00003300
*                                                                        00003400
        DEFINE('INCR(P1)')                                               00003500
        DEFINE('NEWPAGE(X)')                                             00003600
        DEFINE('DISPLAY(P,P2,LIST)')
        DEFINE('PRINTLONG(LINE,P2)X,Y,Z')                                00003800
        DEFINE('GETRULE()')                                              00003900
        DEFINE('PARSE(NON,P2,REST)SAV1,SAV3,VAL,PR,TERM')                00004000
        DEFINE('PUSH(ELEM)')                                             00004100
        DEFINE('POP()')                                                  00004200
        DEFINE('SEMPASS(N)LHS,PR,NXTSON')                                00004300
        DEFINE('FINDVAL(ATT,N)PROD,I,ATTRY,X')                           00004400
        DEFINE('SELECT(P1,P2,P3,P4)')                                    00004500
        DEFINE('UNO(X)')                                                 00004600
        DEFINE('U0(P1,P2)')                                              00004700
        DEFINE('U0(P1,P2)')                                              00004800
        DEFINE('U(P1,P2,P3,P4,P5,P6,P7,P8,P9,P10)')                      00004900
        DEFINE('U2(P1,P2)')                                              00005000
        DEFINE('U1(P1,P2,P3)X')                                          00005100
        DEFINE('COMPOS(P1,P2)')                                          00005200
        DEFINE('SEL(P1,P2)')                                             00005300
        DEFINE('ELEM(P1,P2)')                                            00005400
        DEFINE('MAKESUB(P1,P2,P3,P4)')                                   00005500
        DEFINE('COND(P1,P2,P3)')                                         00005600
```

```
          DEFINE('EQU(P1,P2)')                                                  00005700
          DEFINE('POSI(P1,P2)')                                                 00005800
          DEFINE('LEQ(P1,P2)')                                                  00005900
          DEFINE('NONZ(P1,P2)')                                                 00006000
          DEFINE('LTN(P1,P2)')                                                  00006100
          DEFINE('GTN(P1,P2)')                                                  00006200
          DEFINE('TRU(P1,P2)')                                                  00006300
          DEFINE('AND(P1,P2)')                                                  00006400
          DEFINE('ID(P1)')                                                      00006500
          DEFINE('SQ(P1)')                                                      00006600
          DEFINE('LNX(X)')                                                      00006700
          DEFINE('IDEN(P1,P2)')                                                 00006800
          DEFINE('OR(P1,P2)')                                                   00006900
          DEFINE('MKFLC(P1,P2,P3)')                                             00007000
          DEFINE('PRINT(P1,S)')                                                 00007100
          DEFINE('APPLI(FLC,S)FUN,NXT,P,T')                                     00007200
          DEFINE('APPLIF(P1,S)')                                                00007300
          DEFINE('TYPE(X)')                                                     00007400
          DEFINE('P1K(P1,P2,P3)')                                               00007500
          DEFINE('P2K(P1,P2,P3)')                                               00007600
          DEFINE('P3K(P1,P2,P3)')                                               00007700
          DEFINE('PARTITION(NP,C,N,L)X')                                        00007800
          COORD = ARRAY('20')                                                   00007900
          DEFINE('EQQ(P1,P2)')                                                  00008000
          DEFINE('SUBSTR(P1,P2,P3)')                                            00008100
          DEFINE('PLIM(NP,L)X')                                                 00008200
          DEFINE('PRINTX(P1,P2,P3)')                                            00008300
          DEFINE('READX(P1,P2)')                                                00008400
          DEFINE('MKCON(P1)')                                                   00008500
           DEFINE('PROJECT(P1,P2)')                                             00008600
          B = '      '                                                          00008700
          SPACESX = '                                                          00008800
*HERE WE SET UP SOME STRING CONSTANTS FOR USE IN KMF DESCRIPTOON                 00008900
          ENVL = 'ENV-'                                                         00009000
          PLISTL = 'PLIST-'                                                     00009100
          VARL = 'VAR-'                                                         00009200
          VALL = 'VAL-'                                                         00009300
          VAR = 'VAR'                                                           00009400
          NAME = 'NAME'                                                         00009500
          FLOWCHART = 'FLOWCHART'                                               00009600
          TYPEL = 'TYPE-'                                                       00009700
          AC = 'AC'                                                             00009800
          &TRIM = 1                                                             00010000
          &ANCHOR = 1                                                           00010100
          DEFINE('L50(L50)')                                                    00010200
          DEFINE('XPARTITION(NP,L,S)')                                          00010300
*                                                                               00010400
          INPUT LEN(1) . DFLG LEN(1) . PFLG LEN(1) . SFLG LEN(1) . TFLG         00010500
. LEN(1) . EFLG
          &DUMP = EQ(DFLG,1) 1                                                  00010600
          NEWPAGE()                                                             00010700
          OUTPUT = B B B B 'MINI-LANGUAGE ' TRIM(INPUT)
          OUTPUT =
          OUTPUT = B B 'SYNTACTIC AND SEMANTIC DESCRIPTION'
          OUTPUT =
          NEXTLINE = INPUT                                                      00010900
          OUTPUT = NEXTLINE                                                     00011000
```

```
PPREAD                                                                  00011100
        IDENT(NEXTLINE,'END')                          :S(PPDONE)       00011200
        LINE = GETRULE()                                                00011300
*SYNTAX RULES ARE NOT INDENTED, CHECK FOR SYNTAX RULE                   00011400
        LINE SPAN(' ') =                               :S(PP1)          00011500
*YES                                                                    00011600
        ALT = 0                                                         00011700
*GET NAME AND PICK UP NEXT ALTERNATIVE                                  00011800
        LINE '<' ALPHA1 . LHS '>'    =                                  00011900
PPNXT   RHS =                                                           00012000
PPNXT1  LINE (BREAK(';') ("'" | ';')) . X  =   :F(PPLALT)                00012100
        &ANCHOR = 0                                                     00012200
        X ';' RPOS(0)    =                             :S(PPSUC)        00012300
        &ANCHOR = 1                                                     00012400
        LINE (BREAK("'") "'") . Y  =                   :F(ERROR)        00012500
        RHS = RHS X Y                                  :(PPNXT1)        00012600
PPSUC   RHS = RHS X                                                     00012700
*                                                                       00012800
*STORE PRODUCTION                                                       00012900
        BLN = '     '                                                   00013000
        ALT = ALT + 1                                                   00013100
        $(LHS '.' ALT)  = '<' LHS '>' RHS                               00013200
*COUNT NONTERMINALS ON RHS                                              00013300
        K = 0                                                           00013400
        &ANCHOR = 0                                                     00013500
PPKNT   RHS BNONTERM    =                              :F(PPSTR1)       00013600
        K = K + 1                                        :(PPKNT)       00013700
*STORE KOUNT                                                            00013800
PPSTR1                                                                  00013900
        &ANCHOR = 1                                                     00014000
PPSTR   $(LHS '..' ALT) = K                            :(PPNXT)         00014100
*                                                                       00014200
*                                                                       00014300
*SET UP FOR SEMANTIC RULES                                              00014400
PPLALT  ATT = 0                                                         00014500
        INH = 0                                                         00014600
        SYN = 0                                                         00014700
        $(LHS '.0') = PNODE(ALT,,,)                    :(PPREAD)        00014800
*CHECK FOR INHERITED ATTRIBUTE DECLARATION                              00014900
PP1     LINE '<' ALPHA . X '>' ANY(ALP1) . Y ';' RPOS(0) = X '-' Y      00015000
+                                                      :F(PPSEM1)       00015100
        ATT = ATT + 1                                                   00015200
        INH = INH + 1                                                   00015300
        $(LHS '.ATT.' ATT) = LINE                                       00015400
        $(LHS '.INH.' INH) = LINE                      :(PPSEM10)       00015500
*                                                                       00015600
*                                                                       00015700
PPSEM1  LINE '<' ALPHA . X '>' LEN(2) . Y ANY(ALP1) . Z  = Y Z          00015800
+                                                 :F(PPSEM2)            00015900
        SYNATT = X '-' Z                                                00016000
        ATT = ATT + 1                                                   00016100
        SYN = SYN + 1                                                   00016200
        $(LHS '.ATT.' ATT) = SYNATT                                     00016300
        $(LHS '.SYN.' SYN) = SYNATT                                     00016400
*                                                                       00016500
*PICK OFF FIRST OR NEXT SEMANTIC EQUATION                               00016600
PPSEM2  LINE BREAK(';') . EQ ';'    =                  :F(PPSEM10)      00016700
```

