A Sampler of Formal Definitions*

MICHAEL MARCOTTY**

and

HENRY F. LEDGARD

Department of Computer and Information Science, University of Massachusetts, Amherst, Massachusetts 01002

GREGOR V. BOCHMANN

Département d'Informatique, Université de Montréal, Montréal 101, Canada

From a purely scientific viewpoint, the members of the various working groups concerned with programming language standardization really ought to report to their parent committees that their assigned task is impossible without a major prior effort by the technical community; and that this prior effort would have to produce an effective procedure for describing the languages that are of concern.

Thomas B. Steel, Jr., 1967 [S4]

The current use of formal definitions of programming languages is very limited, largely because of a lack of fully developed techniques and because of user resistance to the poor human engineering of the definitions themselves. Nevertheless, usable formal definitions are essential for the effective design of programming languages and their orderly development and standardization.

We present four well-known formal definition techniques: W-grammars, Production Systems with an axiomatic approach to semantics, the Vienna Definition Language, and Attribute Grammars. Each technique is described tutorially and examples are given; then each technique is applied to define the same small programming language.

These definitions provide a usable basis for a critical discussion of the relative clarity of the different methods. This leads to a review of some of the debatable issues of formal definition. Among these issues are the advantages, if any, to the use of an underlying machine model, the precise nature of a valid program, the relative merits of generative and analytic definitions, and the place of implementation-defined features in the definition.

Finally, a case is made for the importance of formal definitions and the need for a significant effort to make definitions suitable for human comprehension.

Keywords and Phrases: formal definition, language design, programming languages, syntax, semantics, human engineering, W-Grammars, Attribute Grammars, Production Systems, axiomatic definition, Vienna Definition Language

CR Categories: 4.13, 4.22, 5.21, 5.23, 5.24, 5.26

* This work was supported in part by the US Army Research Office and in part by the National Research Council of Canada.

** Work performed while on leave from the Research Laboratories, General Motors Corporation, Warren, Michigan 48090.

CONTENTS

INTRODUCTION

The programming language Tower of Babel is well known. Less discussed is the Tower of Metababel, symbolic of the many ways that programming languages are described and defined. The methods used range all the way from natural language to the ultramathematical. The former are subject to all the vagaries and inconsistencies that result from the use of normal prose; the latter frequently have their meaning hidden under abstruse notation.

Often a mixture of methods is used. The formal part is generally limited to the use of Backus Naur Form (BNF), or some equivalent, to define the context-free aspects of the language. The context-sensitive restrictions and the semantics are then defined by some other method, usually prose. In this paper, we confine ourselves to completely formal techniques.

Computer science has already made considerable progress without having a generally accepted formal technique for defining programming languages, just as the English language was well developed before the advent of Johnson's *Dictionary of the English Language* in 1755. However, the lack of general use of formal definitions has not been without severe consequences. For example:

- Language designers do not have good tools for careful analysis of their decisions.
- Standardization efforts have been impeded by a lack of an adequate formal notation.
- Despite the fact that standards exist for programming languages, it is still risky to move a program from one implementation to another, even on the same hardware.
- It is impossible to make a contract with a vendor for a compiler and be assured that the product will be an exact implementation of the language.
- It is difficult to write reference manuals and tutorial texts without glossing over critical details that may change from implementation to implementation.
- The answers to detailed questions about a programming language frequently have to be obtained by trying an implementation or hoping for a consensus from several implementations.

Most of these problems would be avoided if there were good formal definitions for the languages. There would then be a single place for the precise details of each language, and no question would be left unanswered, and importantly, there would be a tendency to improve the design of languages by bringing their complexities out into the open. It is easy to say, "Language X is block structured and jumps out of blocks are permitted," but without a formal description of language X, the consequences are not obvious.

All methods of definition treat the following general problem. Given an alphabet of symbols S, the set S^* is the set of all possible symbol strings that can be constructed from S. A definition both provides rules for selecting the set $P \subseteq S^*$ of *legal* programs of the language being defined, *and* specifies the *meaning* of each legal program $p \in P$.

There is considerable difference in the way the various definition methods select and specify the set of legal programs and their meanings. These differences give rise to the following questions:

1) What precisely constitutes a valid program: one whose context-free syntax is correct, one whose context-sensitive syntax is correct, or one that does not infringe any of the semantic rules of the language during execution?
2) Should the definition model be based on the concept of an underlying machine?
3) How should a formal definition show errors: explicitly in the definition, or implicitly by rules that only generate valid programs?
4) Should a definition attempt to indicate the places that an implementation may introduce restrictions, and is it possible to foresee all such restrictions?
5) Should a definition also be suitable for automatic (machine) implementation?

Indeed we, the authors, have differing answers to these questions.

In this paper, we make the assumption that the raison d'être of a language definition is to provide information, and in particular, to answer questions about a language. The questions may vary from the very general, "What data types are supported in the language?" to the more detailed, "Are both parts of a disjunction always evaluated?" The usefulness of a definition can, therefore, be judged by the quality of the answers it provides.

Among the characteristics that are important to the successful use of any method are:

- *Completeness.* There must be no gaps in the definition. In particular, there must be no questions about the syntax or semantics of the language that cannot be answered by means of the definition.
- *Clarity.* The user of the definition must be able to understand the definition and to find answers to his questions easily. While it is obvious that some facility with the notation of the language is essential before being able to understand the definition fully, the amount of effort required should be small.
- *Naturalness.* The naturalness of a notation has a very large effect on the ability of a user to understand a definition. The naturalness of a notation is more important than its conciseness, although there is a relation between the two. We have, therefore, used notational abbreviations only where there is a real gain in clarity, and we have chosen mnemonic names wherever possible.
- *Realism.* Although the designer of a language may wish his universe of discourse to be free from such mundane restrictions as finite numeric ranges and bounded storage, these restrictions are the realities of the implementor's world. The definition provided by the designer, which is the implementor's manufacturing specifications, must specify exactly where restrictions or choices can be made, and where the designer's unobstructed landscape must be modeled exactly.

We present here a prose description and four very different formal definitions of the same language. After giving these definitions, we pose several questions about the language being defined and examine the ease with which one of them can be answered by means of the definition. This leads to a critical review and evaluation of the techniques discussed. The language used in the analysis is ASPLE, taken from Cleaveland and Uzgalis [C1] where it is defined by a W-grammer, an extension of the method developed by van Wijngaarden [W2] and used to define ALGOL 68. Our first formal definition of ASPLE is derived from the definition presented in [C1]. During the development of the other formal definitions, this W-grammar definition was taken as the final arbiter on the syntax and semantics of ASPLE.

A W-grammar consists of two sets of rules, the metaproductions and the hyperrules. These combine to permit the formation of a potentially infinite set of productions, which are used to define the syntax and the context-sensitive requirements. The semantics are specified by using these productions to generate all possible execution sequences for a valid program.

The second formal definition is a development of the Production Systems approach proposed by Ledgard [L2, L3]. Production Systems are used to construct a generative grammar that directly specifies both the context-free and the context-sensitive requirements of the language syntax. The semantics are specified by a second set of productions that map legal programs into another target language. In this paper, the axiomatic approach of Hoare [H1] is used as the basis for such a target language.

The next formal definition uses the Vienna Definition Language [L4, L6, L7, W1]. With this method, a procedure is defined that transforms a program string into a tree representation according to the context-free syntax of the language. This tree is then converted into an abstracted form that retains only those parts of the program that are required to express its meaning. During this conversion, the context-sensitive requirements of the language are checked. Finally, the meaning of the abstracted program is defined by its execution on an abstract machine.

The final formal definition technique is that of Attribute Grammars [K1, L5, B1] which augments a context-free grammar with "attributes" attached to the syntactic categories. These attributes are given values computed from the productions of the parent or descendant nodes in the derivation tree for a program. This technique allows the designer to specify the context-sensitive requirements of a language directly and to define the meaning of a program by translating it into a separately defined sequence of actions.

