#75

Research Publication GMR-2297

THE DEFINITION MECHANISM FOR STANDARD PL/I

Michael Marcotty*
Computer Science Department
Research Laboratories
General Motors Corporation
Warren, Michigan 48090

Frederick G. Sayward**
Department of Computer Science
Yale University
New Haven, Conneticut 06520

* Part of this work was done while the author was at the University of Massachusetts, Amherst.

** Part of this work was done while the author was at Brown University, Providence, R. I. under the sponsorship of NSF Grant GJ-28074 and part at Yale University under ONR Grant N00014-75-C-0752.

December 1976

Submitted for publication in

IEEE Transactions on Software Engineering

Abstract

The mechanism used to define the programming language PL/I in the recently adopted American National Standard is presented. This method provides a rigorous though semi-formal specification of the language. It uses the model of translation of programs into an abstract form to define the context-free and context-sensitive syntax. The semantics are defined by the interpretation of the abstract form of the program on a hypothetical machine. The method and metalanguage are presented along with several small examples to illustrate the definition technique's features. The complete definition process is shown by the definition of a small example language.

TABLE OF CONTENTS

1. INTRODUCTION

1.1 Prelude

It is all very well for Humpty Dumpty to say "When I use a word, it means just what I choose it to mean" but unless the audience has access to his dictionary, understanding is very difficult. We rely heavily on the meaning of words being constant. For example, when we buy a bottle of aspirins, we count on the effect of the tablets being essentially the same, no matter who made them, advertisement claims notwithstanding.

In the United States, the U. S. Pharmacopia assures the user that the drugs it lists are of standard composition. Although this is defined primarily for the pharmacist and is written in a precise technical language, it is consulted by many sophisticated users who understand its terminology.

The standards for programming languages form an analogous set of definitions. Their existence assures the user that a program written in a standard language can be moved from one implementation to another. However, the definitions are directed principally at the implementor and not the user. Sophisticated users and text-book authors, for example, will also want to read the formal definition of the language to get the final word on the minutiae of the language. The definition is not a tutorial document but must provide a complete and unambiguous specification of the language and this requires a considerable amount of formalism. In general, this formalism has been absent in the past, resulting in disparities between different implementations of the "same" standard language.

Recently, a definition of the programming language PL/I prepared by a Joint Project sponsored by the European Computer Manufacturers' Association (ECMA) and the American National Standards Institute (ANSI) has been adopted as a standard by ANSI. This language is defined in "BASIS/1" [E2] using a semi-formal definition method. Although languages such as BASIC [L3], ALGOL 68 [W4], and indeed PL/I itself [A2, A3, L4, L5] have been defined formally, this is the first time in a standard that both the syntax and semantics of a programming language have been defined with such a degree of rigor.

The purpose of this paper is to present an introduction to the definition method of BASIS/1 by using it to define a small artificial language, SAL, chosen to illustrate the salient features of the technique. The small size of SAL, helped by many examples, permits an overall view of the method unimpeded by a mass of detail. As a further simplification, we have only described those parts of the BASIS/1 formalism required for the specification of SAL. However, we indicate where metalinguistic extensions are needed in defining a language like PL/I. The merits and demerits of the formalism are not discussed.

The paper is organized as follows: section 1 is an overview and discusses the PL/I standardization project in general, sections 2 through 4 define the metalanguage of BASIS/1, sections 5 and 6 provide a transition point with an informal discussion of SAL, and sections 7 through 13 give a formal definition of SAL using the metalanguage.

1.2 A Short History of BASIS/1

In 1969 the Joint Project for PL/I standardization was launched by ECMA and ANSI. The standard was developed through a process of successive refinements of the working document, BASIS/1, with new versions published about every six months. As a starting point, IBM gave the project their 1969 March PL/I Language Specifications modified to exclude some items thought to be unsuitable for standardization. These PL/I specifications were written in English and the Project soon realized that a more formal style would be necessary to obtain the required precision in the definition [B6].

The IBM Vienna Research Laboratories had by that time published a completely formal definition of a version of PL/I [A2, A3, L4, L5] in what is now known as the Vienna Definition Language [L3, W3]. This definition was based on the notion that interpretive execution of a program on an abstract machine constitutes a semantic description of the program. Originated by McCarthy [M1, M2], Landin [L1, L2], and Elgot [E1], this concept was first followed up by IBM Vienna. In the early stages there was also a parallel effort at the IBM Hursley Laboratories in England by Beech et. al. [A4, B2, B3, B4], who favored a semi-formal definition of PL/I using a machine-state that more closely resembled actual implementations.

Despte the perceived need for a rigorous definition, the Project felt that the strict formalism of the Vienna method would hinder the acceptance of the future standard. It was therefore decided to base the definition on the Hursley approach and retain the English language flavor. The language they developed to define PL/I, the BASIS/1 metalanguage, was

a semi-formal programming language with defined phrases that express the operations used in the definition allied to a completely formal specification of the metalanguage's operands. Although adopted by the whole project, the bulk of the work was carried out by a relatively small subcommittee [B7].

In accordance with ANSI standardization procedures, beginning on March 28, 1976, BASIS/1 was made available to the world-wide computer community for public comment. The result was twofold: first, several small technical errors were detected and second, modifications to the proposed PL/I were incorporated. For example, the standardization committee had decided to exclude the % INCLUDE feature from standard PL/I on the grounds that its inclusion would make unfair requirements on an implementor's operating system. Public outcry from structured programming enthusiasts, however, led to the incorporation of the % INCLUDE statement into standard PL/I.

BASIS/1 was accepted by the ANSI subcommittee on standardizing PL/I and on August 9, 1976, ANSI made this decision official. This has the following implication: if any U.S. government agency wishes to establish a standard regarding PL/I, the BASIS/1 standard must be used. For example, if the U.S. Army wishes to buy only computers with standard PL/I compilers, then those compilers must conform with BASIS/1 [S1].

The current activity of the standardization committee is to develope subsets of PL/I that are suitable for standardization. This task had been started during the preparation of the full standard but was postponed for lack of effort. Standard subsets were also requested during the public comment period. A standard realtime programming sublanguage is being developed and other types of sublanguages are under consideration.

1.3 Prerequisites

To understand BASIS/1 requires some acquaintance with the general notion of formal syntax and operational semantics as well as a familiarity with some existing version of PL/I. Here, we assume some technical background in formal syntax and semantics although a knowledge of PL/I is not essential. The reader is referred to [B5] for a general survey of the structure of PL/I, to [A1] for an introduction to formal syntax, and to [W2] for operational semantics.

2. THE DEFINITION METHOD

In 1962 Garwick[G1] proposed that the best way to provide a complete definition of a programming language was a particular implementation on a specific machine. This method of definition is obviously unsatisfactory for a machine independent language, nevertheless, it is frequently used in practice. It is not unknown for an implementor to "correct" a discrepancy between a compiler and its manual by changing the manual!

The definition method of BASIS/1 escapes from the inevitable interaction with host hardware and operating system by making use of a hypothetical machine devoid of any connection with specific real hardware. The definition is thus operational in nature, i.e. the meaning of a construct of the language is specified by the changes its execution causes in the state of the machine. These changes are described algorithmically in the definition, not because the algorithm must be followed precisely by an implementation but because it is a systematic way of achieving a complete definition of a complex language, thus able to answer unforseen questions. Implementations are free to use other algorithms and to take advantage of particular hardware features to provide the syntax and semantics specified by the definition.

Using the operational technique, BASIS/1 has specified every detail of PL/I in one of three ways:

- The exact specification is supplied in BASIS/1.
- The detail is specified as "implementation defined" in BASIS/1. For example, the maximum value for numbers is left for the implementor to define.
- The detail is specified explicitly as being "undefined". For example, the sequence in which subscripts are evaluated is undefined. This means that the implementor is free to choose the sequence but does not need to specify it to the user.

The essential point is that there are no gaps in the definition where nothing is specified at all.

BASIS/1 describes what an implementation must do to conform with the standard. As indicated above, an implementation is given great flexibility and may also choose to extend the language. The basic measure of conformity is that the implementation must provide all the linguistic features defined in BASIS/1 and that implementation defined extensions must not effect programs not using such extensions. Note that conformity is not that a given program with given input data must produce the same output data on all implementations.

2.1 The Abstract Machine

Although abstract, the hypothetical machine used in the definition has a considerable resemblance to the architecture of the real machine. The abstract machine is shown schematically in Figure 1.

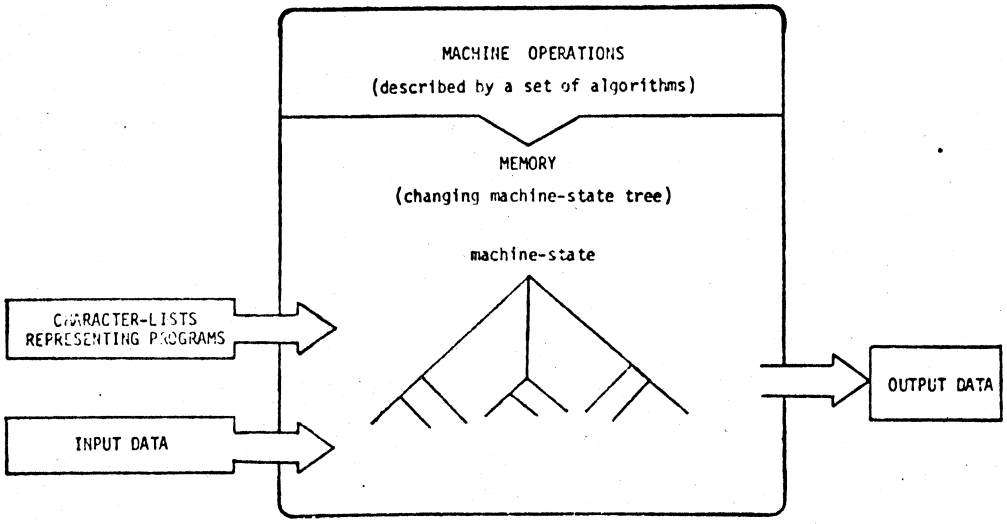


Figure 1. The Abstract Machine

The machine has a set of operations defined by algorithms that make use of a small set of standard basic instructions. Thus the algorithms are analogous to the microcode of a real computer.

The machine also has a memory whose contents can be changed by the machine operations. In this memory are stored the information used to control the execution of machine operations, a representation of the program being defined, and the values of the program's variables and datasets during its interpretation. The abstract machine's memory is thus the equivalent of both a real computer's main store and its microcode working store, together with on-line auxiliary storage. At any point in the definition process, all the information in the memory is represented by a single tree that defines the "machine-state."

2.2 The Definition Process

PL/I is defined by specifying the set of legal states of the abstract

machine and by defining algorithms for the machine operations. These operations are linked together into a single algorithm whose behavior specifies the meaning of any PL/I construct. For clarity of presentation, the algorithm is viewed as two separate processes: a translator which consists of parser and constructor phases, and an interpreter. Figure 2 shows a schematic representation of the definition algorithm.

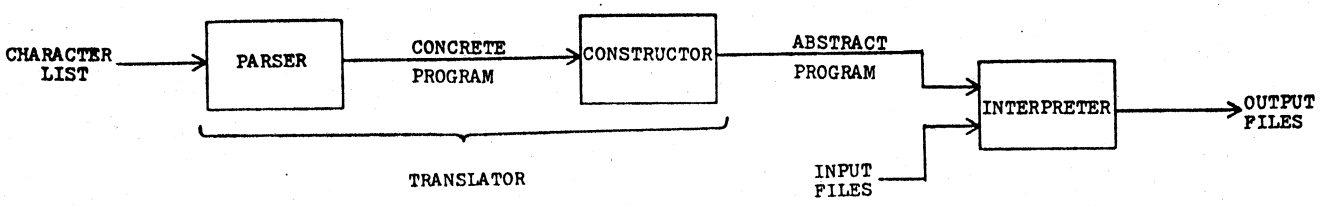


Figure 2. The Definition Process

The first step is to read in a list of characters representing the program to be defined. An attempt is made to parse this character-list according to the syntax of the written language. We use "attempt" here because, throughout the definition, checks are made that the program being defined is valid by the rules of the language. If the program fails any one of these tests it is rejected and the definition process stops at that point, leaving the meaning of the illegal program undefined. Only a valid program has a defined meaning.

Parsing the program transforms it character-list representation into a tree structure, the "concrete program," stored in memory. The next step is to construct from this concrete program an internal form suitable for interpretation, the "abstract program." The translator's parsing and

construction phases have many analogies with the phases of a compiler that build the internal form of a program prior to the code generation stage. The result is an abstract form of the original program where all the syntactic devices required in the written form of the program have been deleted and only those parts that are concerned with its meaning remain. During this translation phase, further validity checks are made on the program. For example, there is a check that no illegal combinations of data-types are present in expressions.

The final step in the definition process is the interpretation of the abstract program. During this phase, the abstract machine behaves very much like a real computer executing a program. For instance, part of the machine-stare contains the values of the program's variables, while another part keeps tract of the current statement being executed. In addition, the abstract machine reads and writes datasets. The meaning of the program is defined by the sequence of machine states generated as the program is interpreted.

The existence and relationships of the machine-state, the three forms of the program, and the datasets, as used in the definition process, are illustrated in Figure 3, an expansion of Figure 2.

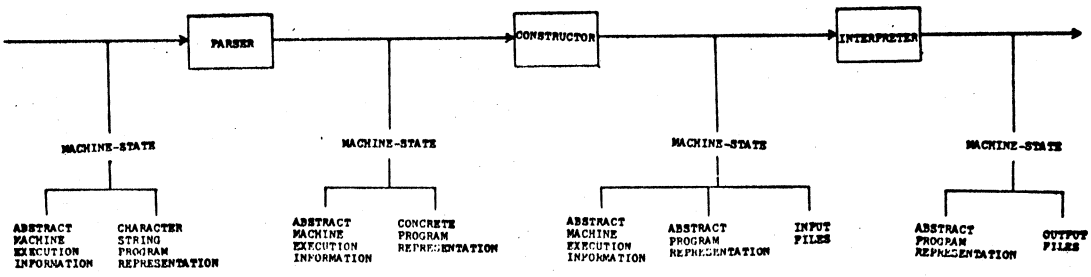


Figure 3. The Expanded Definition Process

3. The Abstract Machine's Data

All the data in the abstract machine are in the form of a tree. In this section we define some general tree terminology and then use these definitions to describe the tree-like data of the abstract machine.

3.1 Tree Terminology

In defining tree terminology, we refer to the example tree in Figure 4.

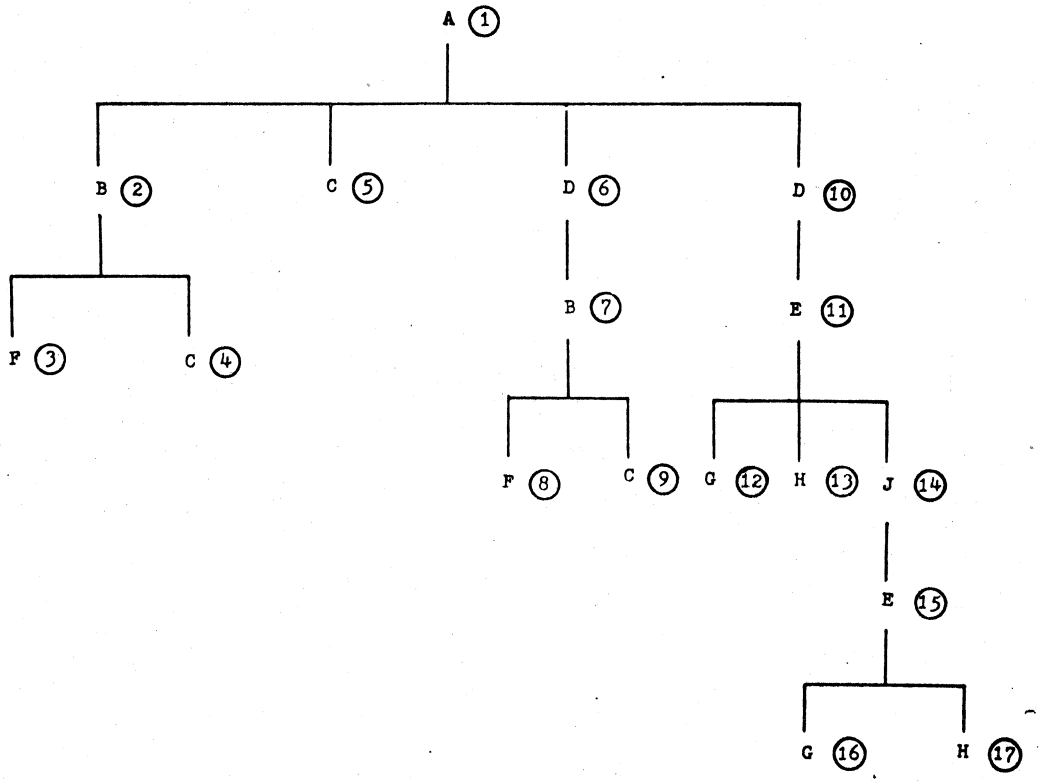


Figure 4. An example tree structure

Each of the points marked with a circled reference number is a node of the tree. Nodes (2), (3), . . . , (17) are subnodes or components

of node (1). Nodes (8) and (9) are immediate subnodes of and are immediately contained by node (7). A node's immediate subnodes are ordered from left-to-right and the meaning of the terms "first", "last", "leftmost", "rightmost", and "follows" are applied intuitively. Some nodes, for example, (3) and (17), do not have any subnodes. These are the terminal nodes of the tree.

Node (1) is the root node of the tree. Each of the other nodes are subtree nodes forming four immediate subtrees. Nodes (2), (5), (6), and (10) are the root nodes of these subtrees and themselves contain subtrees, and so on. Some subtrees are degenerate in that they consist of only one node, the root node. A reference to the root node of a tree is a reference to the whole tree unless otherwise stated.