```
*FIND APPROPRIATE PRODUCTION                                          00016800
          EQ ANY(ALP1) . LET                                          00016900
          I = 0                                                       00017000
PPSEM3    I = I + 1                                                   00017100
          &ANCHOR = 1                                                 00017200
          $(LHS '.' I) NULL                          :S(ERROR2)       00017300
          &ANCHOR = 0                                                 00017400
          $(LHS '.' I) @NONTERM LET                  :F(PPSEM3)       00017500
          &ANCHOR = 1                                                 00017600
*                                                                     00017700
*STORE SEMANTIC EQUATION                                             00017800
          J = 0                                                       00017900
PPSEM4    J = J + 1                                                   00018000
          IDENT(NUL,$(LHS '.SEM.' I '.' J))          :F(PPSEM4)       00018100
          $(LHS '.SEM.' I '.' J) = EQ                :(PPSEM2)        00018200
*FINISHED WITH SEMANTIC RULE, SEE IF NEXT IS SYNTACTIC                00018300
PPSEM10   NEXTLINE SPAN(' ')                         :S(PPREAD)       00018400
*                                                                     00018500
          ATT($(LHS '.0')) = ATT                                      00018600
          INH($(LHS '.0')) = INH                                      00018700
          SYN($(LHS '.0')) = SYN                     :(PPREAD)        00018800
*                                                                     00018900
*                                                                     00019000
```

```
PPDONE                                                                  00019200
        TZ2 = TIME()                                                    00020600
*                                                                       00019300
        START = INPUT                                                   00019400
        STKPTR = 0                                                      00019500
        STACK = ARRAY('100')                                            00019600
        NEWPAGE()                                                       00019700
        OUTPUT = '                    SOURCE PROGRAM'                   00019800
        STR =                                                           00019900
READ001 X = INPUT                                                       00020000
READ002 OUTPUT = X                                                      00020100
        X POS(0) SPAN(' ')     =                                        00020200
        STR = STR X                                                     00020300
        IDENT(X,'END')                          :F(READ001)             00020400
PARSE001                                                                00020500
        TZ2P = TIME()                                                   00020700
        STRING = STR                                                    00020800
        STRING = STR '#'                                                00020900
        SZZZ = SIZE(STRING)                                             00021000
        STR = 0                                                         00021100
        PARSE(START,STR,'*#')                   :F(ERROR)               00021200
        STKPTR = 0 ; TZ3 = TIME()                                       00021300
CALLSEM                                                                 00021400
        SEMPASS(STACK<1>)                                               00021500
        EQ(KEY(STACK<1>),'1')                   :F(CALLSGM)             00021600
        TEMP1 = VALS(STACK<1>)                                          00021700
        NEWPAGE()                                                       00021800
        OUTPUT = 'LISTING OF ATTRIBUTES OF PROGRAM NODE'                00021900
        J = 1                                                           00022000
LIST000                                                                 00022100
        X = 'PROGRAM.ATT.' J                                            00022200
        IDENT(NUL,$X)                           :S(LISTDONE)            00022300
        OUTPUT = ''                                                     00022400
        OUTPUT = ''                                                     00022500
        OUTPUT = B B 'DISPLAY OF ' $X                                   00022600
  TEMP1<J> ('ATT.' *DISPLAY($TEMP1<J>) | *PRINT(TEMP1<J>))              00022700
        J = J + 1                                       :(LIST000)      00022800
LISTDONE                                                                00022900
        NEWPAGE()                                                       00023000
        FLOWLINES = $TEMP1<1>                                           00023100
        PROCLIB = $TEMP1<2>                                             00023200
        IDLIST = PROCLIB                                                00023300
        MFUNLIB = $TEMP1<2>                                             00023400
        EXPLIB = $TEMP1<3>                                              00023500
        ARGLIB = $TEMP1<3>                                              00023600
        LAYOUTLIB = $TEMP1<3>                                           00023700
        DATALIB = $TEMP1<4>                                             00023800
        LABELTABLE = $TEMP1<2>                                          00023900
        VARLIB = $TEMP1<2>                                              00024000
        TRANSLIB = $TEMP1<3>                                            00024100
        CONCATLIB = $TEMP1<4>                                           00024200
        OUTPUT = '          APPLY FLOWCHART TO NULL STATE VECTOR S'     00024300
        TZ4 = TIME()                                                    00024400
        OUTPUT = ''                                                     00024500
        S =                                                             00024600
        STCNT = 0; RXLINE =                                             00024700
        &TRIM = 0
```

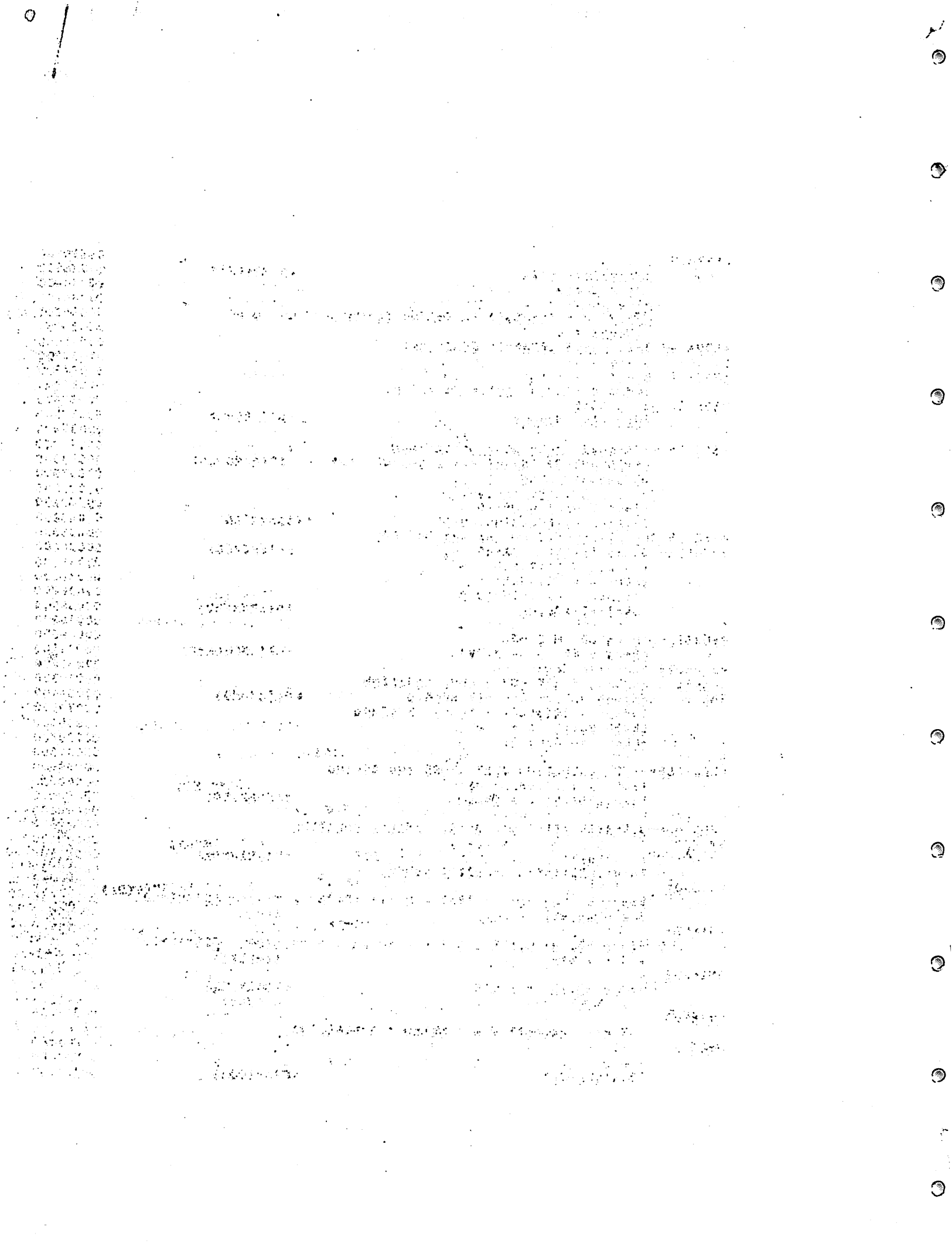
```
          EQ(EFLG,1)      :S(NOEXEC1)
          S = APPLI(FLOWLINES,S)                                                    00024800
NOEXEC1
          OUTPUT = PXLINE ; PXLINE =                                                00024900
          OUTPUT = ''                                                               00025000
          OUTPUT = '          FINAL STATE VECTOR'                                   00025100
          DISPLAY(S)                                                                00025200
          TZ5 = TIME()                                                              00025300
          OUTPUT = ''                                                               00025400
          X = TZ2 - TZ1                                                             00025500
          OUTPUT = 'TIME TO PROCESS SEMANTIC DESCRIPTION ' X ' MILLISEC'            00025600
          X = TZ3 - TZ2P                                                            00025700
          OUTPUT = 'TIME TO PARSE ' X ' MILLISEC'                                   00025800
          X = TZ4 - TZ3                                                             00025900
          OUTPUT = 'TIME TO EVALUATE SEMANTIC ATTRIBUTES ' X ' MILLISEC'            00026000
          X = TZ5 - TZ4                                                             00026100
          OUTPUT = ''EXECUTION' TIME ' X ' MILLISEC'                                00026200
          OUTPUT = ''STATEMENTS' EXECUTED ' STCNT                                   00026300
 STR = ; &TRIM = 1 ; NEWPAGE() ; X = INPUT :F(END)                                  00026400
 OUTPUT = '                    SOURCE PROGRAM' :(READ002)                          00026500
GETRULE                                                                             00026600
          GETRULE = GETRULE NEXTLINE                                                00026700
          NEXTLINE = INPUT                                                          00026800
          NEXTLINE POS(0) '<'      :F(GETRULE1)
          OUTPUT =
GETRULE1
          OUTPUT = NEXTLINE                                                         00026900
          NEXTLINE (NUL | SPAN(' ')) '.'     =      :S(GETRULE)                   00027000
          GETRULE = GETRULE ';'                        :(RETURN)                    00027100
```