One other major definition approach, developed by Scott and Strachey [S2], is not considered in this paper. For a more detailed discussion and bibliography of this method, see the recent works by Donahue [D1] and Tennent [T1].

We make no attempt to provide a formal proof of the equivalence of our four definitions of ASPLE. Such a proof is beyond the scope of this paper. It is a reflection of the current state of formal definitions that an attempt at such a proof, even for a toy language like ASPLE, is excessively difficult. For a real programming language, the quantity of detail involved is beyond the control of unaided human effort. So far, little has been done to provide mechanical aids for checking formal definitions.

There are three important applications of formal definitions that we do not consider in this paper:

1) theoretical study of the foundations of programming languages;
2) automatic implementation of compilers; and
3) automatic validation of programs.

To assist the reader, we have included comments in the bodies of the actual definitions. These are separated from the formal part by the use of square brackets.

1. INFORMAL DESCRIPTION OF ASPLE

ASPLE is a very small language derived from ALGOL 68. Its context-free syntax is defined in Table 1.1 using BNF.

An ASPLE program consists of a sequence of declarations followed by a sequence of executable statements. Each identifier used in an executable statement must appear once and only once in the declarations. A declaration associates a "mode" with one or more identifiers. The mode of an identifier specifies: 1) the type of the value (integer or Boolean) to which it may refer, and 2) whether the reference is made directly or through a declared number of pointers. The executable statements of ASPLE are assignments, *if-then-else* conditionals, *while-do* loops, input and output statements, all of which are of familiar syntax.

```
[B01] <program>         ::= begin <dcl train> ,
                                  <stm train> end

[Declarations]

[B02] <dcl train>       ::= <declaration>
                          | <declaration> ; <dcl train>
[B03] <stm train>       ::= <statement>
                          | <statement> , <stm train>
[B04] <declaration>     ::= <mode> <idlist>
[B05] <mode>            ::= bool
                          | int
                          | ref <mode>
[B06] <idlist>          ::= <id>
                          | <id> , <idlist>

[Statements]

[B07] <statement>       ::= <asgt stm>
                          | <cond stm>
                          | <loop stm>
                          | <transput stm>
[B08] <asgt stm>        ::= <id  =  exp >
[B09] <cond stm>        ::= if <exp>
                               then <stm train> fi
                          | if <exp>
                               then <stm train>
                               else <stm train> fi
[B10] <loop stm>        ::= while <exp> do
                                  <stm train> end
[B11] <transput stm>    ::= input <id>
                          | output <exp>

[Expressions]                ::= <factor>
                               | <exp> + <factor>
[B12] <exp>                  ::= <primary>
                               | <factor> * <primary>
[B13] <factor>               ::= <id>
                               | <constant>
[B14] <primary>              | (<exp>)
                             | (<compare>)
[B15] <compare>              ::= <exp> = <exp>
                               | <exp> ≠ <exp>

[Constants and Identifiers]

[B16] <constant>        ::= <bool constant>
                          | <int constant>
[B17] <bool constant>   ::= true
                          | false
[B18] <int constant>    ::= <number>
[B19] <number>          ::= <digit>
                          | <number> <digit>
[B20] <digit>           ::= 0 | 1 | ... | 9
[B21] <id>              ::= <letter>
                          | <id> <letter>
[B22] <letter>          ::= A | B | ... | Z
```

TABLE 1.1. BNF DESCRIPTION OF ASPLE

An an example of an ASPLE program, consider the following:

```
begin
   int  X, Y, Z;
   input  X;
   Y := 1;
   Z := 1;
   if (X ≠ 0) then
      while (Z ≠ X) do
         Z := Z + 1;
         Y := Y * Z
      end
   fi;
   output Y
end
```

This program reads in a positive integer value, then computes and prints its factorial. The program declares three integer variables X, Y, and Z. It starts by reading the value of X from the input file and setting the values of both Y and Z equal to 1. If the value of X is not zero, the factorial is computed by successively multiplying Y by increasing values of Z until X equals Z. The final value of Y, the factorial of X, is then printed on the output file.

This sample ASPLE program uses only identifiers that refer directly to integral values. These are similar, for example, to the variable A in the declaration:

int A

This variable, like all variables in ASPLE, must be given a value, either by assignment or input, before it can be used in an expression. Since A refers to integral values, its mode is reference-to-integral. This declaration of A may be contrasted with a variable B declared as:

ref int B

Here B is a variable that refers to an integral value through a single level of indirection. Thus the mode of B is a reference-to-reference-to-integral. In this case, we say that the "primitive mode" of B is integral. Executing the assignment:

$$B := A$$

sets the value of B to be a reference to A, which in turn refers directly to an integral value. Executing the assignment:

$$A := 7$$

does not change the value of B, still a reference to A, but it does change the integral value to which A refers, the same value that B refers to indirectly. To obtain the integral value to which B refers, the value of B must be "dereferenced" twice. This is the "primitive value" of B. This mechanism is extended for variables declared with multiple levels of indirection and applies to Boolean values as well.

To evaluate an expression consisting of two identifiers separated by a $+$ or $*$, the value of each of the identifiers must be dereferenced as many times as needed to obtain a primitive value of the same mode, integral or Boolean. The operators $+$ and $*$ placed between integral values represent addition and multiplication, respectively. Between Boolean values, they represent the logical "*or*" and "*and*" operations, respectively. The operators $=$ and $\neq$ apply only to integral values and yield a Boolean value as a result. An expression in parentheses always yields a primitive value.

In an assignment statement, the mode of the identifiers on the left side must be compatible with the mode of the value on the right side. To be compatible, two conditions must be satisfied:

1) both sides must have the same primitive mode;
2) if the mode of the identifier on the left side contains n_l occurrences of "reference to" and the mode of the value of the right side contains n_r such occurrences, then the relation $n_l - 1 \leq n_r$ must hold.

For example, given the declarations:

int A;
bool B;
ref int C;
ref ref int D;

both the assignments:

$$A := 16 \qquad n_l = 1, \quad n_r = 0$$
$$C := D \qquad n_l = 2, \quad n_r = 3$$

satisfy the two compatibility requirements. On the other hand, the assignment

$$A := B$$

violates the first condition, and the assignments

$$C := 20 \qquad n_l = 2, \quad n_r = 0$$
$$D := A \qquad n_l = 3, \quad n_r = 1$$

both violate the second condition and are thus illegal.

The process of assignment takes place as follows:

1) The right side is evaluated to obtain a value v.
2) The value v is dereferenced sufficiently so that the mode of the value obtained contains one fewer occurrence of "reference to" than does the mode of the identifier on the left side.
3) The value referred to by the identifier on the left side is replaced by the value obtained in step 2).

To illustrate the mechanism of the assignment statment, consider the following program[1]

```
begin                                          [01]
  int INTA,  INTB;                             [02]
  ref int REFINTA,  REFINTB;                   [03]
  ref ref int REFREFINTA,  REFREFINTB;         [04]
    INTA := 100;                               [05]
    INTB := 200;                               [06]
    REFINTA := INTA;                           [07]
    REFINTB := INTB;                           [08]
    REFREFINTA := REFINTA;                     [09]
    REFINTA := INTB;                           [10]
    INTB := REFREFINTA;                        [11]
    input  REFREFINTA;                         [12]
    output  REFINTB                            [13]
end                                            [14]
```

After line [09] has been executed, two chains of references will have been set up. The state is shown schematically in Figure 1. Note that *REFREFINTB* has not been assigned a value. The assignment of line [10] causes *REFINTA* to refer to *INTB*, no other value being changed. The situation after executing this statement is as shown in Figure 2.