Each node has a type associated with it. In Figure 3, for example, these are "A", "B", . . . , "J". In one tree there may be several distinct nodes of the same type, for example nodes (6) and (10) are both of type "D".

Although not used in defining SAL, BASIS/1 uses additional tree terminology to aid in defining PL/I's procedure calls, argument lists, and data structures.

3.2. The Machine State

The machine-state is a tree structure that completely represents the state of the abstract machine at all stages in the definition process. There are, of course, rules by which the tree is constructed just as there are rules by which valid sentences of a language can be constructed. These are the rules which comprise the syntax of the language. Similarly, we can refer to the rules for constructing the machine-state tree as the machine-state syntax. These rules specify the types of nodes that may be connected together in the tree.

Although there is just one tree throughout the definition, there are two distinguishable subtrees of the machine-state that play major roles. Hence, for convenience only, we isolate these subtrees and describe them by three quasi-separate syntaxes.

During the translation from the character-list form of the program to its abstract form, the concrete program is a subtree of the machine-state. The syntax of the concrete program is defined by a separate set of rules comprising the concrete syntax. The terminal nodes of the concrete program are the characters of the written form of the program. Thus the concrete-syntax defines the character-list representation of the set of syntactically valid programs.

The abstract program is another separate subtree of the machine-state. It is the end product of the translator and it is constructed in accordance with the abstract-syntax. The terminal nodes of the abstract program are not characters of the language being defined; rather, they are abstract entities used for program interpretation.

3.3 The Metabrackets

The three syntaxes are defined in Backus-Naur Form, BNF [B1], with a few extensions as described in Section 3.4. To help distinguish the type names used in the three syntaxes, characteristic metabrackets are used as part of the name:

SYNTAX	METABRACKETS		EXAMPLE
Concrete	⟨	⟩	⟨program⟩
Abstract	<	>	<program>
Machine-state	≺	≻	≺operation≻

The concrete syntax has the symbols and keywords of the programming language as its terminal symbols. The other two syntaxes denote terminal symbols by underlining the type name. For example, <fixed> and <undefined> are respective terminal symbols of the abstract and machine-state syntaxes.

3.4. The Definition of Trees

The rules of syntax are expressed as production rules in slightly extended BNF. For example, consider the BNF production:

⟨expression⟩ ::= ⟨expression-two⟩
| ⟨expression⟩ + ⟨expression-two⟩

This production specifies that an ⟨expression⟩ node may either have a single subnode, an ⟨expression-two⟩, or it may have three subnodes, an ⟨expression⟩ followed by a "+" followed by an ⟨expression-two⟩.

The symbols "::=", and "|" are metasymbols; they are not part of the language being defined, but part of the definition mechanism. In addition to these metasymbols of BNF, the syntax rules in BASIS/1 also use "[", "]", "{", and"}". These extra metasymbols are used as follows:

1. "[" and "]" enclose an optional syntactis expression. The production rule for ⟨expression⟩ given above can be written equivalently as rule HL16 in the concrete syntax of SAL:

HL16 ⟨expression⟩ ::= [⟨expression⟩ +] ⟨expression-two⟩

2. "{" and "}" enclose a syntactic expression, generally a set of options from which one must be chosen. For example, also from the

concrete syntax of SAL:

HL15 ⟨logical-expression⟩ ::= ⟨identifier⟩
| ⟨expression⟩ {= | ≠} ⟨expression⟩

This production states that a ⟨logical-expression⟩ either has a single ⟨identifier⟩ immediate component or it has three immediate components, two ⟨expression⟩ nodes separated by either an "=" or a "≠" character. Although not needed to define SAL, BASIS/1 uses an additional "permutation" metasymbol to reduce the number of BNF productions needed to express the fact that PL/I's myriad of data attributes may be listed in any order in data declaration statements.

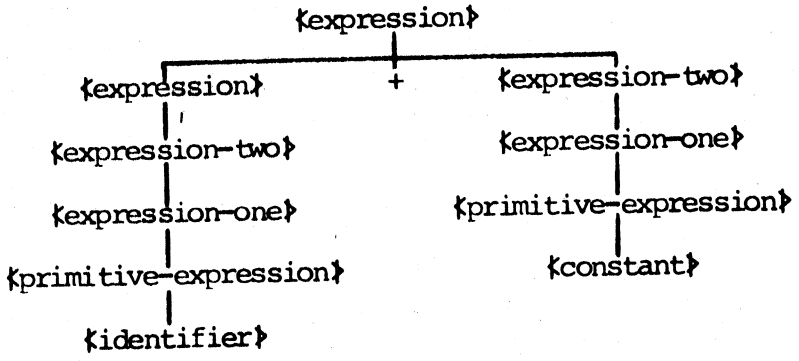
If we add the following productions to HL15 and HL16:

HL17 ⟨expression-two⟩ ::= [⟨expression-two⟩ *] ⟨expression-one⟩

HL18 ⟨expression-one⟩ ::= ⟨primitive-expression⟩
| - ⟨expression-one⟩
| (⟨expression⟩)

HL19 ⟨primitive-expression⟩ ::= ⟨identifier⟩
| ⟨constant⟩

we can draw an example from the set of trees of which ⟨expression⟩ is the root node.



Because this tree does not have all its terminal nodes, it is called a partial tree.

An alternative to this graphic representation of trees is their description by "enumeration". The enumerated tree form of the above partial tree is:

```
⟨expression⟩:
   ⟨expression⟩:
      ⟨expression-two⟩:
         ⟨expression-one⟩:
            ⟨primitive-expression⟩:
               ⟨identifier⟩;;;;
   +
   ⟨expression-two⟩:
      ⟨expression-one⟩:
         ⟨primitive-expression⟩:
            ⟨constant⟩.
```

The rules for describing trees by enumeration are:

1. the type of root node is listed. Optionally, this is followed by a colon and a listing of
2. the immediate components of the root node. Each of these components may itself be an enumerated tree.

An enumerated tree is terminated by

3. a semicolon. A string of semicolons at the end of an enumerated tree may be replaced by a period.

If a particular node of an enumerated tree is to be referenced specifically, the type name of the node can be followed by a comma and a local name for the node. Thus, in the partial tree:

```
{expression}:
   {expression}:
      {expression-two}:
         {expression-one};;
   +
   {expression-two}, rx:
      {expression-one}.
```

the name rx can be used to refer to the second {expression-two} node.

For clarity, enumerated trees are generally shown in an indented form. However, the notation does not depend on this for unambiguous representations of a tree.

Frequent use is made of sequences of one or more nodes of the same type. For example, in the abstract program the statements of the concrete program are represented by a sequence of <executable-unit> subtrees. These are collected together as immediate components of an <executable-unit-list> node. This notation is used wherever lists are required in the <machine-state>. Similarly, in the concrete program, there is frequent use of nodes of the same type separated by comma nodes. These are collected together as immediate subnodes of a -commalist node. The form of the enumerated tree for a {declaration-commalist} is:

```
{declaration-commalist}:
   {declaration}.
```

```
{declaration-commalist}:
   {declaration}
   {,}
   {declaration}.
```

```
{declaration-commalist}:
   {declaration}
   {,}
   {declaration}
   {,}
   {declaration}.
```

and so on. The metabrackets around the commas are used to avoid conflict with the notation of the enumerated tree.

3.5. Unique-names and Designators

Each node of the <machine-state> has a unique-name implicitly associated with it. During the definition process each node, as it is created, is given a unique-name that is different from the unique-names of all previously created nodes, whether or not these nodes still exist. These unique-names can be visualized as the circled reference numbers on the nodes in Figure 4.

Some nodes are of type <designator>. A designator node contains a copy of the unique-name of some other node and thus points to that node. Although designator nodes can point to any type of node, generally, for clarity, they point to one type of node only and have a type name that contains "-designator" as a suffix. For example, a <declaration-designator> is a node that only points to <declaration> nodes. In the abstract program a <variable-reference> contains a <declaration-designator> that points to the <declaration> for the variable being referenced. Figure 5 shows the way that this takes place in the <program>.

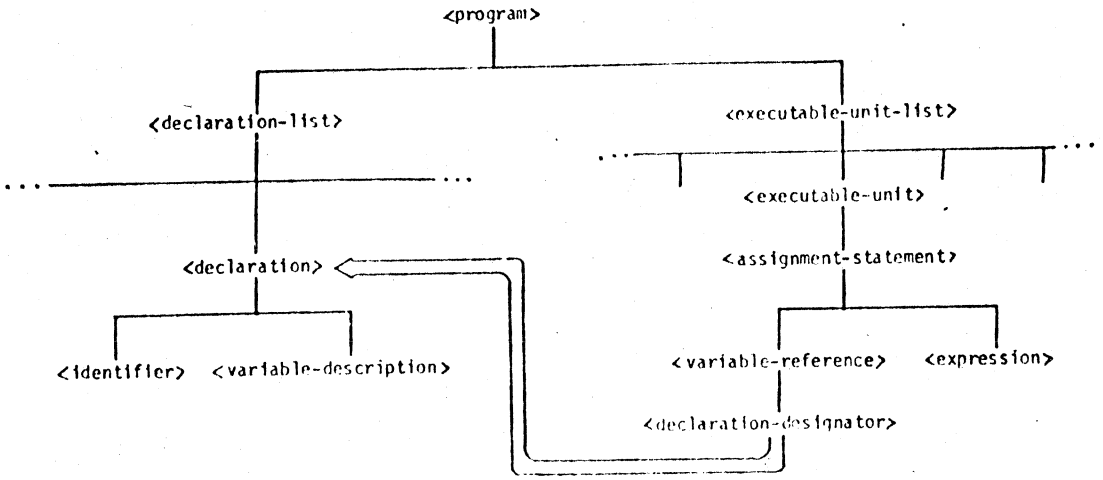


Figure 5. A fragment of an abstract program with a designator

Designators and trees are such that it is possible to reference the nodes that contain a designated node as well as the components of the designated node.

Two trees are said to be equal if they differ only in the unique-names of their nodes. A copy of a tree is constructed by creating a tree equal to the given tree and then changing any designators in the newly created tree that point to nodes in the given tree to point to the corresponding nodes in the new tree.

4. THE ABSTRACT MACHINE OPERATIONS

The operations of the abstract machine are specified by algorithms expressed in English prose. Although this makes the definition somewhat less formal, each algorithm is presented in a standard format and is written using precisely defined keywords and phrases, in effect a kind of programming language. A machine operation algorithm has many characteristics of a program; it has local variables that designate nodes on the <machine-state>, it can create temporary trees, and there are basic instructions for manipulating trees and doing arithmetic. Operations may invoke one another, possibly recursively, passing arguments and returning values. Internally, the control schemata are the usual sequential, conditional, and iterative forms.

In defining SAL we will use all of the abstract machine instructions that are used in BASIS/1 to define PL/I.

4.1 The Execution of Operations

At any time during the abstract machine's execution, there is one "active" operation, i.e., the one that the abstract machine is currently executing. The <operation> tree describing it is the rightmost element of an <operation-list> in the <machine-state>, as described in the next paragraph. Each <operation> has a subtree containing a list of designators pointing to its parameters, local variables with their current values, locally constructed trees, and an indication of where in its algorithm it is currently executing (i.e., a location counter). The invocation of an operation causes its <operation> tree to be added to the right-hand end of the <operation-list> and it thus becomes the active operation. When the operation terminates, it and any temporary trees it has created are deleted from the list and the operation that invoked it once again becomes the active operation, resuming at the point of suspension. In BASIS/1, the exact structure of an <operation> is left unformalized and unspecified since it is assumed that the workings of the operation can be understood without lower level of detail.

The <machine-state> at the start of the definition process has a <control-state> component with an <operation-list> containing a single operation named "define-program". This operation invokes other operations that build the concrete program, translate it into the abstract program and then start its interpretation. At this point, the situation is similar to that of an operating system that has loaded a problem program and is starting its execution. Control is passed to the problem program, often with a change of hardware location-counter. In the abstract machine a <program-control> component of the <machine-state> is created containing a second <operation-list> and, while it exists, its rightmost element is the active operation. The

operation at the right-hand end of the <operation-list> in the <control-state> is put into a state of suspended animation until the <program-control> and its <operation-list> are deleted from the <machine-state>. That happens when the interpretation of the abstract program terminates.

4.2. Operation Format

The following example is an operation of the abstract machine which defines SAL. It is not expected that the reader will fully understand the operation at this point. It is presented here to illustrate the structural features common to all operations.

Operation: create-assignment-statement(cas)

where: cas is an <assignment-statement>

result: an <assignment-statement>

Step 1. Let id and cx be respectively the immediately contained <identifier> and <expresion> of cas.
Step 2. Perform find-abstract-declaration(id) to obtain a <declaration-designator>, dd.
Step 3. Perform create-expression(cx) to obtain an <expression>, ax.
Step 4.
Case 4.1. ax immediately contains a <variable-reference>, vr.
The <attribute> contained by the <declaration> designated by dd must equal the <attribute> contained by the <declaration> designated by the <declaration-designator> of vr.
Case 4.2. ax immediately contains a <constant>, c.
If c contains an <integer-value> then the <declaration> designated by dd must contain <fixed>, otherwise it must contain <bit>.
Case 4.3. (Otherwise).
The <declaration> designated by dd must contain <fixed>.
Step 5. Return an

```
<assignment-statement>:
   <variable-reference>:
      dd;
   ax.
```

The written description of the operation consists of a heading and a body. The heading always contains the word "Operation" and the underlined

operation name. The remainder of the heading depends on the details of the operation, whether it has parameters, and whether it returns a value. The operation create-assignment-statement has a single parameter with the local name "cas". A parameter is a designator pointing at a node in the <machine-state>, possibly in the caller's local storage. Parameters are thus passed by reference and it is possible to change the value of the tree designated by the parameter.

The types of the nodes designated by the parameters are specified in the where-clause. In some operations there may be several alternative types for a parameter, the particular one actually designated varying from invocation to invocation. In our example, the parameter cas designates an <assignment-statement> node and thus, the whole tree of which it is the root node. An operation may return a complete tree, in which case the type of its root node will be specified in the result-clause of the heading. The create-assignment-statement operation returns a complete tree with an <assignment-statement> root node.

The body of an operation consists of either a sequence of Steps or a set of mutually exclusive Cases, numbered sequentially. Each Step or Case can itself contain a nested sequence of Steps or a set of Cases. If so, the numbering in the i'th Step or Case will be sequential from i.1. This nested structure continues to arbitrary depth. The Steps of an operation are executed sequentially except when modified by a control instruction. Each Case is preceded by a predicate whose truth value determines whether the body of the Case is to be executed. There must always be one and only one Case whose predicate is true when any set of Cases is executed. For brevity, the predicate of the last Case may be "(Otherwise)" which is true if and only if all the

other Cases are false. This abbreviation is only used where it saves writing out a lengthy negation of all the previous predicates.

4.3. Variables

In the body of the operation, local variables are used to designate parts of the ⟨machine-state⟩, locally constructed trees, and parameters. They may also be used to contain integer values. These local variables are given names consisting of a few alphanumeric characters, usually of mnemonic significance. By convention, these names are distinct from English words to avoid confusion with the text. Local variables may be subscripted. For example, nt[i] is an element of a vector of local variables nt, the value of the variable i determining a particular element.

Both local variables and locally constructed trees exist only for as long as the operation is on an ⟨operation-list⟩. As soon as the entry is deleted from the list, the local variables and trees cease to exist. However, a local tree may be returned as a value of an operation, in which case, it is copied to form a tree local to the caller.

4.4. Tree manipulation instructions

The let instruction makes a local variable designate an existing tree or a newly created tree. For example, in the operation create-assignment-statement:

Step 1. Let id and cx be respectively the immediately contained ⟨identifier⟩ and ⟨expression⟩ of cas.

The variable cas is a parameter of the operation and designates an ⟨assignment-statement⟩ node in the concrete program. This node is defined by the Concrete

Syntax rule:

HL11. ⟨assignment-statement⟩ ::= ⟨identifier⟩ = ⟨expression⟩

This let instruction creates two local variables, id and cx, which respectively designate the ⟨identifier⟩ and ⟨expression⟩ immediate components of the node designated by cas. Both these trees existed before the let instruction was executed. In the following let instruction:

```
Let dso be
      <output-dataset>:
        <dataset>:
          <alpha>
          <omega>.
```

a tree is constructed and the local variable dso is made to designate it. Another way to construct a tree is by copying trees designated by local variables. For example:

```
Let ids be
     <input-dataset>:
         ds
         <current-position>:
             dq;;
```

Here, a tree with root node of type <input-dataset> is constructed and one of its immediate components is a copy of the tree designated by the local variable ds. Similarly, the new constructed tree contains a copy of the tree designated by the ⊠ local variable dq. The variable ids designates the entire newly constructed tree.

There is an implicit form of the let instruction in which the name of a local variable is listed following some description of a root node and a comma. For example, in the predicate of the case:

Case 1.1.2. cx immediately contains a ⟨constant⟩, cn. If the predicate is true then the local variable cn is made to designate the ⟨constant⟩ node. This form of the let instruction can also be used in

enumerated trees.

The let instruction is also used to introduce a vector of local variables. For example:

> Let nt[i], i = 1,...,n be the ordered list of nodes which are the immediate components of the ⟨delimiter⟩ and ⟨non-delimiter⟩ nodes of t.

sets a vector of n designators nt. References to elements of this vector will be subscripted with a local variable containing an integer value.

The replace instruction is used to substitute a specific tree for a tree designated by a local variable. For example:

> Replace the ⟨basic-value⟩ designated by bvd by a copy of bv.

The replacement takes place at the node designated by the local variable and the unique name of the original node becomes the unique name of the root of the replacement.