```
PARSE    SAV1 = NON                                                    00027300
         SAV3 = REST                                                   00027400
         STRING  LEN(P2) (LEN(10) | RTAB(0)) . X                       00027500
         OUTPUT = EQ(PFLG,1) 'ENTERING PARSE' B NON B X B REST         00027600
         &ANCHOR = 1                                                   00027700
*                                                                      00027800
*IS NON A LITERAL                                                      00027900
         NON '$'     =                                  :F(PRNOTL)    00028000
*SEE IF LITERAL IS IN STRING                                           00028100
         STRING LEN(P2) ($(NON)) . VAL                  :F(FRETURN)   00028200
*LITERAL MATCHES, GATHER TERMINALS IN REST                             00028300
PRLIT1   REST SPACE "'" CHARS . X "'"    =              :F(PRLIT2)    00028400
         TERM = TERM X                                    :(PRLIT1)   00028500
*                                                                      00028600
PRLIT2   STRING LEN(P2 + SIZE(VAL)) TERM                :F(FRETURN)   00028700
*CHECK FOR END OF STRING                                               00028800
         EQ(SZZZ,P2 + SIZE(VAL) + SIZE(TERM))           :S(PRLITP)    00028900
*PICK OFF NEXT NONTERMINAL IS REST                                     00029000
         REST SPACE '<' NONTERM . NON '>' ANY(ALP1)     =              00029100
         STR = P2 + SIZE(VAL) + SIZE(TERM)                            00029200
         PARSE(NON,STR,REST)                            :F(FRETURN)   00029300
*ON SUCESSFUL RETURN FROM PARSE, PUSH A TREEL                          00029400
PRLITP   X = ARRAY('1')                                               00029500
         X<1> = VAL                                                   00029600
         PUSH(TREEL(1,SAV1,0,,1,0,X,))                  :(RETURN)     00029700
*                                                                      00029800
*                                                                      00029900
```

```
*IF NON IS NOT A-LITERAL, START LOOKING AT PRODUCTIONS                  00030100
PRNOTL                                                                  00030200
          PR = 1                                                        00030300
*SET UP RHS                                                            00030400
PRNRHS    RHS = $(NON '.' PR)                                           00030500
          RHS BREAK('=') '='    =                                       00030600
          IDENT(RHS,NUL)                              :S(FRETURN)       00030700
*                                                                       00030800
*SEE IF TERMINALS IN PRODUCTION ARE IN STRING                           00030900
          RHSTERM = ''                                                  00031000
          RHS '''' CHARS . RHSTERM ''''  =            :F(PRCHK)         00031100
          STRING LEN(P2) RHSTERM                      :S(PRCHK)         00031200
*IF THEY DON'T MATCH , TRY NEXT PRODUCTION                              00031300
          PR = PR + 1                                   :(PRNRHS)       00031400
*                                                                       00031500
*MORE GENERALLY ON FAILURE, RESTORE AND TRY NEXT PRODUCTION             00031600
PRNXTP    NON = SAV1                                                    00031700
          REST = SAV3                                                   00031800
          PR = PR + 1                                   :(PRNRHS)       00031900
*                                                                       00032000
*CHECK FOR NONTERMINALS IN RHS                                          00032100
PRCHK     RHS SPACE RPOS(0)                           :F(PRNON)         00032200
*IF NONE, GATHER TERMINALS IN REST                                      00032300
PRTRM     REST SPACE '''' CHARS . X ''''  =           :F(PRMCH)         00032400
          TERM = TERM X                                 :(PRTRM)        00032500
*                                                                       00032600
*SEE IF THEY ARE IN STRING                                              00032700
PRMCH     STRING LEN(P2 + SIZE(RHSTERM)) TERM         :F(PRNXTP)        00032800
*                                                                       00032900
*                                                                       00033000
*CHECK FOR END OF STRING                                                00033100
          EQ(SZZZ,P2 + SIZE(RHSTERM) + SIZE(TERM))    :S(PRSUCC)        00033200
*PICK UP NEXT NONTERMINAL AND CALL PARSE                                00033300
          REST SPACE '<' NONTERM . NON '>' ANY(ALPI)    =     :F(PRNXTP)  00033400
          STR = P2 + SIZE(RHSTERM) + SIZE(TERM)                         00033500
          PARSE(NON,STR,REST)                         :F(PRNXTP) S(PRSUCC)  00033600
*                                                                       00033700
*IF NON HAS NONTERMINALS ON RHS, GET FIRST ONE                          00033800
PRNON                                                                   00033900
          RHS SPACE '<' NONTERM . NON '>' ANY(ALPI)     =     :F(ERROR6)  00034000
          REST = RHS REST                                               00034100
          STR = P2 + SIZE(RHSTERM)                                      00034200
          PARSE(NON,STR,REST)                         :F(PRNXTP)        00034300
*                                                                       00034400
*                                                                       00034500
*ON SUCCESSFUL PARSE,CREATE TREEL                                       00034600
PRSUCC    NVAL = ATT($(SAV1 '.0'))                                      00034700
          NSON = $(SAV1 '..' PR)                                        00034800
*SET UP VALUE AND SONS ARRAYS                                           00034900
          X = ARRAY(NVAL)                                               00035000
          Y =                                                           00035100
          EQ('0',NSON)                                :S(PRPUSH)        00035200
          Y = ARRAY(NSON)                                               00035300
          I = 1                                                         00035400
PRGSONS Y<I> = POP()                                                    00035500
          I = I + 1                                                     00035600
          LE(I,NSON)                                  :S(PRGSONS)       00035700
```

```
PRPUSH   PUSH(TREEL(,SAV1,PR,$(SAV1 '.' PR),NVAL,NSON,X,Y))    :(RETURN)00035800
PUSH     STKPTR = STKPTR + 1                                              00035900
         STACK<STKPTR> = ELEM                          :(RETURN)          00036000
POP      POP = STACK<STKPTR>                                              00036100
         STKPTR = STKPTR - 1                             :(RETURN)        00036200
```