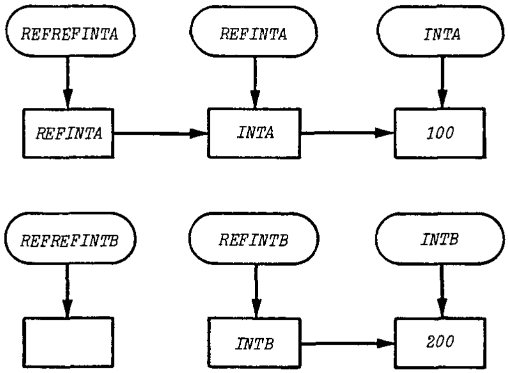

Figure 1.

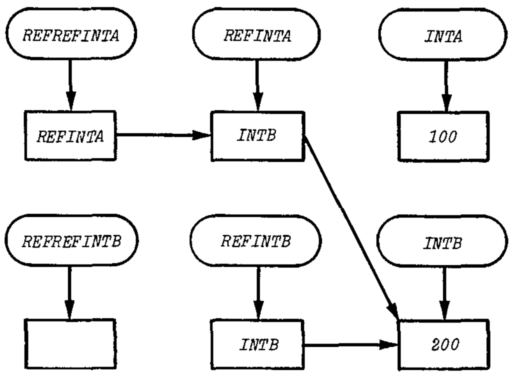

Figure 2.

[1] In this program and throughout this paper, line numbers are included for reference purposes.

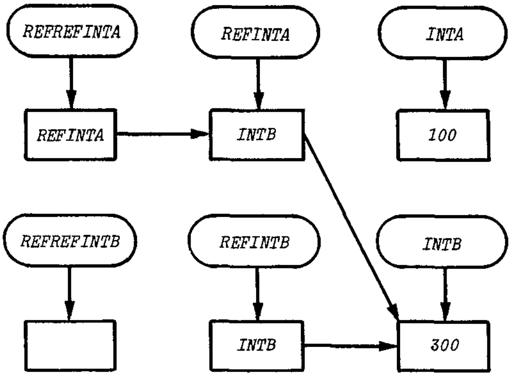

Figure 3.

The assignment of line [11] makes no change in the value of *INTB* because of the effect of the statement of line [10]. The *input* statement of line [12] causes a value, say 300, to be read from the input file and assigned to the variable found by following the chain starting at *REFREFINTA*. The semantics of ASPLE require that this chain be set up by a sequence of assignment statements before an *input* statement is executed. The result is depicted in Figure 3. The final statement thus prints the value 300. An attempt to execute

output REFREFINTB;

in place of line [13] is illegal, since the value of *REFREFINTB* is undefined and cannot be dereferenced to produce a primitive value.

There are a number of details of ASPLE that are left for the implementor to define. For example, the context-free syntax makes no limit on the number of variables that can be declared or on the length of the program. Any actual implementation will be bounded by machine constraints in these areas. Table 1.2 lists the features which the implementor must supply to complete the definition of the language. These values have a bearing on both the syntax and the semantics of ASPLE.

As a final note, this informal introduction makes no pretense of being a complete definition of ASPLE. Indeed, it is our contention that a complete definition is almost impossible without the use of a full formal definition method.

1) Maximum length of an ASPLE program, n_1.
2) Maximum number of declared identifiers, n_2.
3) Maximum number of digits in an integer constant, n_3.
4) Maximum number of letters in an identifier, n_4.
5) Maximum value that can be taken by an integer variable, n_5, and the action performed when the addition and multiplication operations of the actual result exceeds n_5.
6) Maximum size of the output file, n_6.

TABLE 1.2. IMPLEMENTATION-DEFINED FEATURES OF ASPLE

2. W-GRAMMARS

The use of two-level grammars known as W-grammars was developed as a definition technique by van Wijngaarden and used for the description of ALGOL 68 [W2]. Cleaveland and Uzgalis, who have given an easy to read exposition [C1] of W-grammars, are the source of the definition of ASPLE from which we have derived the W-grammar presented in this section. To maintain a consistent notation throughout this paper, we have departed slightly from the usage of [C1,W2].

A finite set of BNF productions is often used to define the context-free parts of a programming language. A W-grammar consists of two finite sets of rules, the *metaproductions* and the *hyperrules*. The hyperrules are prototypes for context-free productions and, together with the metaproductions, describe how the user can derive a conceptually infinite set of productions. This infinite set of context-free productions is able to specify the context-sensitive restrictions and semantics of a language.

Metaproductions

Metaproductions are context-free productions. The nonterminals of metaproductions, called metanotions, are written in upper case letters, for example, **INTBOOL.**[2] Their terminal strings consist of lower case characters with blanks added to improve readability, for example, **letter** and **ref ref**, the so-called protonotions, to be explained in the following paragraph. In conventional BNF, the nonterminals are distinguished by being enclosed in some form of brackets. In Table 1.1, angle brackets "< and >" are used for this purpose. In W-grammars, no such convention is used. The nonterminals of the productions derived from the hyperrules are words and phrases chosen to give an almost prose-like quality to the grammar.

Consider the following metaproductions taken from the W-grammar definition of ASPLE given in Table 2.1 (p. 200).

[MP01]	**ALPHA**	: :	**a; b; . . . ; z.**
[MP03]	**NOTION**	: :	**ALPHA;** **NOTION ALPHA.**
[MP06]	**INTBOOL**	: :	**int;** **bool.**
[MP07]	**MODE**	: :	**INTBOOL;** **ref MODE.**

Each metaproduction specifies all production alternatives for a given metanotion. For example, the first metaproduction specifies that the metanotion **ALPHA** generates the protonotions **a**, **b**, . . . , or **z**. The symbol "::" is used to separate the left side and the right side of the metaproductions, the symbol ";" is used to separate the alternatives of the right side, and the symbol "." is used to terminate a metaproduction. The metaproduction [MP03] specifies that the metanotion **NOTION** generates either the metanotion **ALPHA**, which in turn generates any lower case character, or the metanotion **NOTION** followed by **ALPHA**. Recursive application of this second alternative allows the generation of any string of lower case characters from the metanotion **NOTION**. Similarly, the metanotion **INTBOOL** generates the protonotions **int** and **bool**, and the metanotion **MODE** generates infinitely many protonotions consisting of a (possibly empty) sequence of **ref**'s followed by **int** or **bool.**

[2] Throughout this paper boldface characters used in the text correspond to the sans serif characters found in the tables. [Editorial note]

[MP01]	ALPHA	a , b , . . . , z .
[MP02]	EMPTY	.
[MP03]	NOTION	ALPHA , NOTION ALPHA
[MP04]	NOTETY	NOTION , EMPTY .
[MP05]	TAG	letter ALPHA , TAG letter ALPHA .
[MP06]	INTBOOL	int , bool .
[MP07]	MODE	INTBOOL , ref MODE .
[MP08]	ONES	one , ONES one .
[MP09]	NUMBER	ONES , EMPTY .
[MP10]	RADIX	one one one one one one one one one one .
[MP11]	BOOL	true , false .
[MP12]	VALUE	NUMBER , BOOL .
[MP13]	BOX	VALUE , undefined , TAG
[MP14]	LOC	loc TAG has MODE refers BOX end .
[MP15]	LOCS	LOC , LOCS LOC
[MP16]	LOCSETY	LOCS , EMPTY .
[MP17]	TABLE	LOCS .
[MP18]	UNIT	loop , assignment , conditional , transput .
[MP19]	SNAP	memory LOCS FILE FILE
[MP20]	SNAPS	SNAP , SNAPS SNAP .
[MP21]	SNAPSETY	SNAPS EMPTY
[MP22]	DATA	EMPTY space VALUE DATA .
[MP23]	FILE	DATA end of file .
[MP24]	RELATE	equals , not equals .
[MP25]	OPER	plus , times , RELATE .
[MP26]	EXP	left EXP OPER EXP right; VALUE, DEREFSETY TAG.
[MP27]	DEREFSETY	EMPTY, deref DEREFSETY
[MP28]	REFS	ref REFS ref.
[MP29]	STMT	EMPTY; if EXP then STMTS else STMTS fi , while EXP do STMTS end , TAG becomes EXP val , DEREFSETY TAG input , EXP output .
[MP30]	STMTS	STMT , STMTS STMT .
[MP31]	STMTSETY	STMTS , EMPTY .
[MP32]	ALPHABET	abcdefghijklmnopqrstuvwxyz .
[MP33]	MAXLEN	. . . [*implementation defined measure of maximum program length* n_1]
[MP34]	MAXTABLE	LOC LOC . LOC . [*the number of occurrences of "LOC" is the implementation defined quantity* n_2]
[MP35]	MAXDIG	ONES token ONES token . . ONES token [*the maximum number of digits is the implementation defined quantity* n_3]
[MP36]	MAXLENGID	letter ALPHA . . letter ALPHA [*the number of occurrences of "letter ALPHA" is the implementation defined quantity* n_4]
[MP37]	MAXINT	one one . . one [*the number of occurrences of "one" is the implementation defined quantity* n_5]
[MP38]	MAXFILELEN	space VALUE space VALUE. [*implementation defined measure of maximum size of an output file* n_6]