The append instruction attaches a tree as the rightmost element of a list. By definition, there are no empty list nodes in the ⟨machine-state⟩. To avoid special cases, the append instruction will construct the -list node if it is appending an element to a non-existant list. For example:

```
Append an
        <executable-unit>:
           id
           axs;
to the  <executable-unit-list> of the <program>.
```

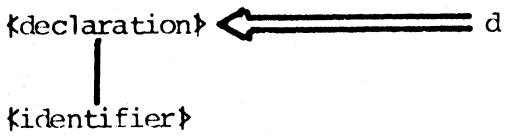
Here, the append instruction causes a tree consisting of an ⟨executable-unit⟩ with two immediate components to be built and then added as the rightmost component of the ⟨executable-unit-list⟩ immediately contained by the abstract program. The first time this instruction is executed, the ⟨executable-unit-list⟩ node will have to be created and connected to the ⟨program⟩ node.

The action of the remaining tree manipulation instructions, unlike those described so far, depends on the syntax of the trees being manipulated. The attach instruction constructs a tree by joining a specific tree to a designated node. To make the link, the instruction may create the minimum number of intervening nodes required by the syntax rules for the tree. For example, the tree for a ⟨declaration⟩ is defined by the rules:

HL5. ⟨declaration⟩ ::= ⟨identifier⟩ [⟨attribute⟩]

HL6. ⟨attribute⟩ ::= FIXED | BIT

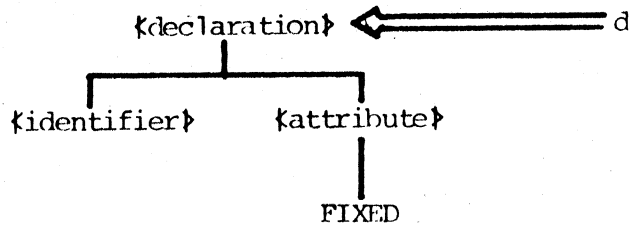
Suppose the local variable d designates a ⟨declaration⟩ that does not contain an ⟨attribute⟩ and thus can be represented as:



Then the result of executing the instruction

Attach FIXED to d.

is to make the tree designated by d look like:



The delete instruction causes a designated tree to be deleted from its containing tree. If the deleted tree was a mandatory component of its immediately containing node, then this node is also deleted and the process is repeated until a legal tree is obtained. All deleted nodes are discarded and cease to exist. For example, part of the ⟨machine-state⟩ is defined by the

rules:

M5. ⟨interpretation-state⟩ ::= [⟨program-state⟩] ⟨datasets⟩

M6. ⟨program-state⟩ ::= ⟨program-control⟩ ⟨allocated-storage⟩

Execution of the instruction

Delete the ⟨program-control⟩ from the ⟨machine-state⟩.

removes the ⟨program-control⟩. But, since it is a required component of ⟨program-state⟩, the ⟨program-state⟩ node and its components are also deleted from the ⟨machine-state⟩. The ⟨program-state⟩ is only an optional node of the ⟨interpretation-state⟩ and therefore the deletions stop at this point.

4.5. Control Instructions

The execution sequence of an operation's steps follows the order in which they are written unless one of the control instructions is executed. In the normal sequential flow of control, once the last step of an operation has been executed the operation is terminated and deleted from the ⟨operation-list⟩, thus returning control to the operation that invoked it.

Of the control instructions, the go to instruction is the simplest. Its execution transfers control to a step in the active operation. For example:

Go to Step 1.

Control can be returned explicitly from an operation to the calling operation either by executing a terminate instruction or by executing a return instruction. The terminate instruction is written:

Terminate this operation.

The return instruction not only returns control to the invoking operation but also passes back a value. If the returned value is a local tree belonging to the operation, the tree is copied to become a local tree of the calling operation. For example:

```
Return
        <logical-expression>:
           <variable-reference>:
              dd.
```

sends the specified tree back to the caller where it will be designated by a local variable.

Control is passed to another operation by invoking it with the perform instruction. For example, the instruction:

> Perform create-logical-expression(cle) to obtain a <logical-expression>, alx.

causes the create-logical-expression operation to be invoked. The local variable cle strictly designates a tree and this designator is passed as an argument. An <operation> for create-logical-expression becomes the active operation. During this activation, the designator value being passed as an argument is given a local name and is treated like a local variable. The "obtain" part of the perform instruction is optional. Where applied, it describes the type of value to be returned and specifies a local variable, in this case alx, to designate this returned value. When control is returned to the calling operation, execution resumes immediately following the perform instruction.

In some circumstances, usually after a program execution error, it is an implementation decision whether an operation is to be performed. In these cases, the phrase "optionally perform" is used. For example,

in the instruction

> "If the magnitude of ir exceeds an implementation-defined maximum, then let ir be an <integer-value> with an implementation-defined value and optionally perform abnormal-termination."

if the computed value ir has a maximum greater than the implementation's maximum allowable value, the implementation has the option of continuing or terminating the program's execution.

The for each instruction specifies that a sequence of instructions is to be executed once with each member of a set of objects. For example:

> For each <variable-reference>, vr, of the <variable-reference-list> of st, taken in left-to-right order, perform Steps 1.1 through 1.4.

Here, the perform instruction is used to cause the execution of a self-contained group of substeps similarly to the way that it is used to cause the execution of a complete operation. The Steps 1.1 through 1.4 will be executed once for each element of the <variable-reference-list>. Each time they are executed, the local variable vr will designate the <variable-reference> currently being operated on. Unless an ordering is specified, as it is in this example, the order in which the elements of the list are chosen for processing is arbitrary.

The if instruction, although strictly speaking not a control instruction since it does not change the order of execution of Steps, does have some effect on the execution of the instructions in the Step. The if instruction specifies that in the case that the stated condition is true, the instruction

list that follows the then is to be executed. For example:

> If the rightmost immediate component of ul does not contain {return-statement} then append
> {unit}:
> {executable-unit}:
> {executable-single-statement}:
> {return-statement}:
> RETURN
> {;}.
> to ul.

Optionally, the if instruction can contain an otherwise part, in which case the instructions that follow it are executed only if the stated condition is false. Thus, for example, in the instruction:

> If cd contains FIXED then attach <fixed> to ad, otherwise attach <bit> to ad.

if the condition "the node designated by cd contains FIXED" is true, then the node <fixed> will be attached to the node designated by ad. If the condition is false, <bit> will be attached. As in BASIS/1, we have no conflict with the scope of an otherwise part since there are no nested uses of the if instruction.

4.6 Validity Checking

In both the translator and the interpreter validity tests are frequently applied to the program. These are specified by the must and the must not instructions. For example:

> The <declaration> designated by dd must contain <fixed>.

or:

> The <basic-value> designated by bvd must not contain <undefined>.

In either case, if the condition is not satisfied the original source program is in error and its meaning is undefined. The abstract machine stops in an undefined state at this point. This is analagous to the situation in a real

machine for some types of program error.

4.7 Dynamic Macro

In both translator and interpreter operations it often happens that one of a set of very similar cases is chosen depending on the type of node being considered. This could, for example, be written as:

Step 2.
Case 2.1. cxs is an {if-statement}.
Perform create-if-statement(cxs) to obtain an <if-statement>, axs.
Case 2.2. cxs is an {assignment-statement}.
Perform create-assignment-statement(cxs) to obtain an <assignment-statement>, axs.

. . . . . .

Case 2.6. cxs is a {write-statement}.
Perform create-write-statement(cxs) to obtain a <write-statement>, axs.

To avoid this rather lengthy case enumeration, a so-called "dynamic macro" instruction is used and the above step is written as follows:

Step 2. Perform create-xxx-statement (cxs) to obtain an <xxx-statement>, axs, where {xxx-statement} is the type of cxs.

Thus the use of "xxx" is analogous to the character string matching and substitution commonly used in macro assembler languages.

5. INFORMAL DESCRIPTION OF SAL

A definition of SAL, a very small language of no practical value, will be used to demonstrate the BASIS/1 method of language definition. The following is an example of a program written in SAL:

```
        DECLARE I FIXED,
                J,
                B BIT;
        I = 2;
TOP:    READ INTO(A, B);
        IF A ≠ I
           THEN J = I;
           ELSE J = A * I;
        WRITE FROM(J);
        I = I + 3;
        IF B
           THEN GO TO TOP;
        RETURN;
    END;
```

A program in SAL is a list of statements terminated by an end-statement. Apart from the end-statement, there are assignment, conditional, declaration, go-to, read, return, and write statements. Like PL/I, there are no reserved words in the language. The distinction between keywords and identifiers is made solely on context.

5.1. Variables

Variables may be declared in a non-executable declare statement that can occur anywhere in the program. One of the two attributes, FIXED or BIT, may be given to a variable. Fixed variables take positive or negative integer values and bit variables take values 0 or 1, meaning false or true respectively. If a variable is not declared, an implicit declaration for it is constructed. In the above example, there is no declaration for the variable A and it will be implicitly declared. If a variable is not given an attribute in a declaration, like J in the written declaration and A in the constructed declaration, it will receive the attribute FIXED by default. Thus both A and J will be FIXED variables.

5.2. Assignment Statements and Expressions

The execution of an assignment statement causes the evaluation of an expression, producing either a fixed or bit value. This value is then assigned to the variable referenced on the left-hand side of the equals symbol. The type of the value must match the attribute of the variable to which it is assigned, i.e., there is no type conversion. An expression may be a constant, as in I = 2;, a reference to a variable, as in J = I;, a prefix expression, an infix expression, as in J = A * I;, or a parenthesized expression. The prefix and infix expressions are restricted to operands that have integer values. The prefix expression uses the negation operation and the infix expression offers a choice of addition and multiplication. These operations have the normal precedence and parentheses may be used to change it in the usual way. In the event of overflow, the effect is implementation defined. It is an implementation decision whether the program is abnormally terminated or an implementation-defined value is produced. Constants can be either decimal representations of integers or bit values represented by "0B" and "1B".

5.3 Conditional Statements

The conditional statement is of a conventional IF-THEN-optional-ELSE form. The then and else parts may only be single statements and may not be another conditional. The logical expression may be either a reference to a bit variable or a comparison between the integer values of two expressions. Both the equals and not-equals comparisons are available.

5.4. Labels

Any executable statement, except the then and the else parts of a conditional statement may have a label. In the example, TOP: is a statement label. The go-to statement causes control to be transferred unconditionally to the named statement.

5.5. Input and Output

The input and output statements interact with a pair of files. The input file, consisting of a list of integer and bit values, is read sequentially by the read-statement. In executing a read-statement values from the input file are assigned to the variables in the read-statement's list, in left-to-right order. The type of the value read must match the type of the variable to which it is assigned. It is an error leading to abnormal termination for a program to read beyond the end of the input file. The output file is initially empty and the write-statement appends integer and bit values to it.

5.6. The Return Statement

Execution of the return-statement causes normal termination of the program. If there is no return statement immediately before the end-statement, one is assumed.

6. The Running Example

Having described the BASIS/1 definition mechanism and informally described SAL, the latter part of this paper will present a formal definition of SAL in terms of the mechanism. To illustrate the workings of the definitional process and the trees that are constructed by the Translator and Interpreter, we make use of an example SAL program, the so-called "running example." The running example is:

```
DECLARE Y BIT,
        Z;
READ INTO (Y, Z);
IF Y THEN X = 2*Z + 1;
     ELSE X = 0;
WRITE FROM (X);
END;
```

Hence, the rest of the paper will consist of the following parts:

(1) Initialization of the Abstract Machine.

(2) Definition of the parser operations for SAL.

(3) Parsing the running example to produce the concrete program.

(4) Definition of the constructor operations for SAL.

(5) Construction of the running example's abstract program.

(6) Definition of the interpreter operations for SAL.

(7) Interpretation of the running example.

The definitions of the three syntaxes for SAL will be interspersed in appropriate places.

7. INITIALIZATION OF THE ABSTRACT MACHINE

In this section we describe the initialization of the abstract machine which takes place at the beginning of the definition process. First we give the syntax rules which define the ⟨machine-state⟩ during the translation of the character string representation of the SAL program into its abstract program equivalent. We then give the algorithms that control the definition process.

7.1. State of the Abstract Machine During Translation

M1. ⟨machine-state⟩ ::= ⟨program⟩ ⟨control-state⟩
[⟨translation-state⟩ | ⟨interpretation-state⟩]

M2. ⟨control-state⟩ ::= ⟨operation-list⟩

M3. ⟨translation-state⟩ ::= [⟨program⟩]

M4. ⟨operation⟩ ::=

The exact structure of ⟨operation⟩ is left unformalized and unspecified. During translation, the ⟨machine-state⟩ contains a ⟨translation-state⟩ component.

The translation phase consists first of reading and parsing the source program to form the ⟨program⟩ component of the ⟨translation-state⟩. The ⟨program⟩ is then translated into its abstract form which is attached to the ⟨program⟩ component of the ⟨machine-state⟩.

7.2 Machine Initialization

To begin the definition process the abstract machine is given the following initial ⟨machine-state⟩ tree

```
⟨machine-state⟩:
   ⟨program⟩
   ⟨control-state⟩:
      ⟨operation-list⟩:
         ⟨operation⟩ for define-program ;;
   ⟨translation-state⟩.
```

At this point, since the ⟨operation⟩ for define-program is the rightmost operation of the ⟨operation-list⟩, define-program becomes the active operation and the abstract machine starts to execute it.

7.3. The Define-program Operation

This is the top-level algorithm that controls the whole definition process.

Operation: define-program

Step 1. Perform translation-parse-phase.
Step 2. Perform translation-construction-phase.
Step 3. Perform interpretation-phase.

The translation-parse-phase operation reads and parses the source program, the translation-construction-phase operation translates the concrete program into its abstract form, and the interpretation-phase operation interprets the abstract program.

7.4. The Running Example

The execution of Step 1 in the define-program operation changes the ⟨machine-state⟩ tree to that shown in Figure 6. At this point, translation-parse-phase is the active operation.

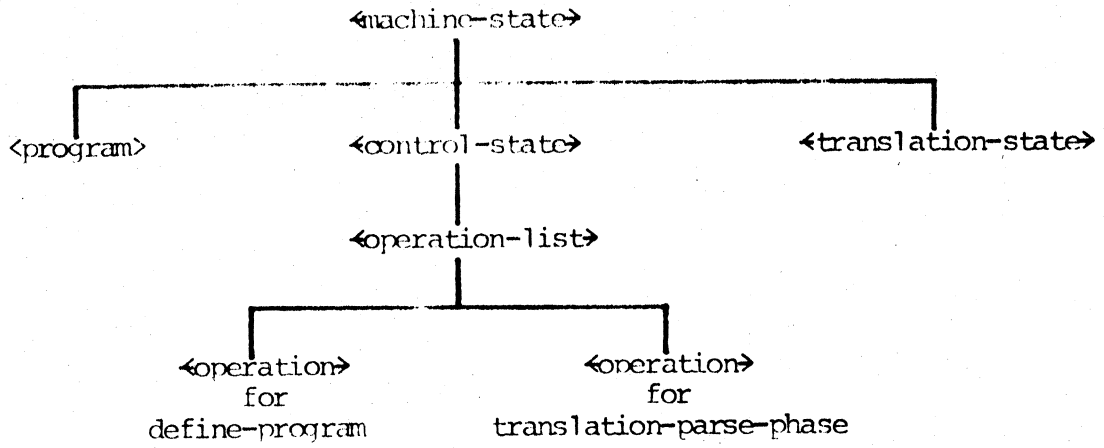


Figure 6. ⟨machine-state⟩ on executing Step 1 of define-program.

8. THE CONCRETE SYNTAX

The concrete syntax of SAL specifies the written form of the language and also the concrete tree. The concrete syntax is divided into two parts, a low-level syntax and a high-level syntax. The low-level syntax classifies sequences of characters from the written form of the program, the "text", into non-delimiters (which are words and constants) separated by delimiters. The high-level syntax defines the way that a program is built from delimiters and non-delimiters. The separation of the concrete syntax into two parts is done to facilitate the context-sensitive removal of blanks and the separation of words into identifiers and keywords. Because SAL does not have reserved words, keywords must be distinguished from identifiers purely on the basis of context.

8.1 The Low-level Syntax

Syntax for text

LL1. ⟨text⟩ ::= [⟨delimiter-list⟩] ⟨delimiter-pair-list⟩

LL2. ⟨delimiter-pair⟩ ::= ⟨non-delimiter⟩ ⟨delimiter-list⟩

LL3. ⟨delimiter⟩ ::= + | * | - | = | ≠ | (|) | , | ; | : | ␣

Note: "␣" denotes a blank.

LL4. ⟨non-delimiter⟩ ::= ⟨identifier⟩
| ⟨constant⟩

Syntax for identifiers and constants

LL5. ⟨identifier⟩ ::= ⟨letter⟩
| ⟨identifier⟩ {⟨letter⟩ | ⟨digit⟩}

LL6. ⟨letter⟩ ::= A | B | C | D | E | F | G | H | I | J | K | L | M | N
| O | P | Q | R | S | T | U | V | W | X | Y | Z

LL7. ⟨digit⟩ ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

LL8. ⟨constant⟩ ::= ⟨fixed-constant⟩
| ⟨bit-constant⟩

LL9. ⟨fixed-constant⟩ ::= ⟨digit-list⟩

LL10. ⟨bit-constant⟩ ::= {0 | 1}B

Syntax for characters

The input to the definition process is a ⟨character-list⟩. There are 47 characters in the SAL character set. Each character of the ⟨character-list⟩ that represents the program being defined belongs to one of three groups: digits, letters, and delimiters.

LL11. ⟨character⟩ ::= ⟨digit⟩
| ⟨letter⟩
| ⟨delimiter⟩

8.2. The High-level Syntax

The goal of the high-level syntax is to classify sequences of delimiters and non-delimiters into units which correspond to SAL statements.