```
SEMPASS                                                                 00036400
          EQ(KEY(N),'1')                                :S(RETURN)      00036500
          LHS = LHS(N)                                                  00036600
          PR = PR(N)                                                    00036700
          OUTPUT = EQ(SFLG,1) 'ENTERING SEMPASS' B LHS B PR             00036800
          &ANCHOR = 0                                                   00036900
*LOOK AT THIS NODES SEMANTIC EQUATIONS                                  00037000
SMBEGN    K = 0                                                         00037100
SMNXTEQ   K = K + 1                                                     00037200
          SEMEQ = $(LHS '.SEM.' PR '.' K)                               00037300
*SEE IF WE'RE DONE                                                      00037400
          IDENT(SEMEQ,NUL)                              :S(SMDONE)      00037500
*                                                                       00037600
*SEE IF ATTRIBUTE VALUE ALREADY DEFINED                                 00037700
          SEMEQ POS(0) BREAK('=') . LHSRUL  '='  =  :F(ERROR20)         00037800
          SEMEQ00 = SEMEQ                                               00037900
          V0 = FINDVAL(LHSRUL,N)                                        00038000
          TEMP1 = VALS(NODE(V0))                                        00038100
          IDENT(NUL,TEMP1<INX(V0)>)                     :F(SMNXTEQ)     00038200
*SEE IF ALL ATTRIBUTES ON RHS ARE DEFINED                               00038300
SMFNDREF  SEMEQ ATTREFX . XREF                          :F(SMEXEC)      00038400
          V = FINDVAL(XREF,N)                                           00038500
          TEMP1 = VALS(NODE(V))                                         00038600
          ATTVAL = TEMP1<INX(V)>                                        00038700
          IDENT(ATTVAL,NUL)                             :S(SMNXTEQ)     00038800
*                                                                       00038900
*SUBSTITUTE VALUE IN SEMEQ                                              00039000
          SEMEQ XREF      = ATTVAL                        :(SMFNDREF)   00039100
*EXECUTE SEMANTIC EQUATIONS                                             00039200
*HERES A KLUDGE, CHECK FOR A COPY EQUATION                              00039300
SMEXEC    SEMEQ00 POS(0) COPYPAT RPOS(0)                :F(SMIMMD)      00039400
          OUTPUT = EQ(SFLG,1) 'COPY' B SEMEQ                            00039500
          SEMEQ POS(0) "'"   =                                          00039600
          SEMEQ "'" RPOS(0)    =                                        00039700
*                                                                       00039800
*FOR DEFERRED EVALUATION JUST STORE THE STRING                          00039900
          TEMP1 = VALS(NODE(V0))                                        00040000
          TEMP1<INX(V0)> = SEMEQ                        :(SMNXTEQ)      00040100
*                                                                       00040200
*FOR NON-DEFERRED ATTRIBUTE DEFS, EXECUTE EQUATION                      00040300
SMIMMD                                                                  00040400
          EQ(SFLG,1)                                    :F(SMIMMDX)     00040500
          PRINTLONG('HERE WE GO' B SEMEQ)                               00040600
SMIMMDX                                                                 00040700
          SEMEQ POS(0) 'U(' ATTPAT . X ',' ATTPAT . Y ')'  :F(SMIMMD1)  00040800
          V = U($X,$Y)                                    :(HERE)       00040900
SMIMMD1                                                                 00041000
          SEMEQ POS(0) NUMBR . X ' + ' NUMBR . Y RPOS(0)  :F(SMIMMD2)   00041100
          V = X + Y                                       :(HERE)       00041200
SMIMMD2                                                                 00041300
          SEMEQ POS(0) "'NUL'"                          :F(SMIMMD3)     00041400
          V = 'NUL'                                       :(HERE)       00041500
SMIMMD3                                                                 00041600
                    :<CODE(' V = ' SEMEQ ' :(HERE)')>                   00041700
HERE                                                                    00041800
          X = V                                                         00041900
          IDENT(V,NUL)                                  :F(SM0001)      00042000
```



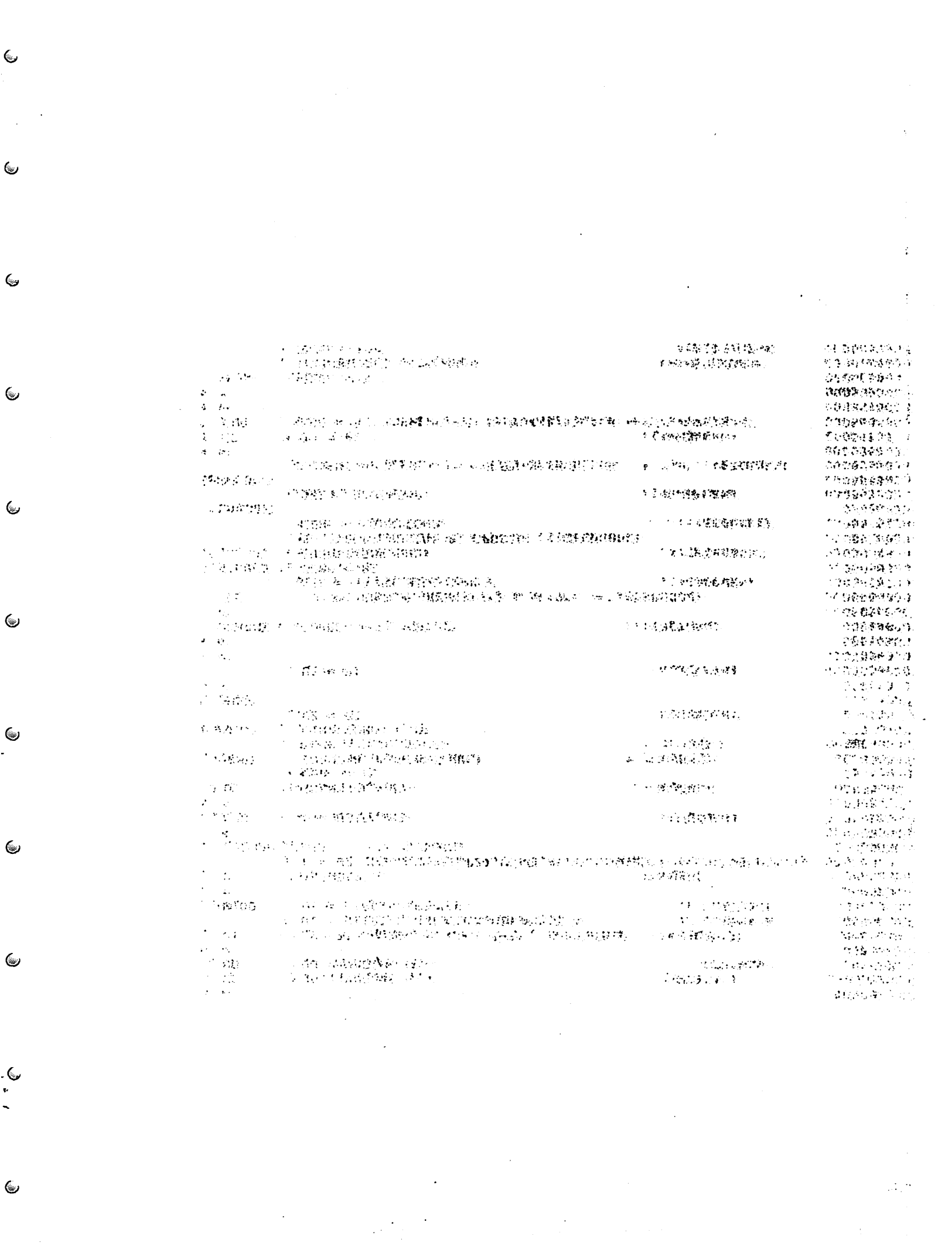
```
          X = 'NUL'                                                          00042100
                                                                             00042200
SM0001
          IDENT(TYPE(V),'STRING')                    :S(SMSRES)              00042300
          IDENT(TYPE(V),'INTEGER')                   :S(SMSRES)              00042400
          X = 'ATT.' UNO()                                                   00042500
          $X = V                                                             00042600
SMSRES                                                                       00042700
          TEMP1 = VALS(NODE(V0))                                             00042800
          TEMP1<INX(V0)> = X                          :(SMNXTEQ)             00042900
*                                                                            00043000
*                                                                            00043100
*CALL SEMPASS FOR SONS                                                       00043200
SMDONE                                                                       00043300
SMDOSON   IDENT(NXTSON,NUL)                           :F(SMKEYCHK)           00043400
          NXTSON = 0                                                         00043500
SMNXTSON  NXTSON = NXTSON + 1                                                00043600
          GT(NXTSON,NSON(N))                          :S(SMBEGN)             00043700
          TEMP1 = SONS(N)                                                    00043800
          SEMPASS(TEMP1<NXTSON>)                      :(SMNXTSON)            00043900
*                                                                            00044000
*SEE IF WE CAN SET KEY FOR N                                                 00044100
SMKEYCHK  I = 0                                                              00044200
SMKEY1    I = I + 1                                                          00044300
          GT(I,NVAL(N))                               :S(SMKEY2)             00044400
          TEMP1 = VALS(N)                                                    00044500
          IDENT(NUL,TEMP1<I>)                         :S(RETURN) F(SMKEY1)   00044600
*                                                                            00044700
SMKEY2    EQ(NSON(N),'0')                             :S(SMRET)              00044800
          I = 0                                                              00044900
SMKEY3    I = I + 1                                                          00045000
          GT(I,NSON(N))                               :S(SMRET)              00045100
          TEMP1 = SONS(N)                                                    00045200
          IDENT(KEY(TEMP1<I>),NUL)                    :S(RETURN) F(SMKEY3)   00045300
*                                                                            00045400
SMRET     KEY(N) = 1                                  :(RETURN)              00045500
FINDVAL                                                                      00045600
          PROD = $(LHS(N) '.' PR(N))                                         00045700
*                                                                            00045800
*DETERMINE WHICH NONTERMINAL WE'RE AFTER                                     00045900
          I = 0                                                              00046000
          ATT ANY(ALP1) . ALET1                       :F(ERROR60)            00046100
FVFNDXXX  PROD BNONTERM ANY(ALP1) . ALET2     =       :F(ERROR61)            00046200
    IDENT(ALET1,ALET2)                              :S(FVFND1)               00046300
          I = I + 1                                   :(FVFNDXXX)            00046400
FVFND1    EQ(I,'0')                                   :S(FVFNDATT)           00046500
          TEMP1 = SONS(N)                                                    00046600
          N = TEMP1<I>                                                       00046700
FVFNDATT                                                                     00046800
          ATT ANY(ALP1) ' '    =                      :F(ERROR23)            00046900
          X = LHS(N)                                                         00047000
          I = 1                                                              00047100
*CHECK FOR LITERAL NODE                                                      00047200
          X POS(0) '$'                                :F(FVFND2)             00047300
          IDENT(ATT,'VAL')                            :S(FVDONE) F(ERROR24)  00047400
FVFND2    GT(I,ATT($(X '.0')))                        :S(ERROR25)            00047500
          $(X '.' ATT '.' I) '=' ATT RPOS(0)    :S(FVDONE)                   00047600
          I = I + 1                                   :(FVFND2)              00047700
```