TABLE 2.1. METAPRODUCTIONS FOR THE W-GRAMMAR DEFINITION OF ASPLE

Hyperrules

A hyperrule is a blueprint from which context-free productions can be obtained. For example, the hyperrule

[HR94] **NOTION sequence**:
NOTION;
NOTION sequence,
NOTION.

is a prototype for the construction of productions for sequences. It contains both metanotions, in uppercase characters, and protonotions, in lower case characters. The notation is the same as that used for metaproductions, except that the symbol ":" is used instead of "::" to separate the left side from the right side of the rule, and the symbol "," is used to separate different protonotions within the same alternative. A context-free production is obtained from a hyperrule by replacing each metanotion by a protonotion derived from the metaproductions. In this example, the metanotion **NOTION** is to be replaced by a protonotion.

The metaproductions [MP01] and [MP03] allow us to generate an infinite set of protonotions from the metanotion **NOTION**, for example, **value** and **identifier**. Replacing **NOTION** by these protonotions in the preceding hyperrule, we can obtain in turn the productions:

value sequence: **value**;
value sequence,
value.
identifier sequence: **identifier**;
identifier sequence,
identifier.

The nonterminals of these context-free productions are **value sequence**, **value**, **identifier sequence**, and **identifier**. This simple substitution technique is used in W-grammars to generate the infinite set of context-free productions required for the specification of the syntax and the semantics of a language.

In making the substitution of protonotions for metanotions, all occurrences of the same metanotion in the hyperrule must be replaced by the same protonotion. This is the uniform replacement rule. For example, the production:

value sequence: **identifier**;
value sequence,
identifier.

cannot be obtained [HR 94] since the uniform replacement rule would be violated; the metanotion **NOTION** has not been replaced by the same protonotion throughout the hyperrule.

The context-free productions obtained correspond closely to BNF productions. As we have already seen, the nonterminals in the generated productions are separated by commas and may consist of sequences that resemble English phrases when the names of metanotions and protonotions are chosen appropriately. Terminal notions, from which ASPLE programs are constructed, can appear on the right side of hyperrules and thus in the generated productions.

It is customary in W-grammars to write terminal notions as symbols, for example, "comma symbol" for the terminal notion that represents a comma in an ASPLE program. The question of how the symbols are actually represented in terms of character strings is left to the

```
[starting hyper-rule]

[HR01]   program.
            begin,
                 dcl train of TABLE_1,
                 TABLE_1 restrictions,
                 TABLE_1 STMTS stm train,
                 end,
                 where MAXLEN contains begin TABLE_1 STMTS end,
                 FILE_1 stream,
                 FILE_2 stream,
                 execute STMTS with
                    memory TABLE_1 FILE_1 end of file
                    SNAPSETY
                    memory TABLE_2 FILE_3 FILE_2.

[hyper-rules for generating the declaration train of a program]

[HR02]   dcl train of LOCS LOCSETY
            MODE declarer,
                 ref MODE definitions of LOCS,
                 ;
                 dcl train of LOCSETY,
            where LOCSETY is EMPTY,
                 MODE declarer,
                 ref MODE definitions of LOCS,
                 ;

[HR03]   ref MODE declarer
            ref,
                 MODE declarer.

[HR04]   int declarer   int.

[HR05]   bool declarer  bool.

[HR06]   MODE definitions of loc TAG has MODE refers undefined end LOCSETY
            TAG identifier,
                 ;
                 MODE definitions of LOCSETY,
            where LOCSETY is EMPTY,
                 TAG identifier.

[hyper-rules for checking context-sensitive requirements on the symbol table]

[HR07]   LOCSETY loc TAG has MODE refers undefined end restrictions
            where TAG is not in LOCSETY,
                 where MAXTABLE contains LOC LOCSETY,
                 LOCSETY restrictions,
            where LOCSETY is EMPTY.

[HR08]   where TAG_1 is not in loc TAG_2 has MODE refers undefined end LOCSETY
            where TAG_1 differs from TAG_2,
                 where TAG_1 is not in LOCSETY,
            where LOCSETY is EMPTY,
                 where TAG_1 differs from TAG_2.
```

TABLE 2.2. HYPERRULES FOR THE W-GRAMMAR DEFINITION OF ASPLE

```
[hyper-rules for generating the statement train of a program]

[HR09]    TABLE STMT STMTSETY stm train:
             TABLE STMT UNIT,
                  ;,
                  TABLE STMTSETY stm train,
             where STMTSETY is EMPTY,
                  TABLE STMT UNIT

[HR10]    TABLE TAG becomes EXP val assignment
             TABLE ref MODE TAG identifier,
                  =,
                  TABLE EXP MODE value.

[HR11]    TABLE if EXP then STMTS1 else STMTS2 fi conditional:
             if,
                  TABLE EXP bool value,
                  then,
                  TABLE STMTS1 stm train,
                  TABLE STMTS2 elsend.

[HR12]    TABLE STMTS elsend
             fi,
                  where STMTS is EMPTY,
             else,
                  TABLE STMTS stm train,
                  fi.

[HR13]    TABLE while EXP do STMTS end loop
             while,
                  TABLE EXP bool value,
                  do,
                  TABLE STMTS stm train,
                  end.

[HR14]    TABLE EXP input transput.
             input,
                  strong TABLE EXP ref INTBOOL identifier.

[HR15]    TABLE EXP output transput
             output,
                  TABLE EXP INTBOOL value.

[hyper-rules for generating an expression]

[HR16]    TABLE left EXP1 plus EXP2 right INTBOOL value
             TABLE EXP1 INTBOOL value,
                  +,
                  TABLE EXP2 INTBOOL factor.

[HR17]    TABLE EXP MODE value
             TABLE EXP MODE factor

[HR18]    TABLE left EXP1 times EXP2 right INTBOOL factor:
             TABLE EXP1 INTBOOL factor,
                  *,
                  TABLE EXP2 INTBOOL primary.

[HR19]    TABLE EXP MODE factor:
             TABLE EXP MODE primary.
```

TABLE 2.2. —*Continued*

```
[HR20]    TABLE EXP MODE primary
             strong TABLE EXP MODE identifier;
             TABLE EXP MODE value pack,
             MODE EXP denotation,
                  where MODE is INTBOOL,
             TABLE EXP compare pack,
             where MODE is bool.

[HR21]    TABLE left EXP1 RELATE EXP2 right compare
             TABLE EXP1 int value,
                  relate symbol,
                  TABLE EXP2 int value.

[HR22]    equals symbol   =.

[HR23]    not equals symbol   ≠

[HR24]    strong TABLE deref EXP MODE identifier
             strong TABLE EXP ref MODE identifier.

[HR25]    strong TABLE TAG MODE identifier
             TABLE MODE TAG identifier.

[HR26]    TABLE MODE TAG identifier
             TAG identifier,
                  where TABLE contains loc TAG has MODE,
                  where MAXLENGID contains TAG.

[HR27]    letter ALPHA identifier·
             letter ALPHA symbol,
                  TAG identifier

[HR28]    letter ALPHA identifier.
             letter ALPHA symbol.

[HR29]    letter  a  symbol   A.

[HR39]    letter  b  symbol   B.

etc.      :

[HR54]    letter  z  symbol   Z.

[HR55]    bool true denotation   true.

[HR56]    bool false denotation  false.

[HR57]    int NUMBER1 denotation
             NUMBER1 token,
             int NUMBER2 denotation,
                  NUMBER3 token,
                  where NUMBER4 equals NUMBER2 times RADIX,
                  where NUMBER1 equals NUMBER4 plus NUMBER3,
                  where MAXDIG contains NUMBER1 denotation.
```