Syntax for program

HL1. ⟨program⟩ ::= ⟨unit-list⟩ ⟨end-statement⟩

HL2. ⟨unit⟩ ::= ⟨declare-statement⟩
| ⟨executable-unit⟩

HL3. ⟨end-statement⟩ ::= END ;

Syntax for declarations

HL4. ⟨declare-statement⟩ ::= DECLARE ⟨declaration-commalist⟩ ;

HL5. ⟨declaration⟩ ::= ⟨identifier⟩ [⟨attribute⟩]

HL6. ⟨attribute⟩ ::= FIXED
| BIT

Syntax for executable-units

HL7. ⟨executable-unit⟩ ::= [⟨statement-name⟩]
{⟨if-statement⟩ | ⟨executable-single-statement⟩}

HL8. ⟨statement-name⟩ ::= ⟨identifier⟩ :

HL9. ⟨if-statement⟩ ::= IF ⟨logical-expression⟩
THEN ⟨executable-single-statement⟩
[ELSE ⟨executable-single-statement⟩]

Syntax for single-statements

HL10. ⟨executable-single-statement⟩ ::= ⟨assignment-statement⟩
| ⟨goto-statement⟩
| ⟨read-statement⟩
| ⟨return-statement⟩
| ⟨write-statement⟩

HL11. ⟨assignment-statement⟩ ::= ⟨identifier⟩ , = ⟨expression⟩ ;

HL12. ⟨goto-statement⟩ ::= GOTO ⟨identifier⟩ ;

HL13. ⟨read-statement⟩ ::= READ INTO (⟨identifier-commalist⟩) ;

HL14. ⟨return-statement⟩ ::= RETURN ;

HL15. ⟨write-statement⟩ ::= WRITE FROM (⟨identifier-commalist⟩) ;

Syntax for expressions

HL16. ⟨logical-expression⟩ ::= ⟨identifier⟩
| ⟨expression⟩ {= | ≠} ⟨expression⟩

HL17. ⟨expression⟩ ::= [⟨expression⟩ +] ⟨expression-two⟩

HL18. ⟨expression-two⟩ ::= [⟨expression-two⟩ *] ⟨expression-one⟩

HL19. ⟨expression-one⟩ ::= ⟨primitive-expression⟩
| - ⟨expression-one⟩
| (⟨expression⟩)

HL20. ⟨primitive-expression⟩ ::= ⟨identifier⟩
| ⟨constant⟩

9. THE TRANSLATOR (PARSE PHASE)

The function of the parse phase of the translator is to take the character list representation of the SAL program and generate a corresponding concrete program. The parsing is performed in two stages corresponding to the two levels of the concrete syntax.

9.1 The Operations

Operation: translation-parse-phase

Step 1. Obtain from a source outside this definition a sequence of characters constructed in the form of a ⟨character-list⟩, cl.
Step 2. Perform parse(cl) to obtain a ⟨program⟩, cp.
Step 3. Attach cp to the ⟨translation-state⟩.

Operation: parse(cl)

where: cl is a ⟨character-list⟩

result: a ⟨program⟩

Step 1. Perform low-level-parse(cl) to obtain a ⟨text⟩, tx.
Step 2. Perform high-level-parse(tx) to obtain a ⟨program⟩, cp.
Step 3. Return cp.

Operation: low-level-parse(cl)

where: cl is a ⟨character-list⟩

result: a ⟨text⟩

Step 1. There must exist one and only one tree, tx, with respect to the low-level syntax for ⟨text⟩, such that the terminal nodes of tx, taken in left-to-right order, form a ⟨character-list⟩ equal to cl.
Step 2. Return tx.

A keyword is an ⟨identifier⟩ which maps into a type specified in the high-level syntax by explicit spelling without any metabrackets.

The following is a production for a type that is used solely in the following operation and is thus specified here rather than in the concrete syntax:

⟨delimiter-or-non-delimiter⟩ ::= ⟨delimiter⟩
| ⟨non-delimiter⟩

Operation: high-level-parse(tx)

where: tx is a ⟨text⟩

result: a ⟨program⟩

Step 1. Let t be a ⟨delimiter-or-non-delimiter-list⟩ which contains a copy of the ⟨delimiter⟩ and ⟨non-delimiter⟩ components of tx in exactly the same order.

Step 2. Delete from t any ⟨delimiter⟩ containing a "ϕ". This must not cause t to be deleted.

Step 3. Let nt[i], i = 1,...,n be the ordered list of nodes which are the immediate components of the ⟨delimiter⟩ and ⟨non-delimiter⟩ nodes of t.

Step 4. There must exist one and only one tree, ht which is a complete tree with respect to the high-level syntax for ⟨program⟩ such that ht contains terminal nodes nht[i] i=1,...n and there is a one-to-one correspondence between nt[i] and nht[i] as specified by Cases 4.1 through 4.3.

Case 4.1. nht[i] is a keyword.
The node nt[i] must be an ⟨identifier⟩ containing the same terminals as the characters appearing in nht[i].

Case 4.2. nht[i] is an ⟨identifier⟩ or ⟨constant⟩.
The nodes nt[i] and nht[i] must be of the same type. Replace nht[i] by nt[i].

Case 4.3. nht[i] is a non-bracketed type other than a keyword.
nt[i] and nht[i] must be equal.

Step 5. Return ht.

9.2 Application to the Running Example

The Parse Phase of the Translator is illustrated, first by taking part of the character representation of the Running Example through the low-level parse and then showing the build up of the entire program. Figure 7 shows the situation at Step 1 of translation-parse-phase. The section of the ⟨character-list⟩, cl that corresponds to:

IF Y THEN X = 2*Z + 1;

is shown with each character classified as a ⟨letter⟩, ⟨digit⟩, or ⟨delimiter⟩ according to the syntax for characters LL11.

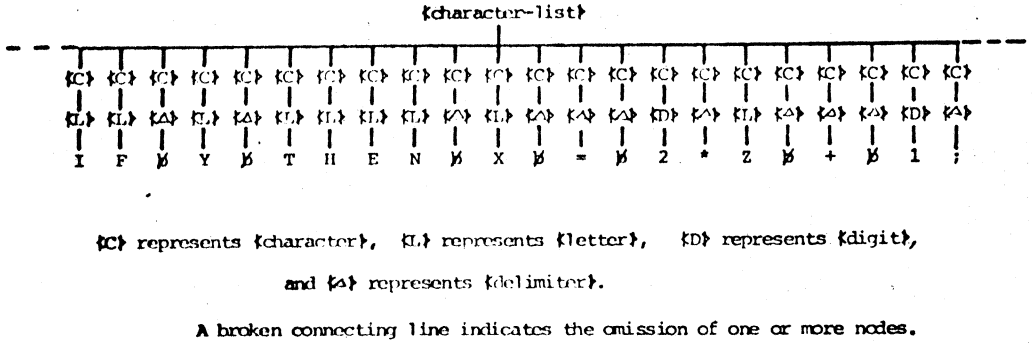


Figure 7. Part of the ⟨character-list⟩ representation of the Running Example.

The result of applying the operation low-level-parse to this ⟨character-

list⟩ is to obtain the ⟨text⟩ tree which consists of a ⟨delimiter-pair-list⟩ as shown in Figure 8. The section of ⟨text⟩ that is shown there corresponds to the same section of the ⟨character-list⟩ that was shown in Figure 7.

The operation high-level-parse constructs a ⟨delimiter-or-non-delimiter-list⟩ that matches the ⟨delimiter⟩ and ⟨non-delimiter⟩ components of the ⟨text⟩. The section of the ⟨delimiter-or-non-delimiter-list⟩ that is derived from the part of ⟨text⟩ shown in Figure 8 is shown in Figure 9a. In order to save space, the trees for ⟨identifier⟩ and ⟨constant⟩ are represented by their root and terminal nodes only. The next step is to remove the blanks from the ⟨delimiter-or-non-delimiter-list⟩. The result of this is shown in Figure 9b. The correspondence between elements of the vector nt of Step 3 and nodes of the ⟨delimiter-or-non-delimiter-list⟩ is also shown.

The high-level-parse continues with the construction of a tree according to the high-level syntax. Step 4 constructs this tree with a ⟨program⟩ root node, designated by the local variable ht. Its terminal nodes are either ⟨identifier⟩, or ⟨constant⟩ nodes or else delimiters. These terminal nodes must match in left-to-right order the nodes of the ⟨delimiter-or-non-delimiter-list⟩. Figure 10 shows the section of the ⟨program⟩ tree ht that corresponds to the section of the ⟨delimiter-or-non-delimiter-list⟩ of Figure 9b. The elements of the vector nht of Step 4 that designate nodes in the figure are also shown.

It is at this point that the distinction is made between keywords and program identifiers and the specific details for each ⟨identifier⟩ and ⟨constant⟩ are filled in. This results in the ⟨program⟩ tree section shown

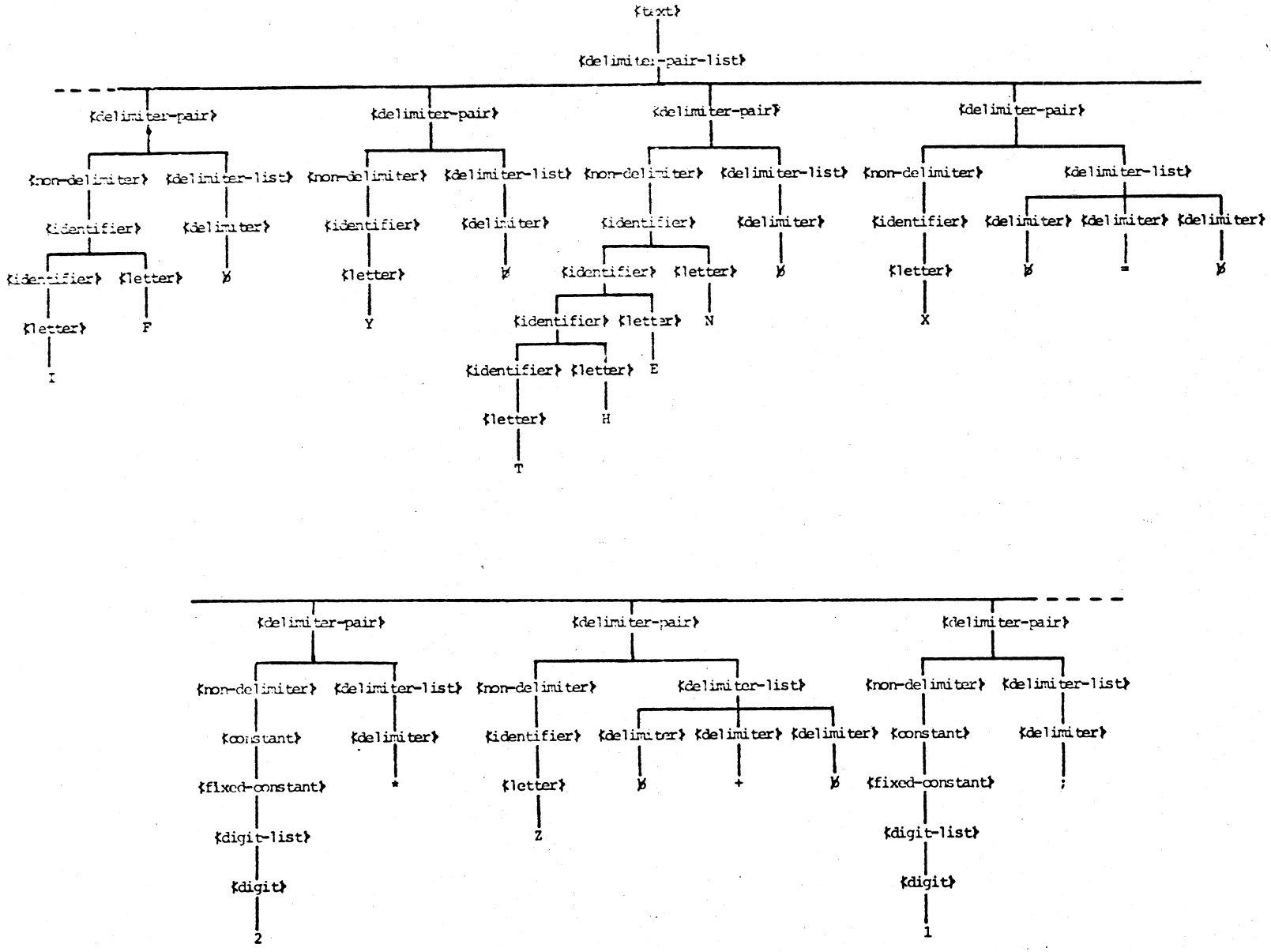


Figure 8. Part of the ⟨text⟩ representation of the Running Example

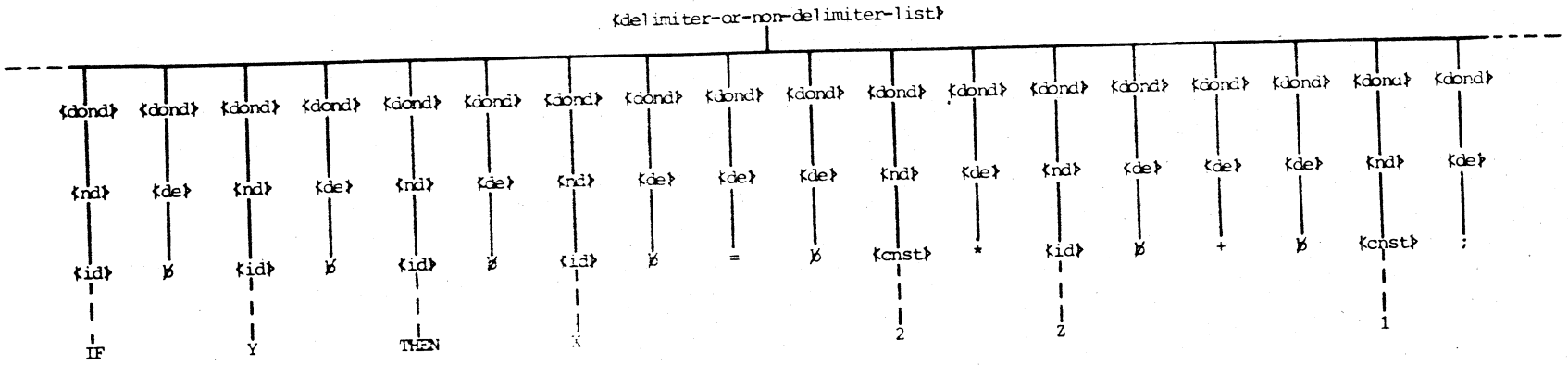
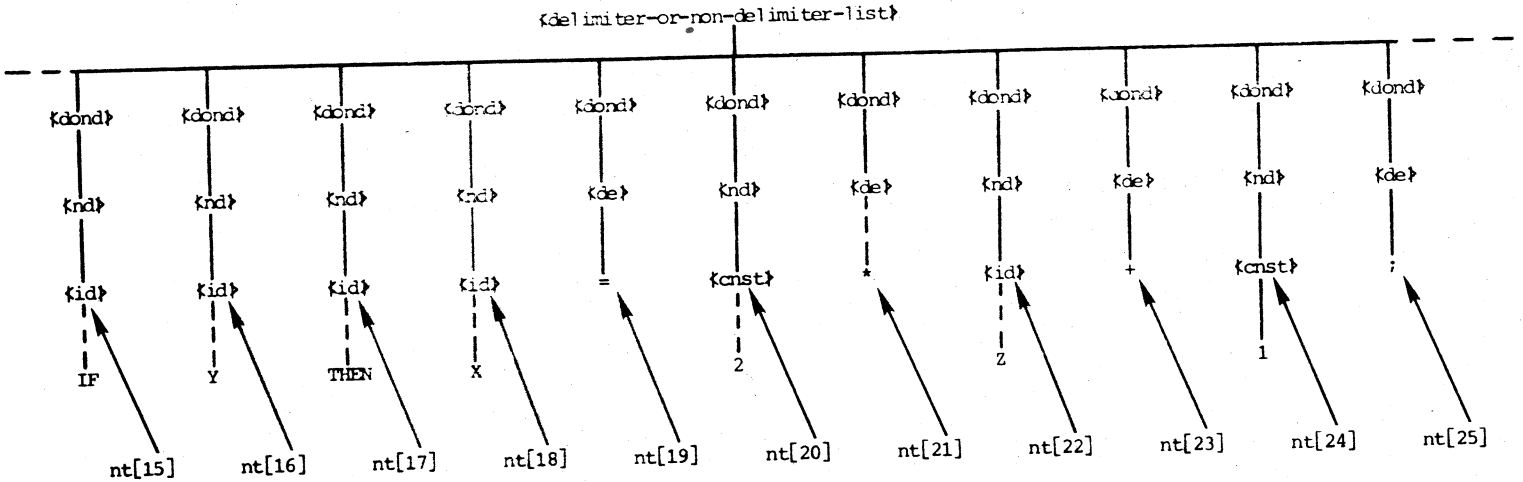


Figure 9 a. Part of the ⟨delimiter-or-non-delimiter-list⟩ representation of the Running Example.



⟨dondl⟩ represents ⟨delimiter-or-non-delimiter⟩, ⟨ndl⟩ represents ⟨non-delimiter⟩,

⟨id⟩ represents ⟨identifier⟩, and ⟨cnst⟩ represents ⟨constant⟩.

A broken connecting line indicates the omission of one or more nodes.

Figure 9b. The section of the ⟨delimiter-or-non-delimiter-list⟩ shown in Figure 8a after removal of blanks.