```
FVDONE                                                                  00047800
          ATL = ATTLOC()                                                00047900
          NODE(ATL) = N                                                 00048000
          INX(ATL) = I                                                  00048100
          FINDVAL = ATL                                  :(RETURN)      00048200
ERROR25   OUTPUT = 'ERROR25' B ATT B PROD B SEMEQ        :(ERROR)       00048300
```

```
PRINTLONG                                                              00048500
          OUTPUT = LINE              :(RETURN)                         00048600
*                                                                      00048700
*                                                                      00048800
DISPLAY                                                                00048900
          IDENT(DATATYPE(P),'VNO')          :S(DISX)
          OUTPUT = P2 P             :(RETURN)
DISX      LIST (POS(0) | '*') SLX(P) '*'    :F(DISO)
          SLX(P) =
DISO      IDENT(LINK(P),)       :S(DISOO)
          DISPLAY(LINK(P),P2,SLX(P) '*' LIST)
DISOO     IDENT(SLX(P),)        :S(DIS2)
          IDENT(DATATYPE(OBJ(P)),'VNO')              :S(DIS1)          00049500
DISOO1    OBJ(P) ' '  = '*'         :S(DISOQ1)                        00049600
          PRINTLONG(SLX(P) B OBJ(P),P2)                  :(DIS2)     00049700
DIS1      OUTPUT = P2 SLX(P) B '----------'                            00049800
          DISPLAY(OBJ(P),P2 '      ')                                 00049900
          OUTPUT = ''                                                 00050000
DIS2                        :(RETURN)
*                                                                      00050400
NEWPAGE   NEWPAGE =                     :(RETURN)                      00050500
INCR      EQ(P1)                    :S(ERROR)                        00050600
          INCR = GT(P1) 1         :S(RETURN)                         00050700
          INCR = 0 - 1          :(RETURN)                            00050800
```

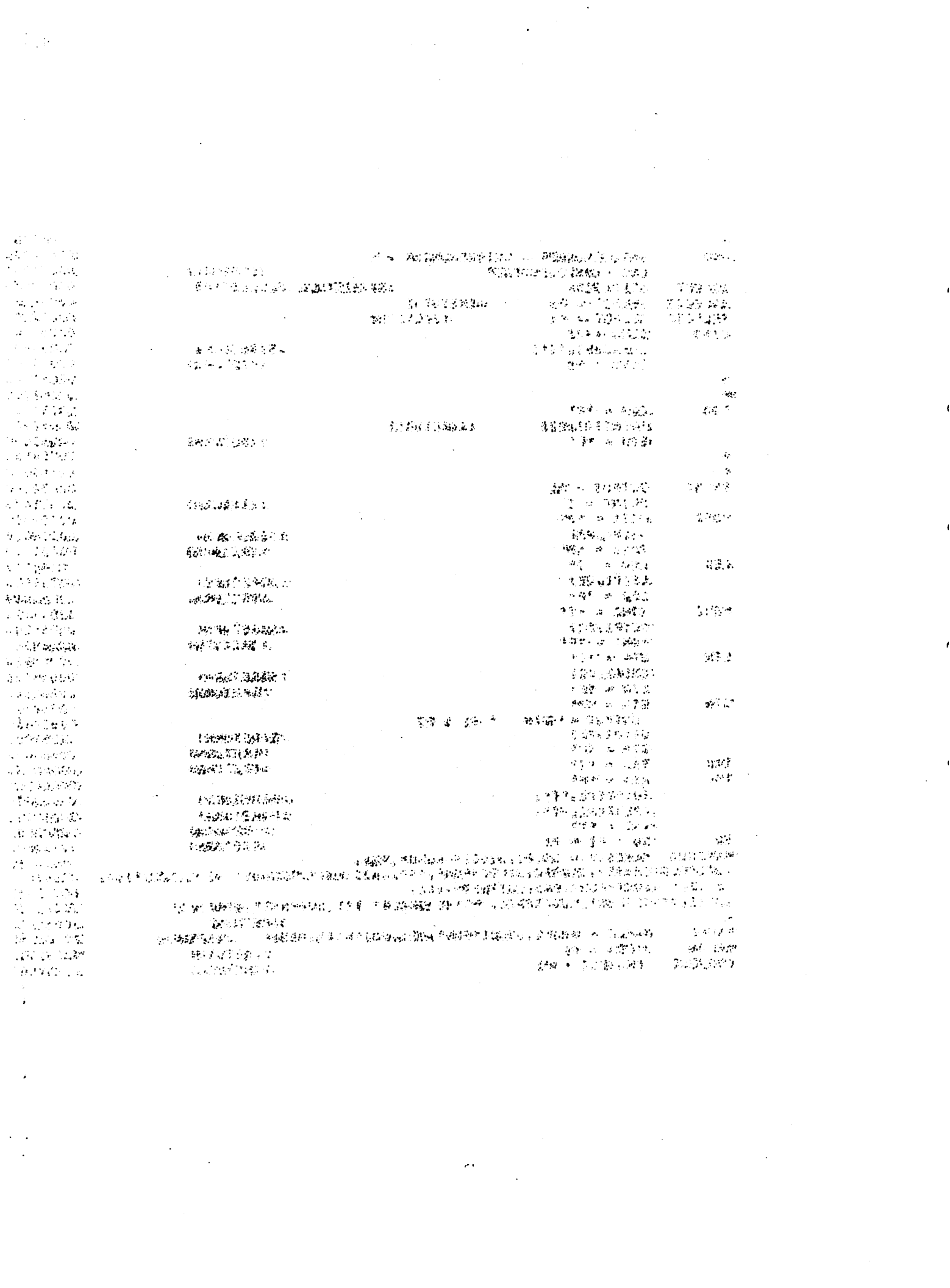
```
LNX     LNX = SIZE(X)                    :(RETURN)          00051000
OR      OR = 'T'                                            00051100
        IDENT(P1,'T')                 :S(RETURN)            00051200
        IDENT(P2,'T')                 :S(RETURN)            00051300
        OR = 'F'                        :(RETURN)           00051400
IDEN    IDEN = 'T'                                          00051500
        IDENT(P1,P2)                  :S(RETURN)            00051600
        IDEN = 'F'                      :(RETURN)           00051700
```