TABLE 2.2. —*Continued*

[HR58] token *0*

[HR59] one token *1.*

[HR60] one one token *2.*

[HR61] one one one token *3.*

[HR62] one one one one token *4.*

[HR63] one one one one one token *5.*

[HR64] one one one one one one token *6.*

[HR65] one one one one one one one token *7.*

[HR66] one one one one one one one one token *8.*

[HR67] one one one one one one one one one token *9.*

[*hyper-rules for generating the representation of a file*]

[HR68] space VALUE FILE stream
VALUE denotation,
,
FILE stream.

[HR69] end of file stream *eof.*

[*hyper-rules for checking the execution semantics of statements*]

[HR70] execute STMT STMTS with $SNAPS_1$ SNAP $SNAPS_2$
execute STMT with $SNAPS_1$ SNAP,
execute STMTS with SNAP $SNAPS_2$.

[HR71] execute if EXP then $STMTS_1$ else $STMTS_2$ fi with SNAP SNAPS
evaluate EXP from SNAP giving true,
execute $STMTS_1$ with SNAP SNAPS,
evaluate EXP from SNAP giving false,
execute $STMTS_2$ with SNAP SNAPS.

[HR72] execute while EXP do STMTS end with $SNAP_1$ $SNAPSETY_1$ $SNAP_2$ $SNAPSETY_2$
evaluate EXP from $SNAP_1$ giving false,
where $SNAP_1$ is $SNAP_2$,
where $SNAPSETY_1$ $SNAPSETY_2$ is EMPTY,
evaluate EXP from $SNAP_1$ giving true,
execute STMTS with $SNAP_1$ $SNAPSETY_1$ $SNAP_2$,
execute while EXP do STMTS end with $SNAP_2$ $SNAPSETY_2$

[HR73] execute TAG becomes EXP val with $SNAP_1$ $SNAP_2$
evaluate EXP from SNAP giving BOX_2,
where $SNAP_1$ is
memory $LOCSETY_1$
loc TAG has MODE refers BOX_1 end
$LOCSETY_2$ $FILE_1$ $FILE_2$,
where $SNAP_2$ is
memory $LOCSETY_1$
loc TAG has MODE refers BOX_2 end
$LOCSETY_2$ $FILE_1$ $FILE_2$

TABLE 2.2. —*Continued*

```
[HR74]  execute DEREFSETY TAG1 input with SNAP1 SNAP2.
            evaluate DEREFSETY TAG1 from SNAP1 giving TAG2,
                where SNAP1 is
                  memory LOCSETY1
                  loc TAG2 has ref INTBOOL refers BOX1 end
                  LOCSETY2 space VALUE FILE1 FILE2,
                where SNAP2 is
                  memory LOCSETY1
                  loc TAG2 has ref INTBOOL refers VALUE end
                  LOCSETY2 FILE1 FILE2,
                where VALUE matches INTBOOL,
             where SNAP1 is memory LOCS end of file FILE.
                [end of file error] abnormal termination.

[HR75]     where NUMBER matches INTBOOL
              where INTBOOL is int,
              where INTBOOL is bool,
                  [input error] abnormal termination

[HR76]     where BOOL matches INTBOOL
              where INTBOOL is bool,
              where INTBOOL is int,
                  [input error] abnormal termination.

[HR77]     execute EXP output with SNAP1 SNAP2
              evaluate EXP from SNAP1 giving VALUE,
                  where SNAP1 is memory LOCS FILE1 DATA end of file,
                  where SNAP2 is memory LOCS FILE1 DATA space VALUE end of file,
                  where MAXFILELEN contains DATA space VALUE,
              evaluate EXP from SNAP1 giving VALUE,
                  where SNAP1 is memory LOCS FILE1 DATA end of file,
                  where SNAP2 is memory LOCS FILE1 DATA space VALUE end of file,
                  where DATA space VALUE contains MAXFILELEN,
                  [output file overflow] abnormal termination.

[HR78]     execute EMPTY with SNAP SNAP.  true

[hyper-rules for evaluating expressions]

[HR79]     evaluate left EXP1 OPER EXP2 right from SNAP giving VALUE1
              evaluate EXP1 from SNAP giving VALUE2,
                  evaluate EXP2 from SNAP giving VALUE3,
                  where VALUE1 equals VALUE2 OPER VALUE3

[HR80]     evaluate deref DEREFSETY TAG from SNAP giving BOX1
              evaluate DEREFSETY BOX2 from SNAP giving BOX1,
                  where SNAP contains loc TAG has MODE refers BOX2 end.

[HR81]     evaluate BOX from SNAP giving BOX
              where BOX differs from undefined,
              where BOX is undefined,
                  [uninitialized variable reference error] abnormal termination
```

TABLE 2.2. *—Continued*

[HR82] where NUMBER$_1$ equals NUMBER$_2$ plus NUMBER$_3$
where MAXINT contains NUMBER$_2$ NUMBER$_3$,
where NUMBER$_1$ is NUMBER$_2$ NUMBER$_3$,
where NUMBER$_2$ NUMBER$_3$ one contains MAXINT,
[*arithmetic overflow*] abnormal termination.

[HR83] where NUMBER$_1$ equals NUMBER$_2$ times NUMBER$_3$ one.
where MAXINT contains NUMBER$_1$,
where NUMBER$_1$ is NUMBER$_4$ NUMBER$_2$,
where NUMBER$_4$ equals NUMBER$_2$ times NUMBER$_3$,
where NUMBER$_4$ NUMBER$_2$ one contains MAXINT,
where NUMBER$_4$ equals NUMBER$_2$ times NUMBER$_3$,
[*arithmetic overflow*] abnormal termination.

[HR84] where NUMBER equals NUMBER times one true.

[HR85] where EMPTY equals NUMBER times EMPTY true

[HR86] where true equals BOOL$_1$ plus BOOL$_2$ where BOOL$_1$ is true,
where BOOL$_2$ is true.

[HR87] where false equals false plus false true

[HR88] where false equals BOOL$_1$ times BOOL$_2$ where BOOL$_1$ is false,
where BOOL$_2$ is false.

[HR89] where true equals true times true true.

[HR90] where true euqals NUMBER equals NUMBER true.

[HR91] where false equals NUMBER$_1$ equals NUMBER$_2$ where NUMBER$_1$ differs from NUMBER$_2$.

[HR92] where false equals NUMBER not equals NUMBER true.

[HR93] where true equals NUMBER$_1$ not equals NUMBER$_2$. where NUMBER$_1$ differs from NUMBER$_2$.

[*hyper-rules for defining sequences and packs, and for checking various conditions*]

[HR94] NOTION sequence
NOTION,
NOTION sequence,
NOTION.

[HR95] NOTION pack
(,
NOTION,
).

[HR96] true EMPTY.

[HR97] where NOTETY is NOTETY true.

[HR98] where NOTETY$_1$ NOTION NOTETY$_2$ contains NOTION true

[HR99] where NOTETY$_1$ ALPHA$_1$ differs from NOTETY$_2$ ALPHA$_2$
where NOTETY$_1$ differs from NOTETY$_2$,
where ALPHA$_1$ precedes ALPHA$_2$ in ALPHABET,
where ALPHA$_2$ precedes ALPHA$_1$ in ALPHABET

[HR100] where ALPHA$_1$ precedes ALPHA$_2$ in NOTETY$_1$ ALPHA$_1$ NOTETY$_2$ ALPHA$_2$ NOTETY$_3$ true.