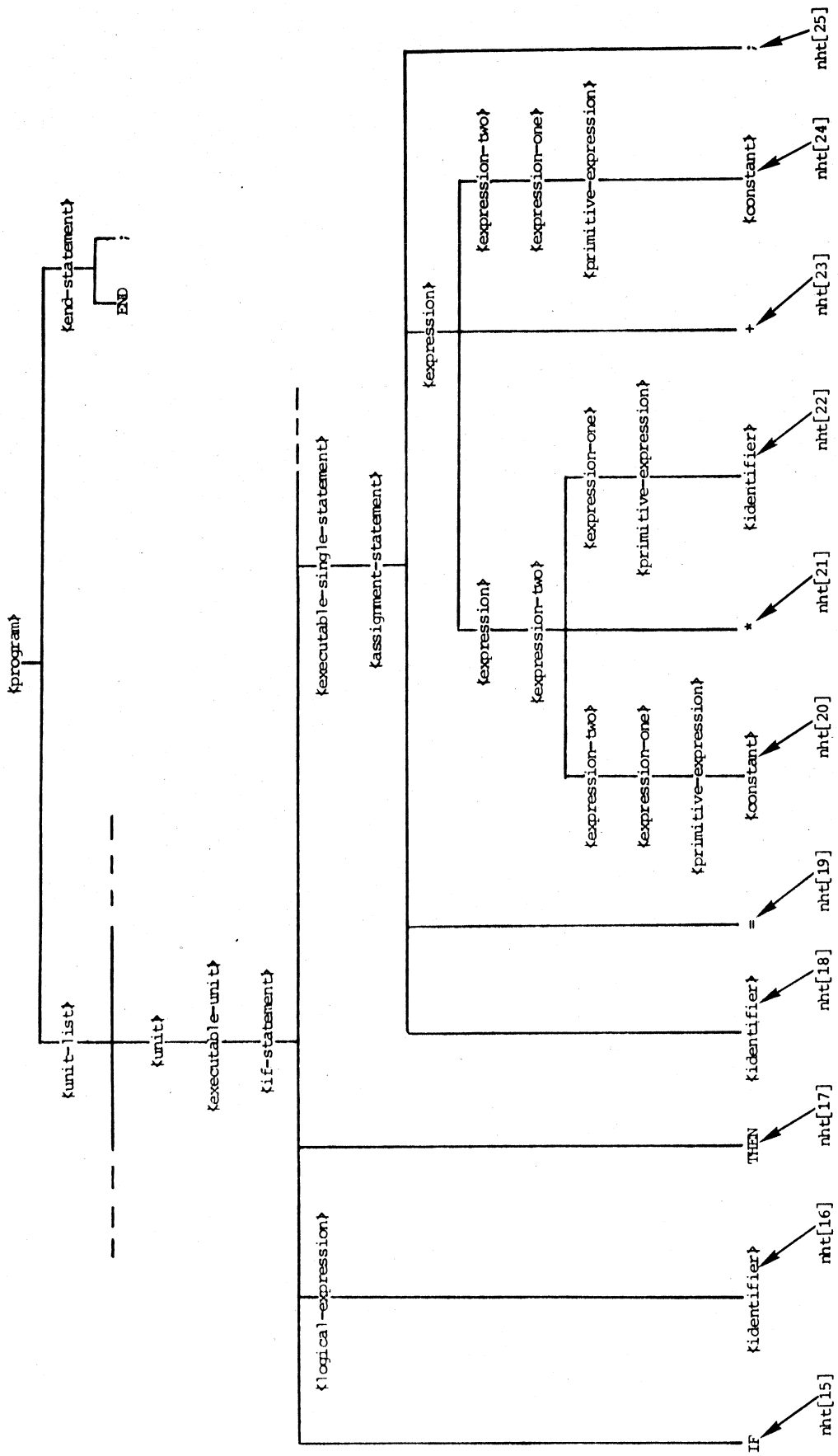


Figure 10. The part of the ⟨program⟩ tree that matches the ⟨delimiter-or-non-delimiter-list⟩ shown in Figure 8b.

in Figure 11. This is done by the three cases of Step 4. Case 4.1 applies where an element of nht designates a keyword. For example nht[15] designates the keyword IF and nt[15] designates an ⟨identifier⟩ containing IF. Case 4.2 applies where elements of nht designate ⟨identifier⟩ and ⟨constant⟩ nodes. For example, nht[18] designates an ⟨identifier⟩ and nht[20] designates a ⟨constant⟩. For these nodes, the ⟨program⟩ tree is completed by copying the details from the ⟨delimiter-or-non-delimiter-list⟩, in these two cases, copying the sybtrees of the nodes designated by nt[18] and nt[20] respectively. Case 4.3 ensures a match between delimiters, for example nht[22] and nt[22] both designate an asterisk.

The ⟨program⟩ tree shown in Figure 11 is a section of the complete concrete program of the running example. Once it has been constructed, the ⟨program⟩ tree is attached to the ⟨translation-state⟩ as indicated in Figure 12.

10. THE ABSTRACT SYNTAX

The abstract syntax deliberately bears a strong resemblance to the corresponding parts of the concrete syntax. The relationship between these parts is intended to be intuitively obvious. The main difference is that those parts of the concrete syntax whose only function is in the written form of the program have been omitted in the abstract syntax.

Syntax for programs

A1. <program> ::= [<declaration-list>] [<executable-unit-list>]

Syntax for declarations

A2. <declaration> ::= <identifier> <attribute>

A3. <attribute> ::= <fixed> | <bit>

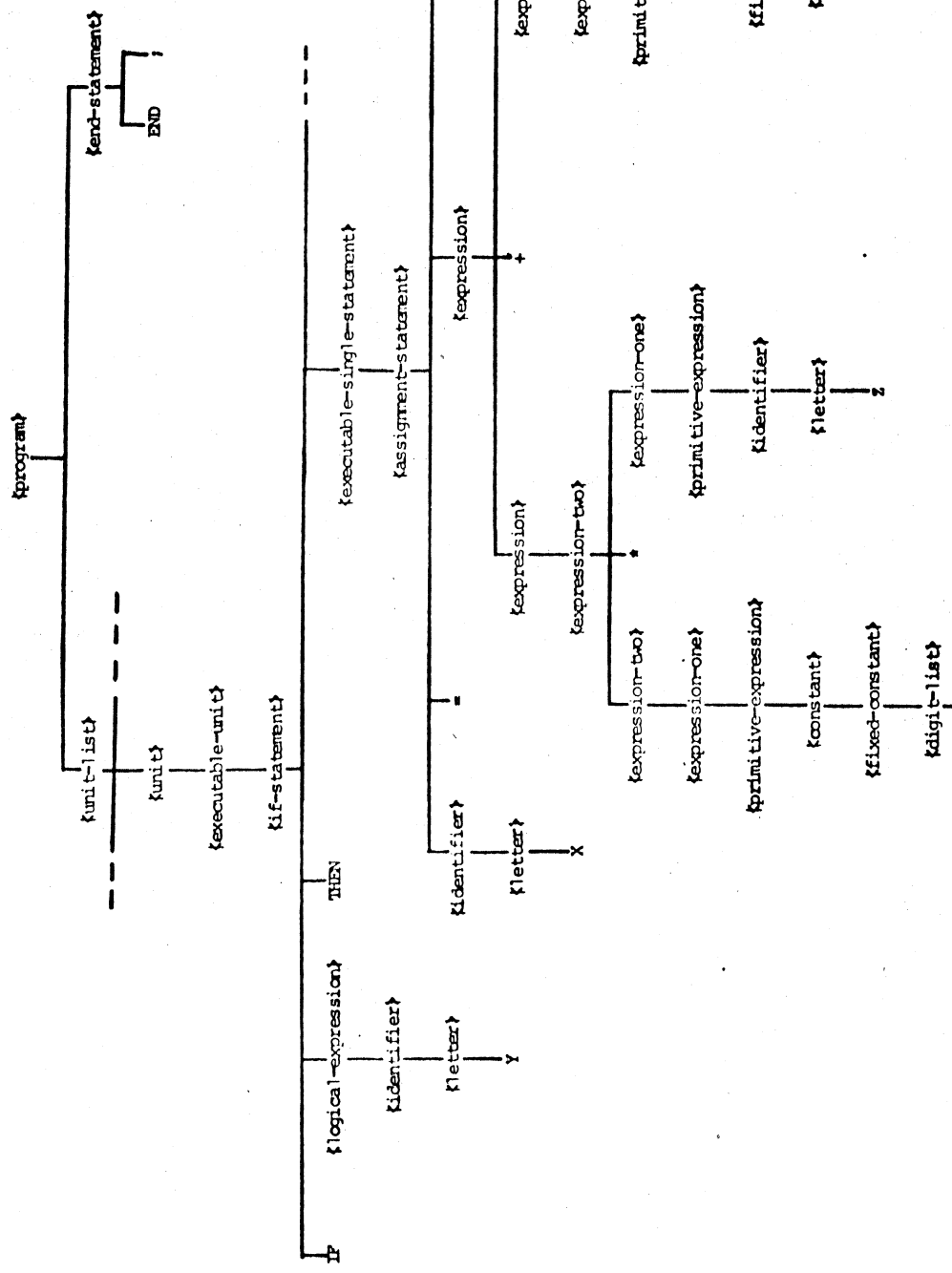


Figure 11. Section of completed ⟨program⟩ tree matching the part of the ⟨delimiter-or-non-delimiter-list⟩ shown in Figure 8b.

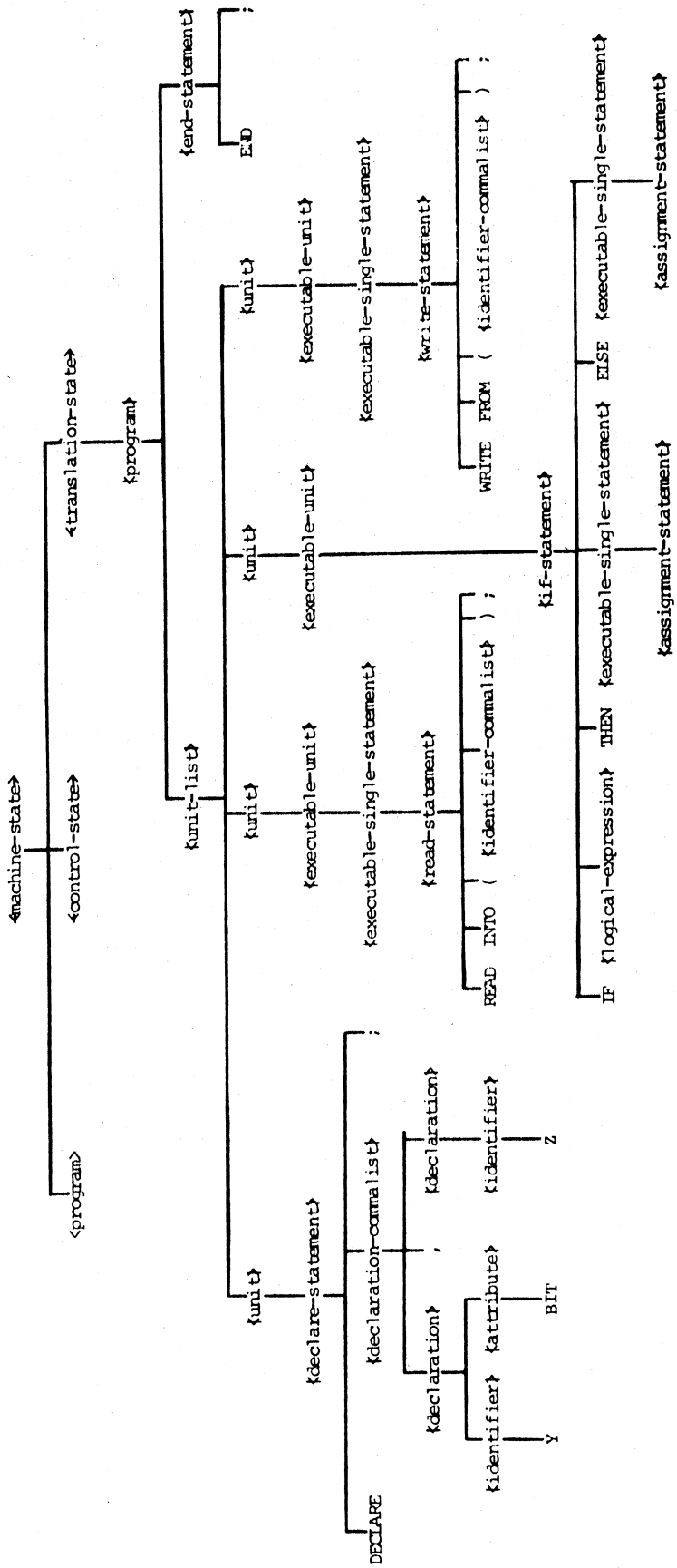


Figure 12. The <machine-state> containing a partial <program> tree for the Running Example.

Syntax for executable-units

A4. <executable-unit> ::= [<statement-name>]
{<if-statement> | <single-statement>}

A5. <statement-name> ::= <identifier>

Syntax for if-statements

A6. <if-statement> ::= <logical-expression> <then-unit> [<else-unit>]

A7. <then-unit> ::= <single-statement>

A8. <else-unit> ::= <single-statement>

Syntax for single-statements

A9. <single-statement> ::= <assignment-statement>
| <goto-statement>
| <read-statement>
| <return-statement>
| <write-statement>

A10. <assignment-statement> ::= <variable-reference> <expression>

A11. <goto-statement> ::= <executable-unit-designator>
| <identifier>

A12. <read-statement> ::= <variable-reference-list>

A13. <write-statement> ::= <variable-reference-list>

Syntax for expressions

A14. <logical-expression> ::= <expression> {<eq> | <ne>} <expression>
| <variable-reference>

A15. <expression> ::= <variable-reference>
| <constant>
| <infix-expression>
| <prefix-expression>

A16. <infix-expression> ::= <expression> {<add> | <multiply>} <expression>

A17. <prefix-expression> ::= <minus> <expression>

Syntax for references

A18. <variable-reference> ::= <declaration-designator>

A19. <identifier> ::=

<identifier> is defined as a ⟨symbol-list⟩ corresponding to the sequence of characters in ⟨identifier⟩.

Syntax for constants

A20. <constant> ::= ⟨integer-value⟩
| ⟨bit-value⟩

An ⟨integer-value⟩ is a ⟨machine-state⟩ type, defined in Section 12, rule M13, that contains a single element of the set of integers. A ⟨bit-value⟩ is defined in rule M12 and contains one of the values ⟨true⟩ or ⟨false⟩.

11. THE TRANSLATOR (CONSTRUCTION PHASE)

The portion of the definition algorithm described in this section first expands the concrete tree by applying the defaults and then constructs the abstract program component of the ⟨machine-state⟩. During the construction, checks are made for context dependent errors.

Operation: translation-construction-phase

Step 1. Perform complete-concrete-program.
Step 2. Perform validate-concrete-declarations.
Step 3. Perform create-program.
Step 4. Delete the ⟨translation-state⟩ from the ⟨machine-state⟩.

11.1 Expanding the Concrete Tree

The operations of this section add components to the ⟨program⟩ corresponding to implicit declarations, attribute defaults and the terminal return statement.

Operation: complete-concrete-program

Step 1. Perform implicit-declaration.
Step 2. Perform attribute-default.
Step 3. Let ul be the ⟨unit-list⟩ of the ⟨program⟩. If the right-most ⟨executable-unit⟩ component of ul does not immediately contain

```
⟨executable-single-statement⟩:
   ⟨return-statement⟩;
```

then append to ul

```
⟨unit⟩:
   ⟨executable-unit⟩:
      ⟨executable-single-statement⟩:
         ⟨return-statement⟩
            RETURN
            ⟨;⟩.
```

Operation: implicit-declaration

Step 1. Let ul be the ⟨unit-list⟩ of the ⟨program⟩.
Step 2. For each ⟨identifier⟩, id, contained in an ⟨executable-unit⟩ of ul, perform Step 2.1.
Step 2.1. If id is not contained in a ⟨statement-name⟩ or ⟨goto-statement⟩ and if there is no ⟨declare-statement⟩ that contains id then attach to ul

```
⟨declare-statement⟩:
   DECLARE
   ⟨declaration-commalist⟩:
      ⟨declaration⟩:
         id ;;
   ⟨;⟩.
```

Operation: attribute-defaults

Step 1. Let ul be the ⟨unit-list⟩ of the ⟨program⟩.
Step 2. For each ⟨declaration⟩, d contained in ul, perform Step 2.1.
Step 2.1. If d does not contain an ⟨attribute⟩ then attach FIXED to d.

11.2 Analyzing Declarations

The operation in this section checks that no identifier is declared more than once.

Operation: validate-concrete-declarations

Step 1. The ⟨program⟩ must not contain two or more ⟨declaration⟩ nodes whose ⟨identifier⟩ components are equal.
Step 2. The ⟨program⟩ must not contain a ⟨declaration⟩ that has an ⟨identifier⟩ that is equal to an ⟨identifier⟩ contained in a ⟨statement-name⟩.

11.3 Building the Abstract Tree

The operations of this section construct and attach to the abstract <program> an abstract <executable-unit> or <declaration-unit> corresponding to each <unit>. Declarations are translated before executable-units to facilitate the building of designator nodes. Since a <goto-statement> may contain a forward reference, the final operation is to resolve the statement-name references in the <goto-statement> and replace them by <executable-unit-designator>s.

Operation: create-program

Step 1. Let ul be the ⟨unit-list⟩ contained in the ⟨program⟩.
Step 2. For each ⟨declaration⟩, d, contained in ul, perform create-abstract-declaration(d).
Step 3. For each ⟨executable-unit⟩, eu, contained in ul, taken in left-to-right order, perform construct-abstract-statement(e u).
Step 4. Perform complete-gotos.

Operations for declarations

Operation: create-abstract-declaration(cd)

where: cd is a ⟨declaration⟩

Step 1. Let cid be the ⟨identifier⟩ of cd. Perform create-identifier(cd) to obtain an <identifier>, id.
Step 2. If cd contains FIXED then let atr be <fixed>, otherwise let atr be <bit>.
Step 3. Append a

```
<declaration>:
   id
   <attribute>:
      atr.
```

to the <program>.

Operations for executable units

Operation: construct-abstract-statement(ce)

where: ce is an ⟨executable-unit⟩

Step 1. If ce immediately contains an ⟨executable-single-statement⟩, ess, then let cxs be the immediate component of ess, otherwise let cxs be the immediately contained ⟨if-statement⟩ of ce.
Step 2. Perform create-xxx-statement(cxs) to obtain an <xxx-statement>, axs, where cxs is an ⟨xxx-statement⟩.
Step 3. Let eu be an <executable-unit>. Attach axs to eu.
Step 4. If ce contains a ⟨statement-name⟩ then perform Steps 4.1 and 4.2.
Step 4.1. Let cid be the ⟨identifier⟩ immediately contained by the ⟨statement-name⟩ of ce. Perform create-identifier(cid) to obtain an <identifier>, id.
Step 4.2. Attach id to eu.
Step 5. Append eu to the <program>.