```
*                                                                                00051900
U0      U0 = VNO(P1,P2,)                                  :(RETURN)              00052000
U0      U0 = VNO(P1,P2,)                                     :(RETURN)           00052100
*                                                                                00052200
U1      P2 '?' SPAN(ALP1 NUM1 '-') . X RPOS(0)  =   :F(U100)                     00052300
        U1 = U1(P1,X,U1(SEL(X,P1),P2,P3))                    :(RETURN)           00052400
U100    U1 = U(P1,U0(P2,P3))                                 :(RETURN)           00052500
*                                                                                00052600
U       IDENT(P3,)                                        :S(UUU)                00052700
        U = U2(U2(U2(U2(P1,P2),U2(P3,P4)),U2(U2(P5,P6),U2(P7,P8))),              00052800
.U2(P9,P10))        :(RETURN)                                                    00052900
*                                                                                00053000
UUU     U = U2(P1,P2)                                     :(RETURN)              00053100
*                                                                                00053200
U2      IDENT(P2,NUL)                                     :S(U003)               00053300
        TEMP = P2                                                                00053400
U001    IDENT(LINK(TEMP),NUL)                             :S(U002)               00053500
        TEMP = LINK(TEMP)                                    :(U001)             00053600
U002    LINK(TEMP) = P1                                                          00053700
        U2 = P2                                           :(RETURN)              00053800
U003                                                                             00053900
*                                                                                00054000
        U2 = P1                                           :(RETURN)              00054100
*                                                                                00054200
*                                                                                00054300
COMPOS  COMPOS = P1 '?' P2                                :(RETURN)              00054400
*                                                                                00054500
SEL         P1 POS(0) BREAK('?') . X '?'  =   :F(SEL000)                         00054600
        SEL = SEL(X,SEL(P1,P2))                              :(RETURN)           00054700
SEL000  TEMP = P2                                                                00054800
SEL001  IDENT(TEMP,NUL)                                   :S(SEL003)             00054900
        SLX(TEMP) POS(0) P1 RPOS(0)   :S(SEL002)                                 00055000
        TEMP = LINK(TEMP)                                    :(SEL001)           00055100
SEL002                                                                           00055200
        SEL = OBJ(TEMP)                                   :(RETURN)              00055300
SEL003                                                                           00055400
        OUTPUT = EQ(TFLG,1) 'FRETURN FROM SEL     ' P1  :(RETURN)                00055500
*                                                                                00055600
ID      ID = P1                                           :(RETURN)              00055700
EQQ     EQQ = 'T' ; EQ(P1,P2) :S(RETURN); EQQ = 'F' :(RETURN)                    00055800
*                                                                                00055900
*                                                                                00056000
ELEM    ELEM = 'T'                                                               00056100
        IDENT(SEL(P1,P2),NUL)                             :F(RETURN)             00056200
        ELEM = 'F'                                           :(RETURN)           00056300
```



```
L50     L50 LEN(50) . L50           :(RETURN)                    00056500
```

```
*                                                                       00056700
UNO     UNIQUENUMBER = UNIQUENUMBER + 1                                 00056800
        UNO = UNIQUENUMBER                                  :(RETURN)   00056900
SELECT  GT(P1,P2)                       :S(SELECTS) F(SELECTF)          00057000
SELECTS SELECT = P3         :(RETURN)                                   00057100
SELECTF SELECT = P4             :(RETURN)                               00057200
COND    COND = P2                                                       00057300
        IDENT(P1,'T')                                   :S(RETURN)      00057400
        COND = P3                                         :(RETURN)     00057500
*                                                                       00057600
*                                                                       00057700
EQU     EQU = 'T'                                                       00057800
        IDENT(P1,P2)        :S(RETURN)                                  00057900
        EQU = 'F'                                         :(RETURN)     00058000
*                                                                       00058100
*                                                                       00058200
PRINT   OUTPUT = P1                                                     00058300
        PRINT = S                                         :(RETURN)     00058400
POSI    POSI = 'T'                                                      00058500
        GE(P1,P2)                                       :S(RETURN)      00058600
        POSI = 'F'                                        :(RETURN)     00058700
LEQ     LEQ = 'T'                                                       00058800
        LE(P1,P2)                                       :S(RETURN)      00058900
        LEQ = 'F'                                         :(RETURN)     00059000
NONZ    NONZ = 'T'                                                      00059100
        NE(P1,P2)                                       :S(RETURN)      00059200
        NONZ = 'F'                                        :(RETURN)     00059300
LTN     LTN = 'T'                                                       00059400
        LT(P1,P2)                                       :S(RETURN)      00059500
        LTN = 'F'                                         :(RETURN)     00059600
GTN     GTN = 'T'                                                       00059700
         OUTPUT = 'GTN    ' P1 B P2                                     00059800
        GT(P1,P2)                                       :S(RETURN)      00059900
        GTN = 'F'                                         :(RETURN)     00060000
TRU     TRU = 'T'                                         :(RETURN)     00060100
AND     AND = 'F'                                                       00060200
        IDENT(P1,'T')                                   :F(RETURN)      00060300
        IDENT(P2,'T')                                   :F(RETURN)      00060400
        AND = 'T'                                         :(RETURN)     00060500
SQ      SQ = P1 * P1                                      :(RETURN)     00060600
MAKESUB  MAKESUB = UO(P1,U(UO('ARGNO',P2),                              00060700
.UO('FLOWCHART',U(UO(0,U(UO('FUN','S'),UO('NXT','COND(' P3 ',1,2)')))),  00060800
.U(UO(1,U(UO('FUN',P4),UO('NXT',))))),                                  00060900
.UO(2,U(UO('FUN','UO(TYPEL, "TYPE ERROR" )'),UO('NXT',)))))))))        00061000
*                                                                       00061100
MKFLC    MKFLC = UO(P1,U(UO('FUN',P2),UO('NXT',P3)))    :(RETURN)       00061200
MKCON    MKCON = P1                                       :(RETURN)     00061300
PROJECT   PROJECT = P1                                    :(RETURN)     00061400
```