TABLE 2.2. —*Continued*

implementation. For example, in [C1] hyperrule [HR03] is given as

ref MODE declarer:
  ref symbol,
  MODE declarer.

The string **ref symbol** is a terminal notion, which in an ASPLE program would have the character string representation *ref*.

For continuity between the four definition techniques, we use a different notation for terminal notions in this paper. These are written in italic or underlined characters, thus suggesting directly their character string representation. For example, the preceding hyperrule is written as

[HR03] **ref MODE declarer:**
  ref,
  MODE declarer.

and the **not equals symbol** is defined by the hyperrule:

[HR23] **not equals symbol:** $\underline{\neq}$.

There is a close interplay between the metaproductions and the hyperrules. The hyperrules are essentially parameterized macrostatements for context-free productions with the metanotions used as formal parameters. These metanotions are abstractions for constructs that are very much program dependent, for example, the symbol table and the abstracted statement train.

Overview of the W-grammar Definition of ASPLE

The metaproductions given in Table 2.1 and the hyperrules given in Table 2.2 form a W-grammar that defines all aspects of the context-free and context-sensitive syntax and semantics of ASPLE. The starting hyperrule [HR01] affords an overview of these three segments:

[HR01] **program:**
begin, [01]
dcl train of TABLE$_1$, [02]
TABLE$_1$ **restrictions,** [03]
TABLE$_1$ **STMTS stm train,** [04]
end, [05]
where MAXLEN contains begin TABLE$_1$ **STMTS end,** [06]
FILE$_1$ **stream,** [07]
FILE$_2$ **stream,** [08]
execute STMTS with [09]
memory TABLE$_1$ **FILE**$_1$ **end of file** [10]
SNAPSETY [11]
memory TABLE$_2$ **FILE**$_3$ **FILE**$_2$. [12]

Lines [01] and [05] give the terminals that mark the start and finish of an ASPLE program. Lines [02] through [04] are the prototypes for the nonterminals from which the declare train and the statement train of an ASPLE program can be derived. Lines [06] through [12] define the semantics of this program. An input file can be derived from line [07], and an output file can be derived from line [08]. Lines [09] through [12] ensure that the output file derived from line [08] is the one that would be obtained by executing the program with the input file derived from line [07].

Line [02] gives the prototype for the nonterminal from which the declare train of the program can be derived. This line contains the metanotion $\mathbf{TABLE}_1$, which is an abstraction of the "symbol table" of the program being defined. The actual symbol table is a protonotion that can be derived from **TABLE** using the metaproductions. The next section describes the derivation of a symbol table from the declare train of a program. The subscript in $\mathbf{TABLE}_1$ serves to distinguish this metanotion from the metanotion $\mathbf{TABLE}_2$ in line [12], since, by convention, the uniform replacement rule applies only to nonterminals with identical subscripts. In addition to serving as a symbol table, $\mathbf{TABLE}_1$ also serves as the initial memory state for the execution of the program, with all variables having the initial value **undefined**.

Line [03] applies the context-sensitive restrictions to the symbol table $\mathbf{TABLE}_1$, which matches the declare train. By applying metaproductions [MP17], [MP15], and [MP14], $\mathbf{TABLE}_1$ in line [03] can be replaced by a protonotion that matches the left side of hyperrule [HR07]:

```
[HR07]      LOCSETY loc TAG has MODE refers undefined end restrictions:
              where TAG is not in LOCSETY,
                where MAXTABLE contains LOC LOCSETY,
                LOCSETY restrictions;
              where LOCSETY is EMPTY.
```

This hyperrule is the only one whose left side contains **restrictions**. Since **restrictions** is a protonotion, it cannot be replaced and will appear on the left side of all productions derived from hyperrule [HR07]. This hyperrule must therefore be used next in the derivation from line [03]. It is used to generate productions that will check that no identifier is declared more than once and that the number of declared identifiers does not exceed the implementation-defined maximum. As we shall see, W-grammars make checks of this kind by using the convention that certain parts of the derivation tree must terminate in an "empty sequence." The restrictions are enforced by ensuring that only for legal programs can every protonotion in the derivation tree be reduced to either an empty sequence or to a sequence of terminals forming the program.

Line [04] specifies a statement train and uses the symbol table $\mathbf{TABLE}_1$ to check the context-sensitive requirements on statements. Line [04] also contains a metanotion **STMTS** which is replaced by protonotions derived from the metaproductions. These protonotions form an abstraction of the statement train described in the Subsection, Internal Representation of the Statement Train [see page 213]. It is this abstracted form of the program that is used to specify the semantics of the program, as is described in the Subsection, Semantic Definition [see page 215]. Line [06] is used to check that the program is not too long, as specified by the implementation-defined metanotion **MAXLEN**.

Lines [07] and [08] generate the input and output files. $\mathbf{FILE}_1$ denotes the input file, and $\mathbf{FILE}_2$ denotes the output file obtained after execution of the program. The terminal string generated by the W-grammar consists of a program text followed by a representation of the initial input file and the final output file.

Lines [09] through [12] specify the semantics of executing **STMTS**, starting with the initial memory state in $\mathbf{TABLE}_1$ and the input file $\mathbf{FILE}_1$. Initially the output file is empty and this is represented by **end of file**. The metanotion **SNAPSETY** is used to derive a series of "snapshots" that record the sequence of memory states caused by the execution of **STMTS**. Each snapshot contains the current memory state and the state of the input and output files. The final snapshot is line [12]. By the uniform replacement rule, the protonotion replacing $\mathbf{FILE}_2$ must be the same as the one in line [08] which generates the final output file. The metanotion $\mathbf{FILE}_3$ denotes the input file at the end of execution and contains the values of the input file that were not used as input to the program.

As already mentioned, there are checks that certain protonotions correspond according to the rules of ASPLE. For example, in line [02], **TABLE**$_1$ must be consistent with the declare train of the program; and in line [09], the sequence of memory snapshots must follow from the abstracted program **STMTS**. These checks are accomplished by rules in the grammar that reduce to **EMPTY**, that is, empty sequence, if and only if certain conditions are satisfied. For example, suppose we had a derivation that terminates in the nonterminal protonotion

where one token equals one token times one

From the hyperrule [HR84]

[HR84] **where NUMBER equals NUMBER times one: true.**

we can derive the production

where one token equals one token times one: true.

The hyperrule [HR96]

[HR96] **true: EMPTY.**

and the metaproduction [MP02]

[MP02] **EMPTY :: .**

show that we can derive the empty sequence from **true**. Thus the empty sequence can be derived from the protonotion

where one token equals one token times one

However, had the nonterminal in the derivation tree of a program been

where one token equals token times one

we would not have been able to generate a production that would lead to an empty sequence. It is in this way that the W-grammar shows that a program is illegal.

Similarly, line [03] generates the empty (terminal) string if and only if the context-sensitive restrictions of the symbol table are satisfied. Lines [09] through [12] will generate an empty sequence only if the input and output files correspond to the semantics of the program. If the conditions are not satisfied, there are no production rules that can be generated that will allow an empty terminal string to be derived from these lines. One of the difficulties with this technique is that, in general, there is no method of proving that the required production rules cannot be generated. The user must be convinced of this fact informally.

Thus a legal program and its meaning are defined by a W-grammar as a program for which there exists a derivation tree whose terminals, taken in left-to-right order, form:

1) the program;
2) the values of the input file before execution of the program;
3) the values of the output file after execution of the program;

and nothing else.

Symbol Table

The symbol table of the W-grammar is the major vehicle for the specification of the context-sensitive requirements and semantics of ASPLE. A symbol table is a protonotion derived from the metanotion **TABLE**. In this subsection, we will follow in detail the derivation of a valid declare train of a program from line [02] of the hyperrule **[HR01]**. This derivation is typical of the rest of the W-grammar.