Operation: create-if-statement(cif)

where: cif is an ⟨if-statement⟩

result: an <if-statement>

Step 1. Let cle be the ⟨logical-expression⟩ contained in cif. Perform create-logical-expression(cle) to obtain a <logical-expression>, alx.
Step 2. Let ess be the leftmost ⟨executable-single-statement⟩ contained in cif. Let cxs be the ⟨xxx-statement⟩ contained in ess. Perform create-xxx-statement(cxs) to obtain an <xxx-statement>, axs.
Step 3. Let aif be

```
<if-statement>:
   alx
   <then-unit>:
      <single-statement>:
         axs.
```

Step 4. If cif contains ELSE then perform Steps 4.1 and 4.2.
Step 4.1. Let ess be the rightmost ⟨executable-single-statement⟩ contained in cif. Let cxs be the ⟨xxx-statement⟩ contained in ess. Perform create-xxx-statement(cxs) to obtain an <xxx-statement>, axs.
Step 4.2. Attach an

```
<else-unit>:
   <single-statement>:
      axs;;
```

to aif.
Step 5. Return aif.

Operations for single statements

Operation: create-assignment-statement(cas)

where: cas is an ⟨assignment-statement⟩

result: an <assignment-statement>

Step 1. Let id and cx be respectively the immediately contained ⟨identifier⟩ and ⟨expression⟩ of cas.
Step 2. Perform find-abstract-declaration(id) to obtain a <declaration-designator>, dd.
Step 3. Perform create-expression(cx) to obtain an <expression>, ax.
Step 4.
Case 4.1. ax immediately contains a <variable-reference>, vr.
The <attribute>contained by the <declaration> designated by dd must equal the <attribute> contained by the <declaration> designated by the <declaration-designator> of vr.
Case 4.2. ax immediately contains a <constant>, c.
If c contains an ⟨integer-value⟩ then the <declaration> designated by dd must contain <fixed>, otherwise it must contain <bit>.
Case 4.3. (Otherwise).
The <declaration> designated by dd must contain <fixed>.
Step 5. Return an

```
<assignment-statement>:
   <variable-reference>:
      dd;
   ax.
```

Operation: create-goto-statement(cgs)

where: cgs is a ⟨goto-statement⟩

result: a <goto-statement>

Step 1. Let cid be the ⟨identifier⟩ contained in cgs and perform create-identifier(cid) to obtain an <identifier>, id.
Step 2. Return a

```
<goto-statement>:
   id.
```

Operation: create-read-statement(crs)

where: crs is a ⟨read-statement⟩

result: a <read-statement>

Step 1. Let ars be a <read-statement>.
Step 2. Let idl be the ⟨identifier-commalist⟩ of crs.
Step 3. For each ⟨identifier⟩, id, of idl, taken in left-to-right order, perform Steps 3.1 and 3.2.
Step 3.1. Perform find-abstract-declaration(id) to obtain a <declaration-designator>, dd.
Step 3.2. Append <variable-reference>: dd; to ars.
Step 4. Return ars.

Operation: create-return-statement(crs)

where: crs is a ⟨return-statement⟩

result: a <return-statement>

Step 1. Return a <return-statement>.

Operation: create-write-statement(cws)

where: cws is a ⟨write-statement⟩

result: a <write-statement>

Step 1. Let aws be a <write-statement>.
Step 2. Let idl be the ⟨identifier-commalist⟩ of cws.
Step 3. For each ⟨identifier⟩, id, of idl, taken in left-to-right order, perform Steps 3.1 and 3.2.
  Step 3.1. Perform find-abstract-declaration(id) to obtain a <declaration-designator>, dd.
  Step 3.2. Append <variable-reference>: dd; to aws.
Step 4. Return aws.

Operations for expressions

Operation: create-logical-expression(clx)

where: clx is a ⟨logical-expression⟩

result: a <logical-expression>

Case 1. clx immediately contains an ⟨identifier⟩, id.

  Step 1.1. Perform find-abstract-declaration(id) to obtain a <declaration-designator>, dd. The <declaration> designated by dd must contain <bit>.
  Step 1.2. Return
    <logical-expression>:
      <variable-reference>:
        dd.

Case 2. clx has three components, cx1, op, and cx2, in left to right order.
  Step 2.1. Perform create-operand(cx1) to obtain the <expression>, ax1.
    Perform create-operand(cx2) to obtain the <expression>, ax2.
  Step 2.2. If op is = then let aop be <eq>, otherwise let aop be <ne>.
  Step 2.3. Return
    <logical-expression>:
      ax1
      aop
      ax2.

Operation: create-expression(cx)

where: cx is an {expression}, {expression-two}, {expression-one}, or {primitive-expression}.

result: an <expression>.

Case 1. cx is an {expression}, {expression-two}, or {expression-one} and cx has only one component, cxc.
Perform create-expression(cxc) to obtain an <expression>, ax. Return ax.

Case 2. cx is an {expression}, or {expression-two} and has three components, cx1, copn, and cx2, in left-to-right order.
Step 2.1. Perform create-operand(cx1) to obtain an <expression>, ax1.
Perform create-operand(cx2) to obtain an <expression>, ax2.
Step 2.2. If copn is + then let aopn be <add> otherwise let aopn be <multiply>.
Step 2.3. Return an

<expression>:
ax1
aopn
ax2.

Case 3. cx is an {expression-one} with two components, copn and cx1, taken in left-to-right order.
Perform create- operand(cx1) to obtain an <expression>, ax. Return an

<expression>:
<prefix-expression>:
<minus>
ax.

Case 4. cx is an {expression-one} with three components, c1, cx1, and c2, taken in left-to-right order.
Perform create-operand(cx1) to obtain an <expression>, ax. Return ax.

Case 5. cx is a {primitive-expression} and contains an {identifier}, id.
Perform find-abstract-declaration(id) to obtain a <declaration-designator>, dd.
Return an

<expression>:
<variable-reference>:
dd.

Case 6. cx is a {primitive-expression} and contains a {constant}, c.
Perform create-constant(c) to obtain a <constant>, ac. Return an

<expression>:
ac.

Operation: create-operand(cx)

where: cx is an {expression}, {expression-two}, {expression-one}, or {primitive-expression}.

result: an <expression>

Step 1. If cx is a {primitive-expression} then perform Step 1.1.
Step 1.1.
Case 1.1.1. cx immediately contains {identifier}, id.

Perform find-abstract-declaration(id) to obtain a <declaration-designator>, dd. The <declaration> designated by dd must contain <fixed>.

Case 1.1.2. cx immediately contains ⟨constant⟩, cn.
The ⟨constant⟩, cn must not contain ⟨bit-donst⟩.
Step 2. Perform create-expression(cx) to obtain an <expression>, ar.
Return ar.

Utility operations

Operation: create-identifier(cid)

where: cid is an ⟨identifier⟩

result: an <identifier>

Step 1. Return an <identifier> whose concrete representation is the same as that of cid.

Operation: find-abstract-declaration(cid)

where: cid is an ⟨identifier⟩

result: a <declaration-designator>

Step 1. Perform create-identifier(cid) to obtain an <identifier>, id.
Step 2. Let dl be the <declaration-list> contained in the <program>.
Step 3. Let dd be a <declaration-designator> for the <declaration containing id.
Step 4. Return dd.

Operation: create-constant(cc)

where: cc is a ⟨constant⟩.

result: a <constant>

Case 1. cc immediately contains ⟨bit-const⟩, bc.
If bc contains 1B then let abv be ⟨true⟩, otherwise let abv be ⟨false⟩. Return a

<constant>:
⟨bit-const⟩:
abv.

Case 2. cc immediately contains a ⟨digit-list⟩, dl.
Let iv be an ⟨integer-value⟩ equal to the value obtained by interpreting the ⟨digit⟩s of dl in left-to-right order as a decimal integer.
Return a

<constant>:
iv.

Operation for goto cleanup

Operation: complete-gotos

Step 1. Let eul be the <executable-unit-list> immediately contained in the <program>.
Step 2. There must not be two or more equal <statement-name> components of eul.
Step 3. For each <goto-statement>, g, contained in eul perform Steps 4.1 through 4.3.
  Step 4.1. Let id be the <identifier> contained in g.
  Step 4.2. There must exist in eul a <statement-name>, sn, which contains an <identifier> equal to id.
  Step 4.3. Replace id by the <executable-unit-designator> that designates the <executable-unit> containing sn.

11.4 Application to the Running Example

The first stage of this phase of the Translator is to complete the concrete program by constructing a DECLARE statement for any variables that were not declared in the original program. The variable X in the running example was not declared and a declaration with no specified attribute is constructed for it. The FIXED attribute is then included in any DECLARE statement without an attribute specified. The FIXED attribute is therefore added to the ⟨declaration⟩ for X just constructed and to the ⟨declaration⟩ for Z which had no attribute specified in the source program. Finally, if there is no final RETURN statement in the program, one is constructed and appended to the ⟨unit-list⟩. The result of completing the concrete program for the Running Example is shown in Figure 13 as a partial ⟨program⟩ tree. This may be compared with the partial tree shown in Figure 12.

The second stage is the construction of the abstract program from the concrete program. The <program> for the running example is shown in Figure 14. This will form the <program> component of the ⟨machine-state⟩ during the interpretation phase of the definition. In Figure 14, unique-names are represented as we have done before by means of circled numbers. Designators are shown by arrows pointing to copies of unique-names.

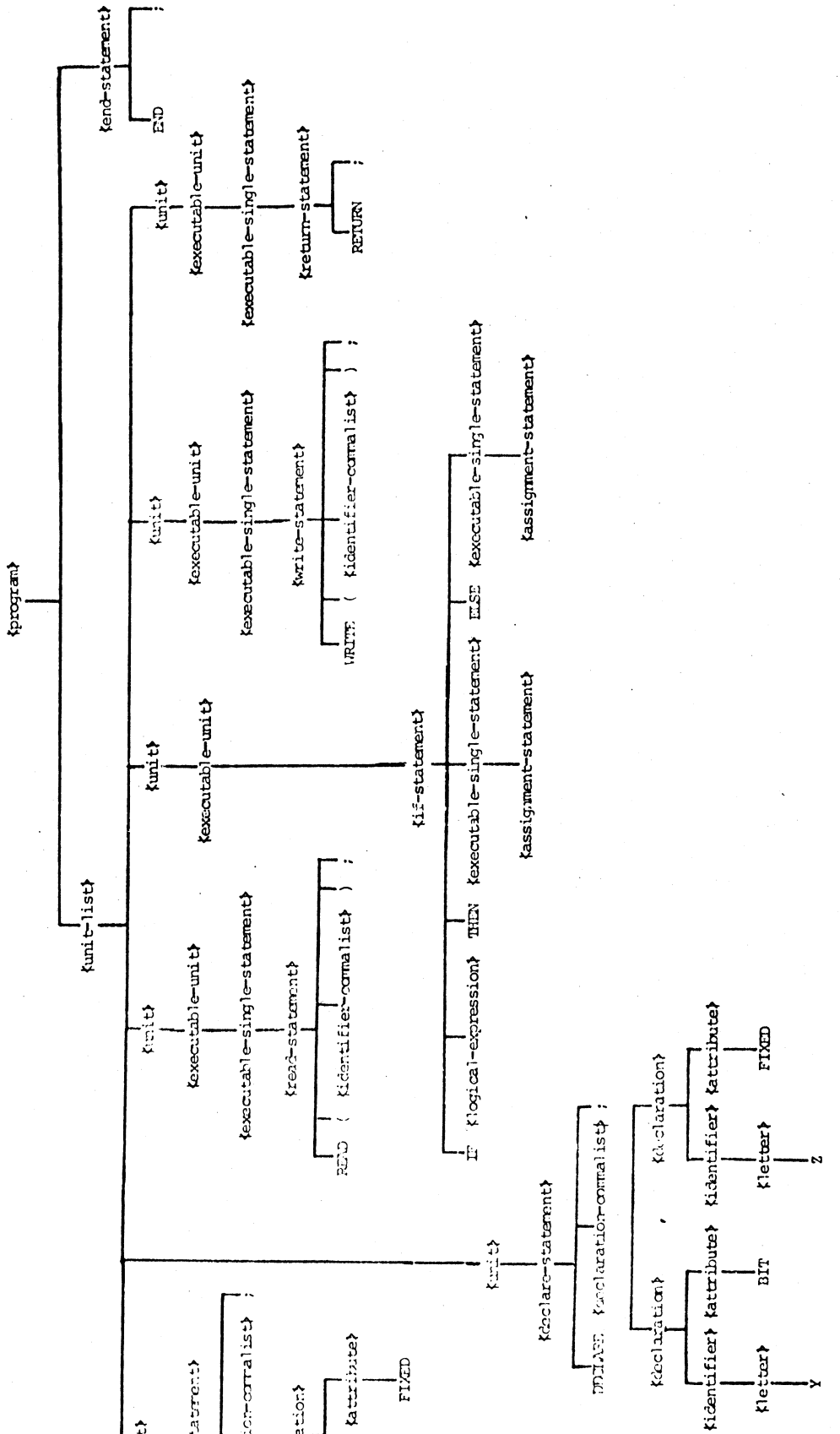


Figure 13. Completed concrete program for the Running Example.

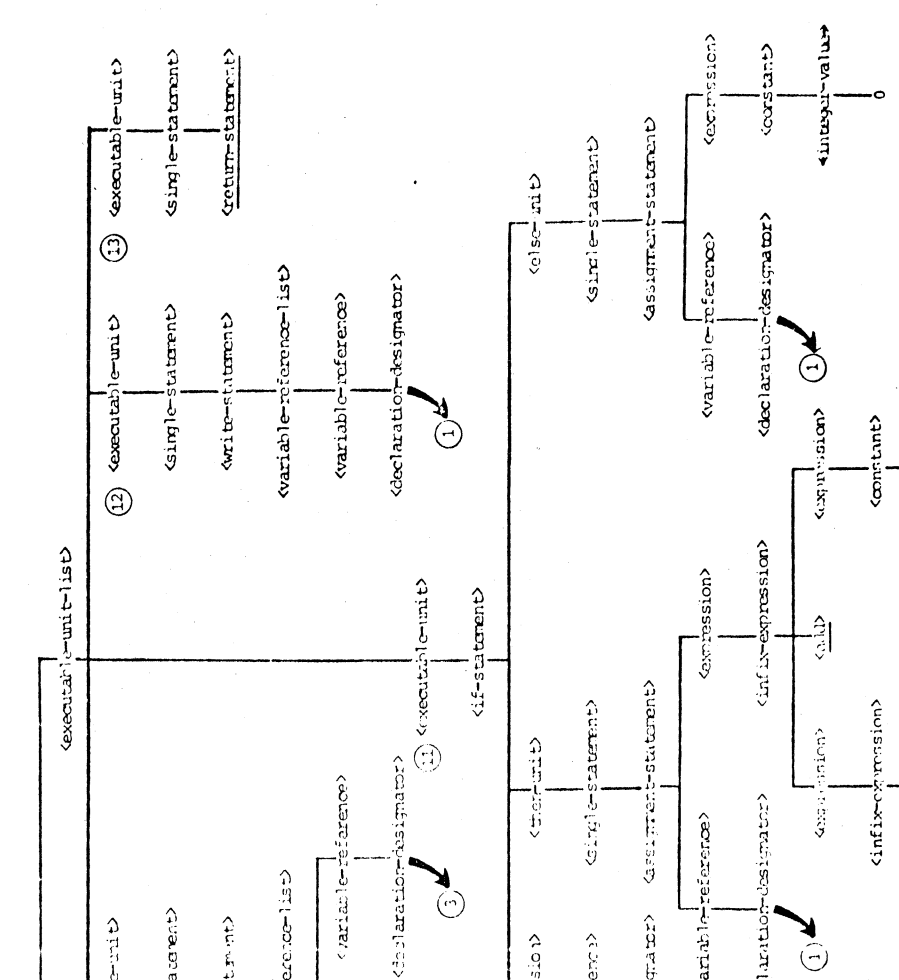


Figure 14. Abstract program corresponding to the Running Example.

12. THE MACHINE-STATE SYNTAX

This part of the <machine-state> syntax rules describes the <interpretation-state>. The productions for <machine-state> and <translation-state>, rules M1 through M4, were given in section 7.1

M5. <interpretation-state> ::= [<program-state>] <datasets>

Syntax for program-state

M6. <program-state> ::= <program-control> <storage-state>

M7. <program-control> ::= <executable-unit-designator> <operation-list>

M8. <storage-state> ::= <storage-directory> <allocated-storage>

M9. <storage-directory> ::= <storage-directory-entry-list>

M10. <storage-directory-entry> ::= <identifier> <basic-value-designator>

Syntax for allocated storage

M11. <allocated-storage> ::= <basic-value-list>

M12. <basic-value> ::= <integer-value> | <bit-value> | <undefined>

M13. <integer-value> ::=

The terminal component of an <integer-value> is a single element from the set of integers.

M14. <bit-value> ::= <true> | <false>

Syntax for datasets

M15. <datasets> ::= <input-dataset> <output-dataset>

M16. <input-dataset> ::= <dataset> <current-position>

M17. <output-dataset> ::= <dataset>

M18. <dataset> ::= <alpha> [<dataset-value-list>] <omega>

M19. <current-position> ::= <designator>

M20. <dataset-value> ::= <integer-value> | <bit-value>

13. THE INTERPRETER

In this section, we describe the portion of the definition algorithm that defines the meaning of the ⟨program⟩ by interpreting the corresponding ⟨program⟩ constructed by the translator.

13.1 Initialization

First the data to be input is obtained and the ⟨program-state⟩ is initialized.

Operation: interpretation-phase

Step 1. Let ds be a
⟨dataset⟩:
⟨alpha⟩
⟨omega⟩.