```
*THESE FUNCTIONS ARE USED IN MINI-LANGUAGE 9                                    00061600
PARTITION                                                                       00061700
          GLOBALN = 0                                                           00061800
          XPARTITION(NP,L,1)     :F(FRETURN)                                    00061900
          X = 1                                                                 00062000
          PARTITION = 1                                                         00062100
PLOOPP    GT(X,C / 2)     :S(RETURN)                                            00062200
          PARTITION = PARTITION + COORD<X>                                      00062300
          X = X + 1       :(PLOOPP)                                             00062400
XPARTITION                                                                      00062500
          GT(NP,L)      :S(FRETURN)                                           00062600
          EQ(NP,1)       :F(GOINCR)                                           00062700
          GLOBALN = GLOBALN + 1                                                 00062800
          EQ(N,GLOBALN)     :F(FRETURN)                                         00062900
          COORD<S> = L        :(RETURN)                                         00063000
GOINCR                                                                          00063100
          COORD<S> = 1                                                          00063200
LOOPXP    XPARTITION(NP - 1,L - COORD<S>,S + 1)  :S(RETURN)                     00063300
          COORD<S> = COORD<S> + 1                                               00063400
          GT(NP - 1,L - COORD<S>)    :S(FRETURN) F(LOOPXP)                      00063500
PLIM                                                                            00063600
          GLOBALN = 0; PLIM = PLIM + 1  :(PLIMO)                                00063700
          PLIM = 0                                                              00063800
PLIMO     PARTITION(NP,0,PLIM + 1,L)      :F(RETURN)                          00063900
          PLIM = PLIM + 1                                :(PLIMO)               00064000
PLIMOOO    GLOBALN = 0                                   :(RETURN)             00064100
SUBSTR                                                                          00064200
   OUTPUT = EQ(TFLG,1) 'SUBSTR      ' P1 B P2 B P3                              00064300
          LE(P2,0)                        :S(RETURN)                            00064400
          P3 POS(P1 - 1) LEN(P2 - P1) . SUBSTR  :S(RETURN)                      00064500
ER7       OUTPUT = 'ER7    ' P1 B P2 B P3              :(ERROR7)                00064600
P1K P1K = IDEN(P1,SUBSTR(PARTITION(P2,2 * P3 - 1,                               00064700
+SEL("N",S),LNX(SEL("STR",S))),PARTITION(P2,2 * P3,SEL("N",S),                  00064800
+LNX(SEL("STR",S))),SEL("STR",S)))      :(RETURN)                              00064900
P2K P2K = IDEN(SUBSTR(PARTITION(P2,2 * P3 - 1,SEL("N",S),LNX(SEL("STR",S00065000
+))),PARTITION(P2,2 * P3,SEL("N",S),LNX(SEL("STR",S))),SEL("STR",S)),   00065100
+SUBSTR(PARTITION(P2,2 * P1 - 1,SEL("N",S),                                      00065200
+LNX(SEL("STR",S))),PARTITION(P2,2 * P1,SEL("N",S),                              00065300
+LNX(SEL("STR",S))),SEL("STR",S)))      :(RETURN)                              00065400
P3K P3K = APPLI(SEL(P1,VARLIB ),U1(S,"STR",SUBSTR(PARTITION(P2,2 * P3   00065500
+ - 1,SEL("N",S),LNX(SEL("STR",S))),PARTITION(P2,2 * P3,SEL("N",S),      00065600
+LNX(SEL("STR",S))),SEL("STR",S))))     :(RETURN)                              00065700
```

```
*THESE ROUTINES REALIZE THE I/O FUNCTIONS IN MINI-LANGUAGE 10          00065900
READX                                                                  00066000
*                                                                      00066100
        IDENT(RXLINE,NUL)                              :F(READX01)     00066200
        RXLINE = INPUT                                                 00066300
*                                                                      00066400
*                                                                      00066500
READX01 IDENT(P2,'L')                                  :F(READX03)     00066600
        RXLINE = INPUT                                                 00066700
READX02 RXLINE POS(0) LEN(P1) =                                        00066800
        OUTPUT = 'READ(' P1 ',' P2 ')'          :(RETURN)              00066900
READX03 IDENT(P2,'B')                                  :S(READX02)     00067000
*                                                                      00067100
        RXLINE POS(0) LEN(P1) . READX  =                               00067200
        READX POS(0) SPAN(' ')   =                                     00067300
        OUTPUT = 'READ(' P1 ',' P2 ') RETURNS ' READX   :(RETURN)      00067400
*                                                                      00067500
PRINTX  IDENT(P2,'L')                                  :F(PRINTX03)    00067600
        OUTPUT = PXLINE                                                00067700
        PXLINE =                                                       00067800
PRINTX01 X = DUPL(' ',P1)                                              00067900
PRINTX02 PXLINE = PXLINE X                               :(RETURN)     00068000
*                                                                      00068100
PRINTX03 IDENT(P2,'B')                                  :S(PRINTX01)   00068200
        (SPACESX P3)  LEN(P1) . X RPOS(0)              :(PRINTX02)     00068300
APPLI   P = FLC                                                        00068400
        IDENT(DATATYPE(P),'VNO')                        :S(APPLY0)     00068500
        APPLI = APPLIF(P,S)                              :(RETURN)     00068600
APPLY0                                                                 00068700
        IDENT(LINK(P),NUL)                             :S(APPLY00)     00068800
        P = LINK(P)                                      :(APPLY0)     00068900
APPLY00 P = OBJ(P)                                                     00069000
APPLY1                                                                 00069100
        T = S                                                          00069200
        FUN = OBJ(LINK(P))                                             00069300
        NXT = OBJ(P)                                                   00069400
*                                                                      00069500
   OUTPUT = EQ(TFLG,1) L50(FUN) DUPL(' ',55 - SIZE(L50(FUN))) L50(NXT) 00069600
        STCNT = STCNT + 1                                              00069700
                :<CODE(' S = ' FUN ' :(HERE1)')>                       00069800
HERE1                                                                  00069900
        APPLI = S                                                      00070000
            :<CODE(' NXT = ' NXT ' :(HERE2)')>                         00070100
HERE2                                                                  00070200
        IDENT(NXT,NUL)                                  :S(RETURN)     00070300
        P = SEL(NXT,FLC)                                 :(APPLY1)     00070400
APPLIF      :<CODE(' APPLIF = ' P1 ' :(RETURN)')>                      00070500
TYPE    TYPE = DATATYPE(X)                              :(RETURN)      00070600
END                                                                    00070700
```

APPENDIX III

AN EASILY READABLE FORMAT FOR LANGUAGE DEFINITIONS

APPENDIX III

When the definitions for the ten mini-languages were formulated, a number of decisions were made which involve a trade-off between readability and ease of implementation (i.e., in conjunction with the semantics-based interpreter). A number of these decisions were resolved in favor of ease of implementation, and unquestionably there has been some toll on the readability of the definitions. Also, the limited character set of typical line printers resulted in some notational constructs that are less than ideal. In this Appendix, the definitions of mini-languages 1 and 4 will be re-written and all efforts will be made to give easily readable definitions.

The definitions will be identical to the ones given in Sections 4.1.3 and 4.4.3, except for the following changes:

1) Vienna Definition Language notation will be used for the Vienna operators, thus

UO(X,Y)	is replaced by	<X:Y>
SEL(X↦Y,Z)	"	X↦Y(Z)
U(X,Y)	"	μ(X,Y)
U1(X,S,Y)	"	μ(X,<S:Y>)

2) The letter associated with each attribute will be the first letter in the name of the attribute. Trivial attribute equations of the form A↑X=B↑X, when there is only one non-terminal on the right-hand side of the production having the attribute referenced by X, will be omitted. Inherited attributes will be indicated by "-I" following the attribute name.

3) To eliminate an overabundance of quotes, strings will represent themselves. Functions that are to be evaluated when the semantic equation is evaluated will be underlined.

4) The VO representation of the FLOWCHART attribute will be eliminated in favor of writing the flowchart as a set of recursive functional equations represented by strings; the name of this attribute will be changed to FUNCTIONALS. Function variables will have the form F[name]. In mini-language 4, name may be a state vector function; in this case, F[name] represents the function

IF(name=η_1 THEN F[η_1] ELSE (IF name=η_2 THEN F[η_2]...
ELSE F[η_n]))...)

where MFUNLIB has the form

{F[η_1]=α_1, F[η_2]=α_2,...,F[η_n]=α_n}

5) The function COND(α,β,γ) will be replaced by "if α then β else γ".

6) In the new definition of mini-language 4, we will assume the existence of an appropriately defined primitive function SELECT (see Section 4.4.2). This eliminates the need for the EXPLIB attribute. Also, UNO(S) will be assumed to be a monotonic function of S.

Mini-Language 1 - Syntactic and Semantic Description

```
<PROGRAM>A=<BLOCK>B
          <FUNCTIONALS>
          <SYMBOLTABLE>
          B↑N=η;B↑G=η


<BLOCK>A='BEGIN ' <DECLARATION>B ';' <COMMAND LIST>C ';END'
          <NEXT>-I
          <GLOBALS>-I
          <VARTABLE>A↑V=μ(A↑G,B↑V)
          <FUNCTIONALS>
          <STATENO>