The metaproductions:

[MP17]	**TABLE**	: :	**LOCS.**
[MP15]	**LOCS**	: :	**LOC;** **LOCS LOC.**

define a **TABLE** as a nonempty sequence of protonotions derived from **LOC** according to:

[MP14]	**LOC**	: :	**loc TAG has MODE refers** **BOX end.**

Here, the strings **loc, has, refers**, and **end** are included in the protonotion to be derived from **LOC** to help the user with the pattern matching required when searching the table for an applicable hyperrule to use, and to make these protonotions unambiguous. The metanotion **TAG** is defined by:

[MP05]	**TAG**	: :	**letter ALPHA;** **TAG letter ALPHA.**

Thus **TAG** produces a protonotion that represents an identifier. For example, the ASPLE identifier *ABC* is represented by the protonotion **letter a letter b letter c.** As shown earlier, the metanotion **MODE** generates protonotions for the mode of an identifier. The metanotion **BOX**, which holds the value of an identifier, is defined as

[MP13]	**BOX**	: :	**VALUE;** **undefined;** **TAG.**

showing that the value of an identifier is either an integral or a Boolean value, an identifier, or undefined. The fact that the replacement of **TABLE**$_1$ in line [03] of hyperrule **[HR01]**

TABLE$_1$ **restrictions,**

must form a protonotion that matches a left side of hyperrule **[HR07]** requires that **BOX** be replaced in **TABLE**$_1$ by **undefined**. This shows that the initial value of an identifier is undefined in ASPLE.

As an example we consider a program with the declare train:

int A;
bool AB;
ref int C

The protonotion derived from **TABLE** corresponding to this declare train is

loc letter a has ref int refers undefined end
loc letter a letter b has ref bool refers undefined end
loc letter c has ref ref int refers undefined end

Substituting this protonotion in line [02] of hyperrule **[HR01]**,

dcl train of TABLE$_1$ **,**

we obtain the protonotion

dcl train of loc letter a has ref int refers undefined end
loc letter a letter b has ref bool refers undefined end
loc letter c has ref ref int refers undefined end

We next show how the hyperrules of the W-grammar can be used to derive the given declare train from this protonotion. The only hyperrule whose left side starts with **dcl train** is [HR02]:

[HR02] **dcl train of LOCS LOCSETY:**
MODE declarer,
ref MODE definitions of LOCS,
$\underline{;}$,
dcl train of LOCSETY;
where LOCSETY is EMPTY,
MODE declarer,
ref MODE definitions of LOCS,
$\underline{;}$.

If we make the following replacements:

- **loc letter a has ref int refers undefined end**
  for **LOCS**
- **loc letter a letter b has ref bool refers undefined end**
  loc letter c has ref ref int refers undefined end
  for **LOCSETY**
- **int**
  for **MODE**

and, since **LOCSETY** is not **EMPTY**, if we choose the first alternative, we will obtain the following context-free production rule, which we refer to as production [X]:

dcl train of loc letter a has ref int refers undefined end
loc letter a letter b has ref bool refers undefined end
loc letter c has ref ref int refers undefined end:
int declarer,
ref int definitions of loc letter a has ref int refers undefined end,
$\underline{;}$,
dcl train of loc letter a letter b has ref bool refers undefined end
loc letter c has ref ref int refers undefined end.

The right side of production [X] has three nonterminal protonotions and a terminal notion $\underline{;}$. The hyperrule [HR04]

[HR04] **int declarer:** *int*.

allows us to derive the terminal *int* from the first of the three protonotions. The second protonotion contains **definitions of loc** which forces us to choose hyperrule [HR06]:

[HR06] **MODE definitions of loc TAG has MODE refers undefined end LOCSETY:**
TAG identifier,
$\underline{,}$,
MODE definitions of LOCSETY;
where LOCSETY is EMPTY,
TAG identifier,

since this is the only hyperrule whose left side contains this sequence. By making the substitutions:

- the empty string
  for **LOCSETY**
- **ref int**
  for **MODE**
- **letter a**
  for **TAG**

we are able to form a production whose left side matches the second protonotion of production [X]. Since **LOCSETY** is **EMPTY**, we choose the second alternative, and have the production

[X] **ref int definitions of loc letter a has ref int refers undefined end**:
where is,
letter a identifier.

The protonotion **where is** generates the empty string. This can be seen by applying the hyperrules

[HR96] **true**: **EMPTY.**

[HR97] **where NOTETY is NOTETY**: **true.**

with the substitution of the empty string for **NOTETY**, to obtain the production:

where is: true.

As we have already seen, hyperrule [HR96] allows us to derive the empty string from **true**. The protonotion **letter a identifier** generates the terminal symbol *A* by using the hyperrule

[HR28] **letter ALPHA identifier:**
letter ALPHA symbol.

with **ALPHA** replaced by *a*, and

[HR29] **letter a symbol**: *A.*

Applying these production rules to the protonotions we have derived from the production [X]:

int A;
dcl train of loc letter a letter b has ref bool refers undefined end
loc letter c has ref ref int refers undefined end

This is the first part of the declare train of the program followed by a protonotion to which the same technique can be applied to derive the remaining part of the declare train.

Internal Representation of the Statement Train

The symbol table derived from the metanotion **TABLE** serves as an internal representation of the program's declare train. The W-grammar uses the symbol table, together with an internal representation of the program's statement train, to specify the semantics. The internal form of the statement train is a protonotion that can be derived from the metanotion **STMTS** using the metaproductions [MP30] and [MP29]:

```
[MP30]        STMTS  :: STMT;
                        STMTS STMT.
[MP29]        STMT   :: EMPTY;
                        if EXP then STMTS else
                          STMTS fi;
                        while EXP do STMTS end;
                        TAG becomes EXP val;
                        DEREFSETY TAG input;
                        EXP output.
```

For example, the protonotion that corresponds to the statement train of the program

begin
  bool A;
  ref bool C;
  C := *A*;
  input A;
  output C
end

is as follows:

letter c becomes letter a val
letter a input
deref deref letter c output

The correspondence between this protonotion and the written form of the statements is established in the same way as the correspondence between **TABLE** and the written form of the declare train, described in the Subsection, Symbol Table.

The rules that establish this correspondence also specify the context-sensitive requirements of ASPLE. For example, for the assignment statement, the hyperrule:

```
[HR10]        TABLE TAG becomes EXP val assignment:
                TABLE ref MODE TAG identifier,
                  :=,
                  TABLE EXP MODE value.
```

contains in the left-side part the string **TAG becomes EXP val** which is the internal representation of the statement. The right side of the hyperrule reflects the written form:

identifier $\underline{:=}$ **value**

of the assignment statement. The protonotion derived from **TAG** is the representation of the left-side identifier of the assignment statement, and **EXP** is a representation of the right-side expression. The mode of the identifier and the mode of the expression value must be compatible, that is, their primitive modes must be the same. This is ensured by the uniform replacement rule which causes both occurrences of **MODE** in hyperrule [HR10] to be replaced by the same protonotion. In addition, the mode of the value must contain one less **ref** than the declared mode of the identifier in the **TABLE**. This is indicated by the additional **ref** for the mode of the identifier. The fact that **ref MODE** is the declared mode of the identifier in the **TABLE** is enforced by the production rules generated from:

[HR26] **TABLE MODE TAG identifier:**
TAG identifier,
where TABLE contains loc TAG has MODE,
where MAXLENGID contains TAG.