Step 2. Obtain, from a source outside this definition, information to be used for input, constructed in the form of a ⟨basic-value-list⟩, bvl.

Step 3. If bvl exists then attach bvl to ds. Let dg be the ⟨designator⟩ that designates the ⟨alpha⟩ of ds. Let dsi be
⟨input-dataset⟩:
ds
⟨current-position⟩:
dg.

Step 4. Perform interpret(dsi).

Operation: interpret(dsi)

where: dsi is an ⟨input-dataset⟩

Step 1. Let dso be
⟨output-dataset⟩:
⟨dataset⟩:
⟨alpha⟩
⟨omega⟩

Step 2. Attach to the ⟨machine-state⟩ the tree
⟨interpretation-state⟩:
⟨datasets⟩:
dsi
dso.

Step 3. Perform activate-program.

Operation: activate-program

Step 1. Let eud be an <executable-unit-designator> that designates the first element of the <executable-unit-list> of the <program>.

Step 2. Attach to the <interpretation-state> the tree

```
<program-state>:
   <program-control>:
      eud;
   <storage-state>:
      <storage-directory>
      <allocated-storage>.
```

Step 3. For each <declaration>, d, perform Steps 3.1 and 3.2.

Step 3.1. Perform allocate to obtain a <basic-value-designator>, bvd.

Step 3.2. Let id be the <identifier> of d. Append

```
<storage-directory-entry>:
   id
   bvd.
```

to the <storage-directory>

Step 4. Append an <operation> for advance-execution to the <program-control>.

Note that the execution of Step 4 of this operation brings into existence the <operation-list> of <program-control>. For as long as this list exists, the rightmost <operation> is the active one and the execution of the rightmost <operation> of the <operation-list> of <control-state> is suspended. It will remain suspended until the <program-control> is deleted by the execution of either the execute-return-statement or abnormal-termination operation.

13.2 Statement Interpretation Control

These operations control the sequence of statement interpretation.

Operation: advance-execution

Step 1. Let eu the the <executable-unit> designated by the <executable-unit-designator> of the <program-state>.

Step 2. If eu immediately contains a <single-statement>, ss, then let st be the immediate component of ss. Otherwise, let st be the immediately contained <if-statement> of eu.

Step 3. Perform execute-xxx(st) where "xxx" is replaced by the sequence of symbols forming the type of st.

Step 4. Go to Step 1.

Operation: normal-sequence

Step 1. Let eul be the <executable-unit-list>. Let eu be the <executable-unit> of eul that is designated by the <executable-unit-designator>, eud, of the <program-state>.

Step 2. Let eud designate the <executable-unit> that immediately follows eu in eul.

13.3 Interpretation of Statements

There is an operation for each statement type.

Operation: execute-if-statement(st)

where: st is an <if-statement>

Step 1. Let le be the <logical-expression> immediate component of st. Perform evaluate-logical-expression(le) to obtain tv.

Step 2.

Case 2.1. tv is <true>.
Let ss be the <single-statement> component of the <then-unit> of st.

Case 2.2. tv is <false>.
If st does not contain an <else-unit> then perform normal-sequence and terminate this operation. Otherwise, let ss be the <single-statement> component of the <else-unit> of st.

Step 3. Perform execute-xxx(ss) where "xxx" is replaced by the sequence of symbols forming the type of ss.

Operation: execute-assignment-statement(st)

where: st is an <assignment-statement>

Step 1. Let xp be the <expression> of st. Perform evaluate-expression(xp) to obtain a <basic-value>, bv.

Step 2. Let vr be the immediately contained <variable-reference> of st. Perform assign(vr, bv).

Step 3. Perform normal-sequence.

Operation: execute-goto-statement(st)

where: st is a <goto-statement>

Step 1. Replace the <executable-unit-designator> of the <program-state> by a copy of the <executable-unit-designator> of st.

Operation: execute-read-statement(st)

where: st is a <read-staement>.

Step 1. For each <variable-reference>, vr, of the <variable-reference-list> of st, taken in left-to-right order perform Steps 1.1 through 1.3.

Step 1.1. The <dataset-value-list>, dvl, of the <input-dataset> must not be empty. If the <current-position>, cp, of the <input-dataset> designates <alpha> then let dv be the first element of dvl, otherwise let dv be the element of dvl that immediately follows the one designated by cp. This element must exist. Let cp designate dv.

Step 1.2. Let d be the <declaration> designated by the <declaration-designator> of vr. If dv contains an <integer-value> then d must contain <fixed>, otherwise d must contain <bit>.

Step 1.3. Perform assign(vr, bv).

Step 2. Perform normal-sequence.

Operation: execute-return-statement(st)

where: st is a <return-statement>.

Step 1. Delete the <program-control> from the <machine-state>.

Note that this causes the rightmost <operation> of the <operation-list> in the <control-state> to become the active operation.

Operation: execute-write-statement(st)

where: st is a <write-statement>.

Step 1. For each <variable-reference>, vr, of the <variable-reference-list> of st, taken in left-to-right order, perform Steps 1.1 through 1.4.

Step 1.1. Perform evaluate-variable-reference(vr) to obtain a <basic-value-designator>, bvd.

Step 1.2. Perform obtain-basic-value(bvd) to obtain a <basic-value>, bv. Let v be the immediate component of bv. Let dsv be a <dataset-value>: v.

Step 1.3. If the number of elements in the <dataset-value-list>, dvl, of the <output-dataset> is greater than some implementation-defined number, then perform abnormal-termination.

Step 1.4 Append dsv to dvl.

Step 2. Perform normal sequence.

13.4. Expression Evaluation

The following operations perform the evaluation of expressions and references.

Operation: evaluate-logical-expression(e)

where: e is a <logical-expression>

result: <true> or <false>

Case 1. e immediately contains a <variable-reference>, vr.
Perform evaluate-variable-reference(vr) to obtain a <basic-value-designator>, bvd. Perform obtain-basic-value(bvd) to obtain a <basic-value>, bv. Return the immediate component of bv.
Case 2. e immediately contains two <expression> components, e1 and e2.
Step 2.1. Perform evaluate-expression(e1) to obtain a <basic-value>, bv1, and evaluate-expression(e2) to obtain a <basic-value>, bv2.
Step 2.2.
Case 2.2.1. e immediately contains <eq>.
If bv1 = bv2 then return <true>, otherwise return <false>.
Case 2.2.2. e immediately contains <neq>.
If bv1 = bv2 then return <false>, otherwise return <true>.

Operation: evaluate-expression(e)

where: e is a <fixed-expression>

result: b <basic-value>

Case 1. e immediately contains a <variable-reference>, vr.
Perform evaluate-variable-reference(vr) to obtain a <basic-value-designator>, bvd. Perform obtain-basic-value(bvd) to obtain a <basic-value>, bv. Return bv.
Case 2. e immediately contains a <constant>, c.
Let bv be a <basic-value>: v where v is the immediate component of c. Return bv.
Case 3. e immediately contains an <infix-expression>, ix.
Step 3.1. Let e1 and e2 be the two immediately contained <expression> components of ix. Perform evaluate-expression(e1) to obtain the <basic-value>, bv1 and evaluate-expression(e2) to obtain the <basic-value>, bv2.
Step 3.2. If ix immediately contains <add> then let ir be the <integer-value> whose value is the sum of the two <integer-value> components of bv1 and bv2. Otherwise let ir be the <integer-value> whose value is the product of the two <integer-value> components of bv1 and bv2.
Step 3.3. If the magnitude of ir exceeds an implementation-defined maximum, then let ir be an <integer-value> with an implementation defined value and optionally perform abnormal-termination.
Step 3.4 Return <basic-value>: ir.

Case 4. e immediately contains a <prefix-expression>, px.

Step 4.1. Let e1 be the <expression> immediately contained by px. Perform evaluate-expression(e1) to obtain a <basic-value>, bv.

Step 4.2. Let ir be the <integer-value> whose value is -iv, where iv is the <integer-value> immediately contained by bv.

Step 4.3. If the magnitude of ir exceeds an implementation defined maximum then let ir be an implementation-defined value and optionally perform abnormal-termination.

Step 4.4 Return <basic-value>: ir.

13.5 Storage Manipulation

These two operations are the only operations that directly change the state of <allocated-storage>.

Operation: allocate

result: a <basic-value-designator>

Step 1. Let bv be a <basic-value>: <undefined>.
Step 2. Append bv to the <basic-value-list> of <allocated-storage>.
Step 3. Let bvd be a <basic-value-designator> that designates bv. Return bvd.

Operation: assign(vr, bv)

where: vr is a <variable-reference>
bv is a <basic-value>

Step 1. Perform evaluate-variable-reference(vr) to obtain a <basic-value-designator>, bvd.
Step 2. Replace the <basic-value> designated by bvd with a copy of bv.

13.6. Storage Reference

Operation: evaluate-variable-reference(vr)

where: vr is a <variable-reference>

result: a <basic-value-designator>.

Step 1. Let d be the <declaration> designated by the immediately contained <declaration-designator> of vr.
Step 2. Let id be the <identifier> of d. Let bvd be a copy of the <basic-value-designator> component of the <storage-directory-entry> that contains an <identifier> equal to id.
Step 3. Return bvd.

Operation: obtain-basic-value(bvd)

where: bvd is a ⟨basic-value-designator⟩

result: a ⟨basic-value⟩

Step 1. The ⟨basic-value⟩ designated by bvd must not contain ⟨undefined⟩.
Step 2. Return a copy of the ⟨basic-value⟩ designated by bvd.

13.7. Abnormal Termination

Operation: abnormal-termination

Step 1. Perform an implementation-defined action.
Step 2. Delete the ⟨program-control⟩ from the ⟨machine-state⟩.

The implementation-defined action permits the implementation to give some indication to the programmer of the reason why the program is being abnormally terminated. This operation also causes, by the deletion of the ⟨program-control⟩, the rightmost ⟨operation⟩ of the ⟨operation-list⟩ in the ⟨control-state⟩ to become the active operation again.

13.8 Application to the Running Example

To continue with our running example, we will suppose that the input file contains two values, the ⟨bit-value⟩: ⟨true⟩ and the ⟨integer-value⟩: 9. Figure 15 shows the ⟨machine-state⟩ just before the operation activate-program is performed. The ⟨current-position⟩ of the ⟨input-dataset⟩ designates the ⟨alpha⟩ marker at the start of the file and the ⟨output-dataset⟩ is empty.

After the operation activate-program has been completed, the ⟨interpretation-state⟩ is as shown in Figure 16. The ⟨allocated-storage⟩ contains three ⟨basic-value⟩ components, each initialized to ⟨undefined⟩, for the three variables, X, Y, and Z. The ⟨storage-directory⟩ contains entries designating these values. The ⟨executable-unit-designator⟩ has been set to designate

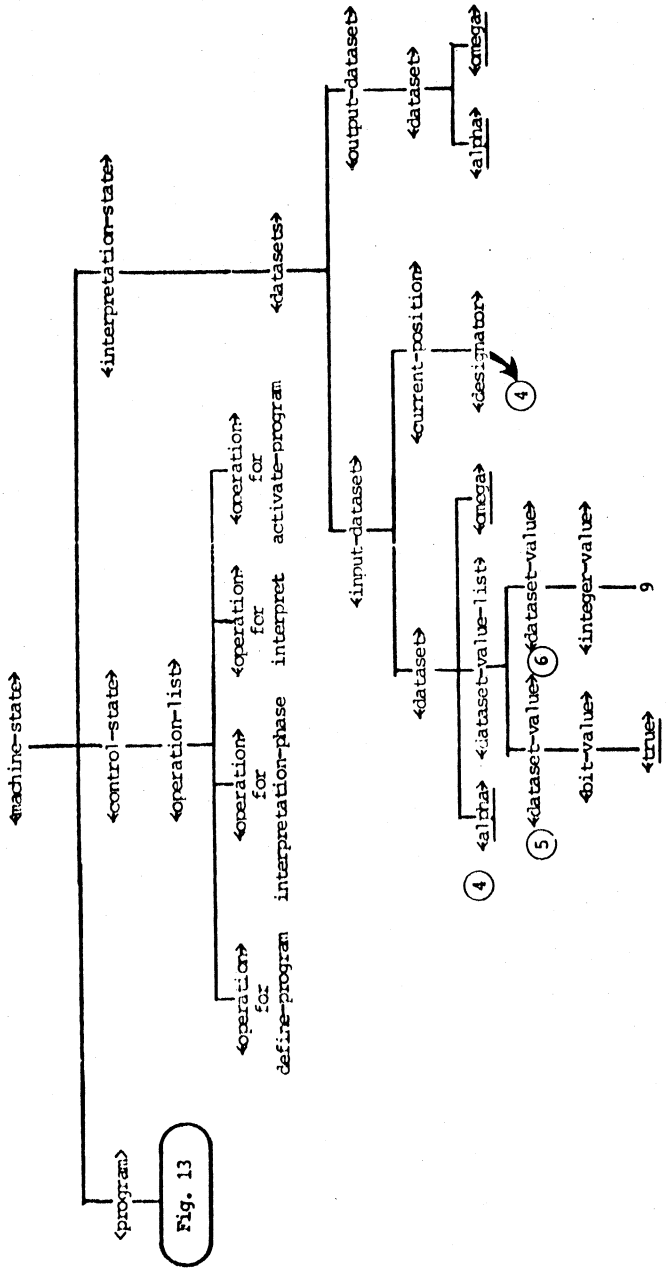


Figure 15. The <machine-state> just prior to interpreting the Running Example.

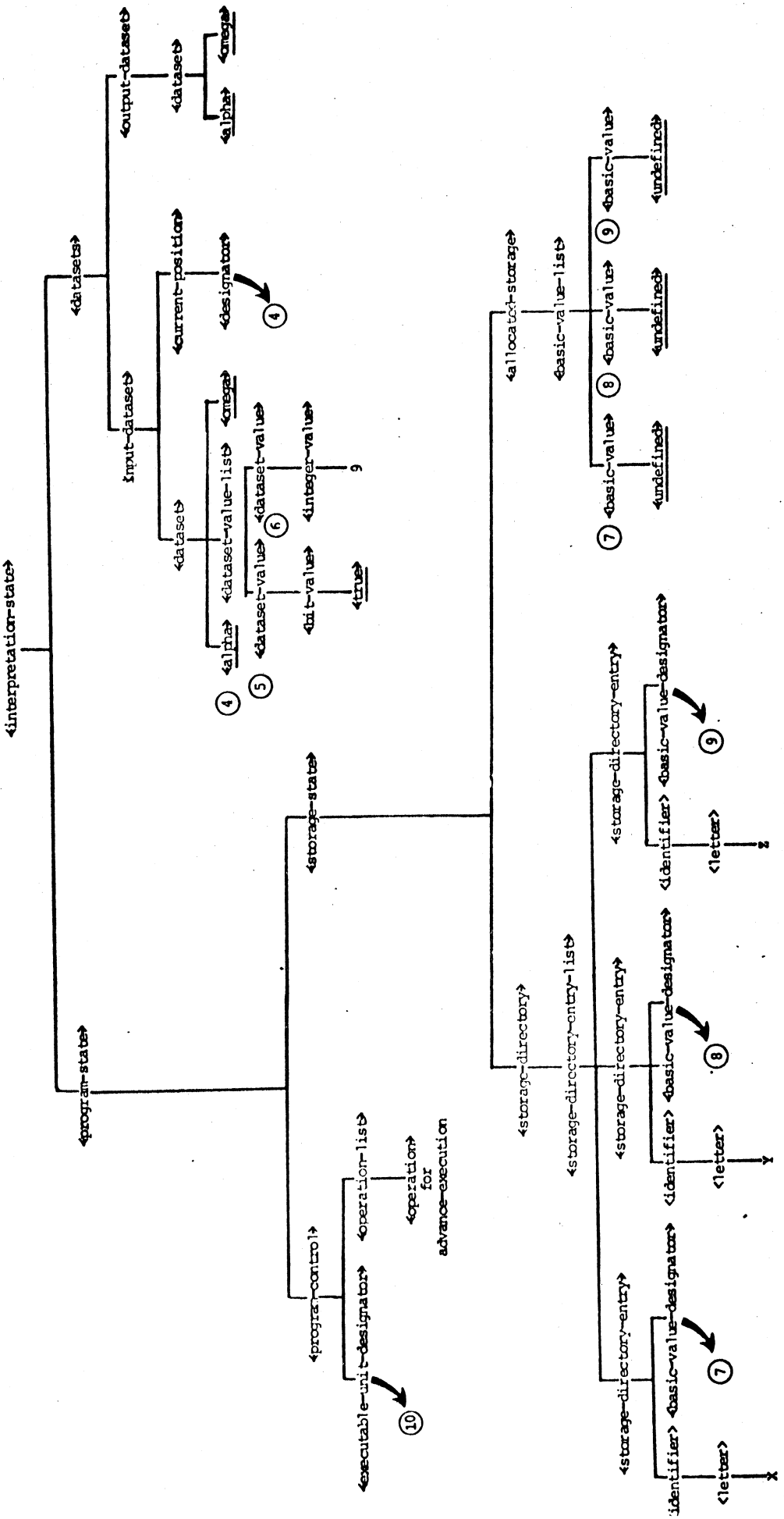


Figure 16. The <interpretation-state> just prior to interpreting the READ statement in the Running Example.

the first statement in the <program> (shown in Figure 14). The <program-control> contains an <operation> for advance-execution and this is now the active operation.

Following the execution of the first statement of the <program>, the READ statement, the <interpretation-state> is as shown in Figure 17. The <allocated-storage> has changed so that the <basic-value> components for the variables Y and Z now contain the values read from the <input-dataset>. The <basic-value> for X is unchanged, it is still <undefined>. The <current-position> of the <input-dataset> now designates the value just read from it and the <executable-unit-designator> has been advanced to the next statement.