<DECLARATION>A='IDEN ' <VARLIST>B
          <VARTABLE>


<VARLIST>A=<$LETTER>B  ',' <VARLIST>C;D=<$LETTER>E
          <VARTABLE>A↑V=μ(<B↑VAL:UNO()>,C↑V);
                    .D↑V=<E↑VAL:UNO()>


<COMMAND LIST>A=<COMMAND>B ';' <COMMAND LIST>C;D=<COMMAND>E
          <NEXT>-I
          <VARTABLE>-I
          <LABELTABLE>-I
          <XLABELS>A↑X=μ(B↑X,C↑X)
          <FUNCTIONALS>A↑F=μ(B↑F,C↑F)
          <STATENO>


<COMMAND>=<LABEL>B ' ' <UNLABELLED COMMAND>C;D=<UNLABELLED COMMAND>E
          <NEXT>-I
          <VARTABLE>-I
          <LABELTABLE>-I
          <XLABELS>A↑X=<B↑V:C↑S>;
                    .D↑X=η
          <FUNCTIONALS>
          <STATENO>


<LABEL>A='L' <$NUMBER>
          <VALUE>A↑V=L B↑VAL
```

SOURCE PROGRAM

```
BEGIN IDEN A,B;
   A=1;
   B=2;
   BEGIN IDEN B,C;
      B=A;
      C=7;
   END;
   HOPTO L1;
   A=3;
L1 B=3;
END
```

LISTING OF ATTRIBUTES OF PROGRAM NODE

DISPLAY OF FUNCTIONALS

F[#1]=F[#2](μ(S,<#A:1>))

F[#2]=F[#3](μ(S,<#B:2>))

F[#3]=F[#4](μ(S,<#B2:#A(S)>))

F[#4]=F[#5](μ(S,<#C2:7>))

F[#5]=F[#7]

F[#6]=F[#7](μ(S,<#A:3>))

F[#7]=μ(S,<#B:3>)

DISPLAY OF SYMBOLTABLE

A #A
B #B

Mini-Language 4 - Syntactic and Semantic Description

```
<PROGRAM>A=<EXPRESSION>B
          <FUNCTIONAL>A↑F=F[PROGRAM]=B↑F
          <MFUNLIB>
          B↑D=η


<EXPRESSION>A=<$NUMBER>B ;C=<LET EXPRESSION>D;E=<COMBINATION>F;G=<IDENTIFIER>H
          <DECLNAMES>-I
          <MFUNLIB>A↑M=η;G↑M=η
          <FUNCTION>A↑F=B↑VAL


<IDENTIFIER>G=<$ALPHA>H
          <FUNCTION>G↑F=if TYPE→H↑VAL(G↑D)=FUN then μ(<NAME:H↑VAL>,<ENV:ENV(S)>)
                        else if TYPE→H↑VAL(G↑D)=EXP then F[H↑VAL]
                        else NO→H↑VAL(G↑D)→(FNAME→H↑VAL(G↑D)→ENV(S))(S)


<LET EXPRESSION>A='LET ' <$ALPHA>B '=' <EXPRESSION>C ' IN ' <EXPRESSION>D;
.E='LET ' <$ALPHA>F '(' <PARAM LIST>G ')=' <EXPRESSION>G ' IN ' <EXPRESSION>I
          <DECLNAMES>-I
          <FUNCTION>A↑F=D↑F;E↑F=I↑F
          <MFUNLIB>A↑M=μ(F[B↑VAL]=C↑F,C↑M,D↑M);
                   .E↑M=μ(F[F↑VAL]=H↑F,H↑M,I↑M)
          C↑D=μ(A↑D,<B↑VAL:<TYPE:EXP>>)
          H↑D=μ(E↑D,<F↑VAL:<TYPE:FUN>>)
          G↑I=1;G↑N=F↑VAL


<PARAM LIST>A=<$ALPHA>B ',' <PARAM LIST>C;D=<$ALPHA>E
          <NAME>-I
          <INDEX>-I
          <DECLNAMES>A↑D=μ(C↑D,<B↑VAL:μ(<TYPE:PARAM>,<NO:A↑I>,<FNAME:A↑N>));
                    .D↑D=<E↑VAL:μ(<TYPE:PARAM>,<NO:D↑I>,<FNAME:D↑N>))
          C↑I=A↑I + 1


<COMBINATION>A='(' <EXPRESSION>B '+' <EXPRESSION>C ')';
.D='IF ' <EXPRESSION>E ' ' <EXPRESSION>F ' THEN ' <EXPRESSION>G
. ' ELSE ' <EXPRESSION>H ')';
.L=<IDENTIFIER>M '(' <EXPRESSION LIST>N ')'
          <DECLNAMES>-I
          <MFUNLIB>A↑M=μ(B↑M,C↑M);
                  .D↑M=μ(E↑M,F↑M,G↑M,H↑M)
          <FUNCTION>A↑F=B↑F + C↑F;
                    .D↑F=SELECT(E↑F,F↑F,G↑F,H↑F);
                    .L↑F=F[NAME(M↑F)](μ(S,<ENV:μ(ENV(S),<NAME(M↑F):UNO(S)>)>,
                                                          <UNO(S):N↑F>))
          N↑I=1
```

APPENDIX III

When the definitions for the ten mini-languages were formulated, a number of decisions were made which involve a trade-off between readability and ease of implementation (i.e., in conjunction with the semantics-based interpreter). A number of these decisions were resolved in favor of ease of implementation, and unquestionably there has been some toll on the readability of the definitions. Also, the limited character set of typical line printers resulted in some notational constructs that are less than ideal. In this Appendix, the definitions of mini-languages 1 and 4 will be re-written and all efforts will be made to give easily readable definitions.

The definitions will be identical to the ones given in Sections 4.1.3 and 4.4.3, except for the following changes:

1) Vienna Definition Language notation will be used for the Vienna operators, thus

U0(X,Y)	is replaced by	<X:Y>
SEL(X→Y,Z)	"	X→Y(Z)
U(X,Y)	"	μ(X,Y)
U1(X,S,Y)	"	μ(X,<S:Y>)

2) The letter associated with each attribute will be the first letter in the name of the attribute. Trivial attribute equations of the form A↑X=B↑X, when there is only one non-terminal on the right-hand side of the production having the attribute referenced by X, will be omitted. Inherited attributes will be indicated by "-I" following the attribute name.

```
<EXPRESSION LIST>A=<EXPRESSION>B ',' <EXPRESSION LIST>C;D=<EXPRESSION>E
          <DECLNAMES>-I
          <INDEX>-I
          <MFUNLIB>A↑M=μ(B↑M,C↑M)
          <FUNCTION>A↑F=μ(<A↑I:B↑F>,C↑F);D↑F=<D↑I:E↑F>
          C↑I=A↑I + 1
```

SOURCE PROGRAM

```
LET G(X,P)=(IF X>3 THEN X ELSE (X+P((X+1)))) IN G(2,G)
```

LISTING OF ATTRIBUTES OF PROGRAM NODE

DISPLAY OF FUNCTIONAL

```
F[PROGRAM]=F[NAME(μ(<NAME:G>,<ENV:ENV(S)>))](μ(S,<ENV:μ(ENV(μ(<NAME:G
  <ENV:ENV(S)>)),<NAME(μ(<NAME:G>,<ENV:ENV(S)>)):UNO(S)>)>,<UNO(S):μ(
  <2:μ(<NAME:G>,<ENV:ENV(S)>)>))
```

Note: this simplifies to

```
F[PROGRAM]=F[G](μ(S,<ENV:μ(ENV(S),<G:UNO(S)>)>,<UNO(S):μ(<1:2>,
  <2:μ(<NAME:G>,<ENV:ENV(S)>)>))
```

DISPLAY OF MFUNLIB

```
F[G]=SELECT(1→(G→ENV(S))(S),3,1→(G→ENV(S))(S),1→(G→ENV(S))(S) +
  F[NAME(2→(G→ENV(S))(S)](μ(S,<ENV:μ(ENV(2→(G→ENV(S))(S)),
  <NAME(2→(G→ENV(S))(S)):UNO(S)>),<UNO(S):μ(<1:1→(G→ENV(S))(S) + 1>,
  <2:2→(G→ENV(S))(S)>)>))
```