Figure 18 shows the <interpretation-state> following the execution of the IF statement. The <basic-value> for the variable X has now received the <integer-value> 19 because the <then-unit> was executed since the <basic-value> for Y contained <true>. The <executable-unit-designator> has been advanced to designate the WRITE statement.

Execution of the WRITE statement causes a copy of the <integer-value> for the variable X to be appended to the <output-dataset>. Since the <output-dataset> was empty, a <dataset-value-list> was constructed by the append instruction. The <output-dataset> is as shown in Figure 19. At this point, the <executable-unit-designator> designates the RETURN statement constructed by the Translator. Execution of this statement causes the program to terminate.

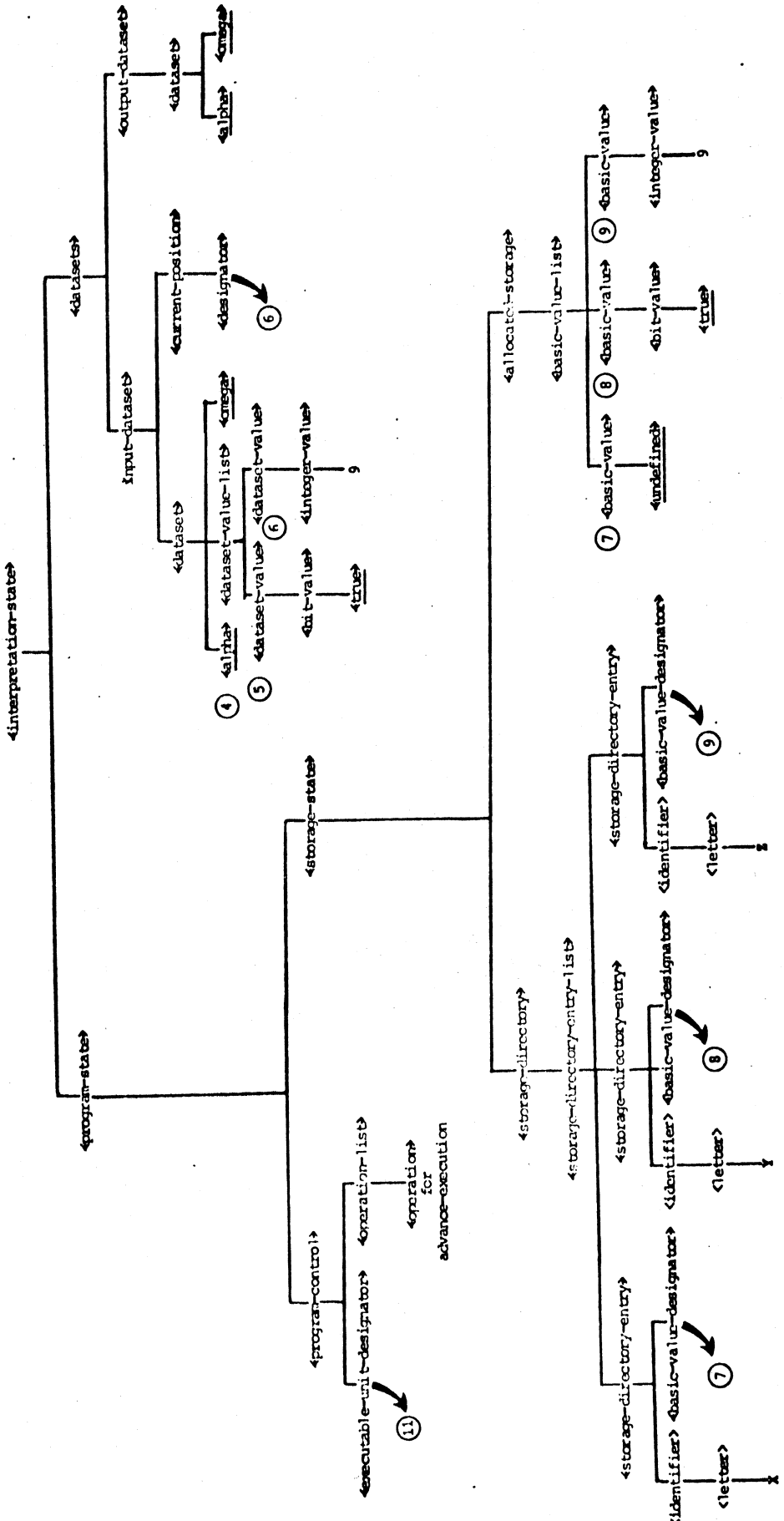


Figure 17. The <interpretation-state> just prior to interpreting the IF statement in the Running Example.

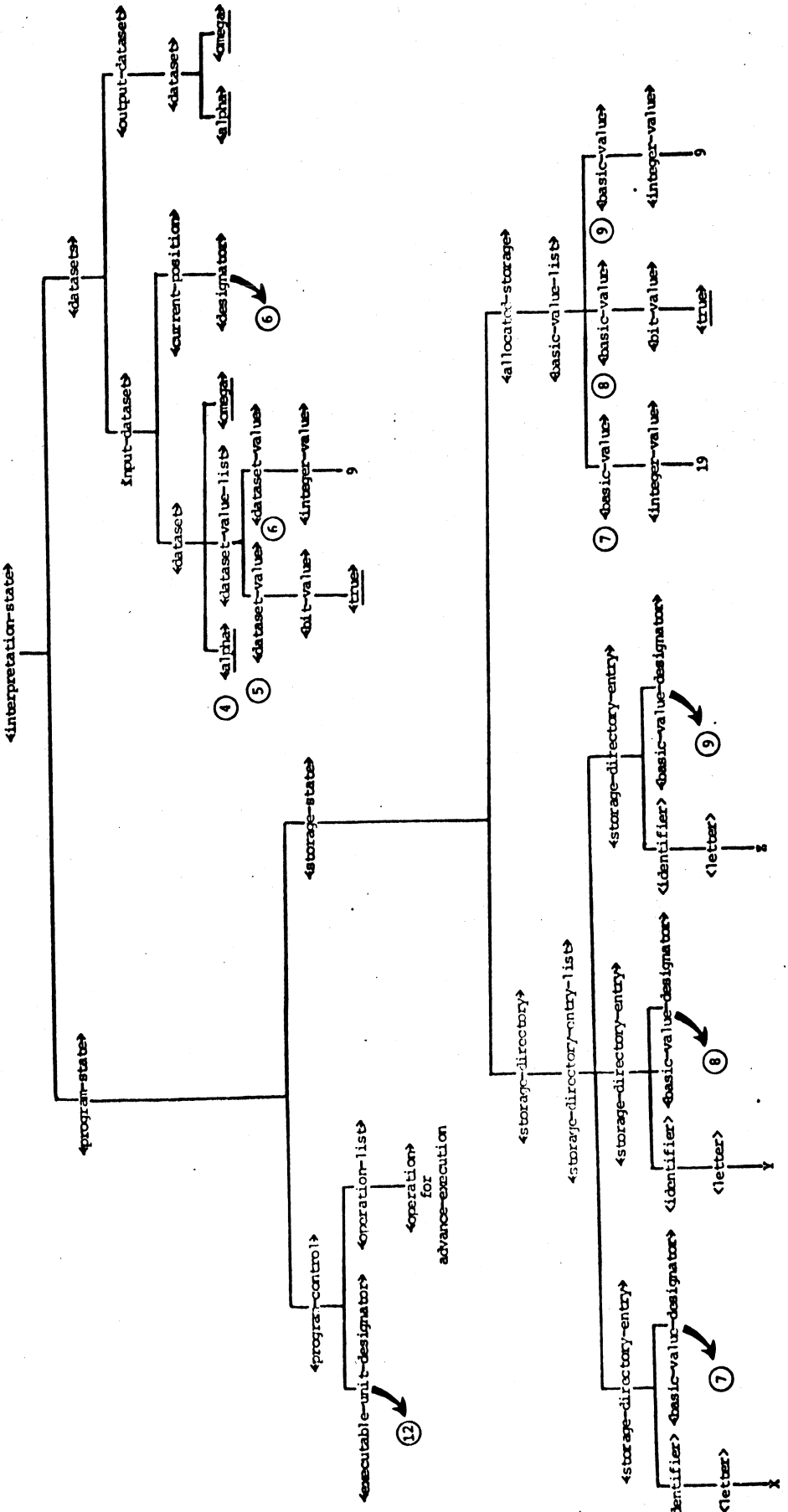


Figure 18. The <interpretation-state> just prior to interpreting the WRITE statement in the Running Example.

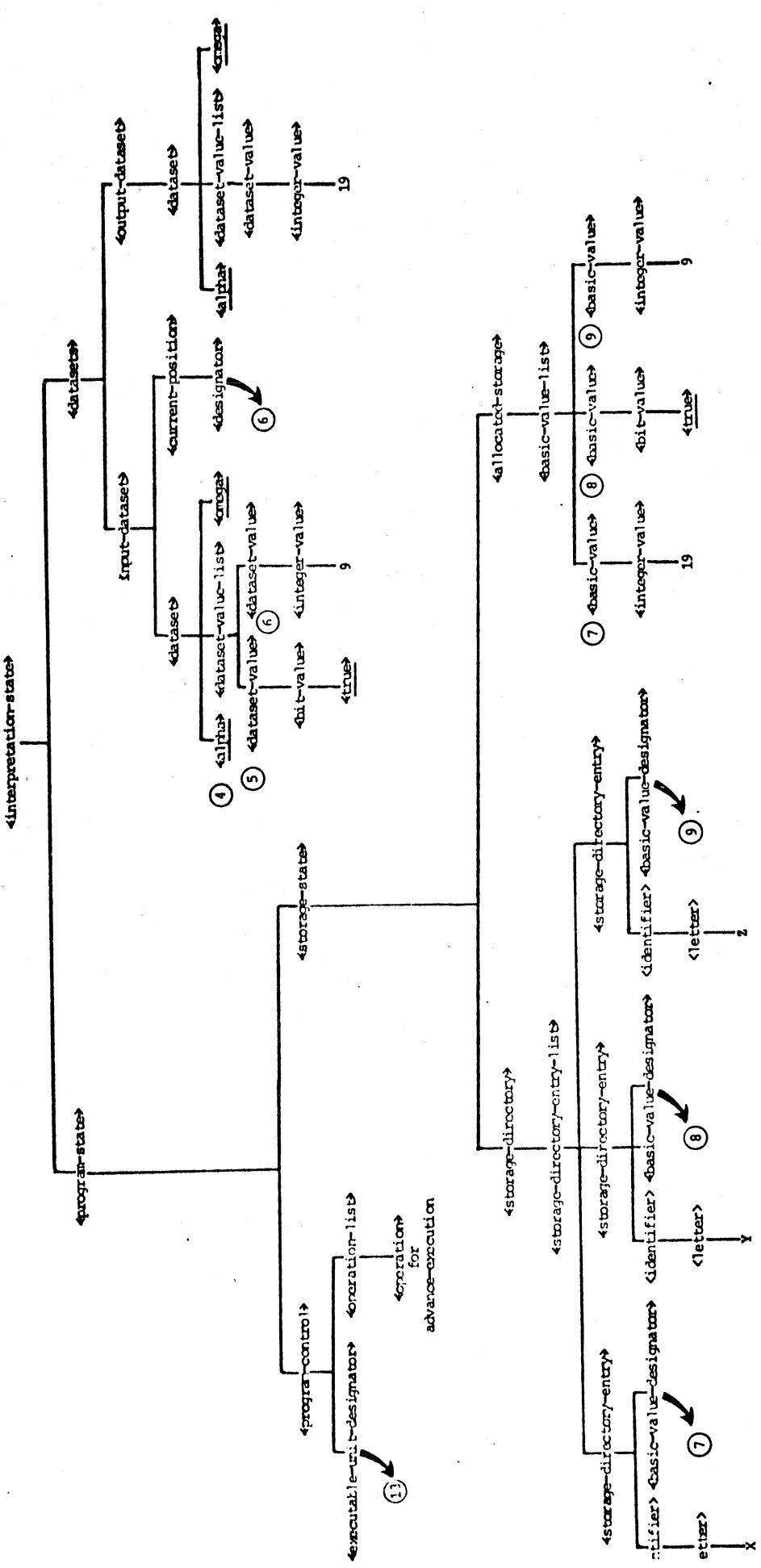


Figure 19. The ⟨interpretation-state⟩ just prior to interpreting the RETURN statement in the Running Example.

14. Postlude

This definition of the trivial language SAL has used the definition technique of BASIS/1; however, we have only introduced those terms and concepts that were required for specifying this small language. The reader who is using this as an introduction to BASIS/1 will find that, because of the much greater complexity of PL/I, some additional constructs have been needed for its definition. Nevertheless, the mechanism used here is essentially the same as that used BASIS/1 and a reading of the first chapter of BASIS/1 will serve to introduce the additional features of the metalanguage.

The BASIS/1 metalanguage is sufficiently powerful to give formal definitions for any sequential programming language, such as ALGOL 60, SNOBOL, LISP, or COBOL. However, since tasking is not considered in the BASIS/1 version of PL/I, the method in its current state is not suitable for defining non-deterministic or parallel programming languages such as ALGOL 68.

We would like to thank David Beech, John Kelly, Henry Ledgard, and Peter Wegner for their helpful comments on previous drafts of this paper.

15. REFERENCES

[A1] Aho, A. and Ullman, J. The Theory of Parsing, Translation and Compiling, Vol. 1 & 2. Prentice-Hall, Inc., Englewood Cliffs, N. J. 1973.

[A2] Alber, K., Oliva, P., and Urschler, H., Concrete Syntax of PL/I, IBM Laboratory Vienna, Technical Report TR 25.084 (June 1968).

[A3] Alber, K., and Oliva, P., Translation of PL/I into Abstract Text, IBM Laboratory Vienna, Technical Report TR 25.086 (June 1968).

[A4] Allen, C.D., Beech, D., Nicholls, J.E., and Rowe, R., An Abstract Interpreter of PL/I, Technical Note TN3004, IBM United Kingdom Laboratories Ltd, 1966.

[B1] Backus, J.W., "The Syntax and Semantics of the Proposed International Algebraic Language of the Zurich ACM-GAMM Conference", Proc. International Conf. on Information Processing, UNESCO (1959), pp125-132.

[B2] Beech, D., Rowe, R., Larner, R.A., and Nicholls, J.E., Concrete Syntax of PL/I, Technical Note TN3001, IBM United Kingdom Laboratories Ltd, 1966.

[B3] Beech, D., Nicholls, J.E., and Rowe, R., A PL/I Translator, Technical Note TN3003, IBM United Kingdom Laboratories Ltd, 1966.

[B4] Beech, D., Rowe, R., Larner, R.A., and Nicholls, J.E., Abstract Syntax of PL/I, Technical Note TN3002, IBM United Kingdom Laboratories, 1966.

[B5] Beech, D., "A Structural View of PL/I", Computing Surveys 2, 1 (March 1970) pp33-64.

[B6] Beech, D., and Marcotty, M., "Unfurling the PL/I Standard", SIGPLAN Notices 8, No. 10 (Oct 1973), pp12-43.

[B7] Beech, D., "On the Definitional Method of Standard PL/I", In Conference Records on the Principles of Programming Languages, Boston, Oct. 1973, pp87-94.

[E1] Elgot, C.C., and Robinson, A., "Random-access stored-program machines, an approach to programming languages", Journal of the ACM 11, No. 4, (1964), pp365-399.

[E2] European Computer Manufacturers' Association and American National Standards Institute, PL/I BASIS/1-12 Published as BSR X3.53, American National Standards Committee X3, Washington D.C. 1975.

[G1] Garwick, J.V., "The definition of programming languages by their compilers", In: Steel, T.B., Jr., Ed., Formal Language Description Languages for Computer Programming, North-Holland Publ. Co., Amsterdam, 1966, pp139-147.

[L1] Landin, P.J., "Correspondence between Algol 60 and Church's Lambda-Notation, Part I", Commun. of the ACM 8, No. 2, (1965) pp89-101.

[L2] Landin, P.J., "Correspondence between Algol 60 and Church's Lambda-Notation, Part II", Commun. of the ACM 8, No. 3, (1965), pp158-165.

[L3] Lee, J.A.N., Computer Semantics, Van Nostrand Reinhold Co., New York, 1972

[L4] Lucas, P., Laber, K., Bandat, K., Bekic, H., Oliva, P., Walk, K., and Zeisel, G., Informal Introduction to the Abstract Syntax and Interpretation of PL/I, IBM Laboratory Vienna, Technical Report TR 25.083 (June 1968).

[L5] Lucas, P., Lauer, P., and Stigleitner, H., Method and Notation for the Formal Definition of Programming Languages, IBM Laboratory Vienna, Technical Report TR 25.087 (June 1968).

[M1] McCarthy, J., "A formal description of a subset of ALGOL", In: Steel, T.B., Jr., Ed., Formal Language Description Languages for Computer Programming, North-Holland Publ. Co., Amsterdam, 1966, pp1-12.

[M2] McCarthy, J., "Towards a Mathematical Science of Computation", In: Proc. IFIP Cong. 1962, North-Holland Publ. Co., Amsterdam, 1963.

[S1] Steel, T.B., "Standards for Computers and Information Processing", in Alt & Rubinoff, Eds., Advances in Computers Vol. 8, Academic Press, New York, 1967, pp103-152.

[W1] Walk, K., Alber, K., Bandat, K., Bekic, H., Chroust, G., Kudielka, V., Oliva, P., and Zeisel, G., Abstract Syntax for Interpretation of PL/I IBM Laboratory Vienna, Technical Report TR 25.082 (June 1968).

[W2] Wegner, P., "Operational Semantics of Programming Languages", Proc. ACM Conf. on Proving Assertions about Programs, SIGPLAN Notices 7, 1 (Jan. 1972), pp128-141.

[W3] Wegner, P., "The Vienna Definition Language", Computing Surveys 4, 1 (March 1972), pp5-63

[W4] van Wijngaarden, A., (ed), et al. "Report on the Algorithmic Language ALGOL 68", Numerische Mathematik 14, 2 (1969), pp79-218.