The protonotion substituted for **MODE** in this hyperrule contains the additional **ref** so that the left side of the resulting production matches the protonotion on the right of the production derived from hyperrule [HR10]. The protonotion obtained by substitution in

where TABLE contains loc TAG has MODE

can only be reduced to the empty string if the symbol table contains **TAG** declared with **MODE.**

Semantic Definition

The execution of a program is defined by the sequence of states through which the memory and the input and output files pass. The transition from one state to the next corresponds to the execution of a statement of the program. The sequence of states is represented by the protonotion derived from **SNAPSETY**. This is a sequence of protonotions derived from SNAP (meaning snapshot) which is of the form **memory LOCS FILE FILE** (see metaproductions [MP15] and [MP23]). As we have already seen, **LOCS** generates a protonotion that records the values of the variables and was initially set up as part of **TABLE**$_1$. The two protonotions derived from **FILE** represent the input and output files. Lines [09] through [12] of hyperrule [HR01] provide the root of the derivation tree for the execution

execute STMTS with
memory TABLE$_1$ **FILE**$_1$ **end of file**
SNAPSETY
memory TABLE$_2$ **FILE**$_3$ **FILE**$_2$.

The initial snapshot is **memory TABLE**$_1$ **FILE**$_1$ **end of file**, where **TABLE**$_1$ is the symbol table, in which all the variables have the value **undefined, FILE**$_1$ is the input file, and the output file is empty since it consists only of **end of file**. The final snapshot contains the output file **FILE**$_2$ which, by the uniform replacement rule, will be the same as the protonotion substituted into line [08] of hyperrule [HR01]. Lines [09] through [12] of hyperrule [HR01] will reduce to **EMPTY** only if this sequence of snapshots corresponds exactly to the execution of the protonotion derived from **STMTS.** For each executed statement of the program, a production must be generated that will check that the differences in the states of the memory and files before and after execution of the statement correspond exactly to the semantics of the statement.

The starting and final snapshots corresponding to the execution of the ASPLE program:

begin
bool A;
ref bool C;
input A;
C := *A*;
output C
end

with an initial input file containing the sequence of three values *true*, are:

```
memory loc letter a has ref bool refers undefined end
       loc letter c has ref ref bool refers undefined end
       space true space true space true end of file
       end of file
```

and

```
memory loc letter a has ref bool refers true end
       loc letter c has ref ref bool refers letter a end
       space true space true end of file
       space true end of file
```

respectively.

The execution semantics of the assignment is described by the hyperrule:

```
HR73]   execute TAG becomes EXP val with SNAP1 SNAP2 :
          evaluate EXP from SNAP giving BOX2 ,
            where SNAP1 is
              memory LOCSETY1
              loc TAG has MODE refers BOX1 end
              LOCSETY2 FILE1 FILE2 ,
            where SNAP2 is
              memory LOCSETY1
              loc TAG has MODE refers BOX2 end
              LOCSETY2 FILE1 FILE2 .
```

This hyperrule specifies that the snapshot before execution, **SNAP**$_1$, is identical to the snapshot after execution, **SNAP**$_2$, except that the **BOX**$_1$ to which **TAG** refers in **SNAP**$_1$ has been replaced by **BOX**$_2$, which contains the result of evaluating the expression **EXP** with the variable values of snapshot **SNAP**$_1$.

The arithmetic involved in the evaluation of the expression is performed with numbers expressed in an internal form consisting of strings of the digit **one**. The metanotion **MAXINT** is used to apply the implementation-defined restriction on the maximum value that can be taken by an integer value.

A similar technique is used to define the semantics of all the ASPLE statements. The series of snapshots traces the execution of the program, and the output file shows the result of the computation.

Although the two-level form of W-grammar seems complex, the consistent use of the underlying derivation tree is claimed to give the model an inherent simplicity.

3. PRODUCTION SYSTEMS AND THE AXIOMATIC APPROACH

We now explore the use of Ledgard's Production Systems [L2, L3] and Hoare's axiomatic approach [H1] to define the syntax and the semantics of ASPLE. The Production Systems approach has had a long history, stemming originally from the Production Systems described by Post [P1] and later developed by Smullyan [S3], and by Donovan and Ledgard [D2]; Ledgard continued to develop and describe the approach in writings which, after several iterations, resulted in [L3].

A Production System is a generative grammar somewhat like BNF. Compared with BNF, Production Systems possess an additional power that allows one to define sets of n-tuples and to name specific components of n-tuples. These capabilities are sufficiently powerful to describe any recursively enumerable set, including the set of syntactically legal programs in a language and the translation of those programs into a target language.

In addition to the use of a theoretically complete formal system, the recent development of the Production Systems notation has been mainly guided by principles believed important to a clear and concise notation. These principles include: 1) the strict adherence to a given underlying formal system, allowing only abbreviations that can be mapped directly into the underlying notation; 2) the isolation of the context-free requirements from the context-sensitive requirements on syntax; 3) the belief that many aspects of a definition are better suited to an algorithmic (versus generative) notation. These principles are more fully described in [L3].

Hoare's axiomatic approach is used as a target language to define the semantics of ASPLE and is discussed in the Subsection, Semantics Using the Axiomatic Approach [see page 224].

Syntax Using Production Systems

A definition of the complete ASPLE syntax, including context-sensitive requirements, is given in Table 3.1. To understand this definition, the concept of a syntactic "environment" must first be clarified. An environment is a correspondence between identifiers and modes derived from ASPLE declarations. An environment is computed by applying the function <u>**DERIVED ENV**</u> [PS26]–[PS27] to the declare train of a program. For example, applying this function to the declare train:

int A;
ref int B;
ref ref int C

yields the environment:

$$\begin{aligned}\rho_1 \;\equiv\; \{&A \rightarrow \textbf{REF INTEGER},\\ &B \rightarrow \textbf{REF REF INTEGER},\\ &C \rightarrow \textbf{REF REF REF INTEGER}\}\end{aligned}$$

To specify the context-sensitive requirements of ASPLE, several other functions are defined. The <u>**DOMAIN**</u> [PS48] of an environment ρ is the list of identifiers occurring in ρ. For example, using ρ_1 from the preceding environment:

$$\underline{\textbf{DOMAIN}}(\rho_1) \;\equiv\; A, B, C$$

The function <u>**DERIVED EXP MODE**</u> [PS28]–[PS37] operates over pairs. Given an expression and an environment, this function yields the mode of the expression obtained by using the modes of the identifiers given in ρ. Using ρ_1 above:

$$\begin{aligned}&\underline{\textbf{DERIVED EXP MODE}}(B : \rho_1) &&\equiv \textbf{REF REF INTEGER}\\ &\underline{\textbf{DERIVED EXP MODE}}(A{+}B : \rho_1) &&\equiv \textbf{INTEGER}\end{aligned}$$

The derived mode of $A + X$ in ρ_1 is undefined (in the sense that it is not derivable) since X has not been declared. A function <u>**DERIVED PRIM MODE**</u> [PS38] is also defined, which, given an expression and an environment, yields the primitive mode obtained by dereferencing the derived mode to obtain one of the primitive modes, **INTEGER** or **BOOLEAN**. For example,

$$\begin{aligned}&\underline{\textbf{DERIVED PRIM MODE}}(B : \rho_1) &&\equiv \textbf{INTEGER}\\ &\underline{\textbf{DERIVED PRIM MODE}}(A{+}B : \rho_1) &&\equiv \textbf{INTEGER}\end{aligned}$$

Similarly, the functions <u>**PRIM MODE**</u> [PS39]–[PS41] and <u>**NUM REFS**</u> [PS45]–[PS47], when applied to a mode, yield the corresponding primitive mode and the number of references. For example,

$$\begin{aligned}&\underline{\textbf{PRIM MODE}}(\textbf{REF INTEGER}) &&\equiv \textbf{INTEGER}\\ &\underline{\textbf{NUM REFS}}(\textbf{REF INTEGER}) &&\equiv 1\end{aligned}$$

Next consider the production [PS07] for assignment statements:

[Main Productions]

[PS01] prog PROGRAM <*begin* dt ; st *end*>

← ρ ≡ DERIVED ENV(dt) & DIFF IDLIST<DOMAIN(ρ)> &

[All declared identifiers must be different]

LEGAL<st:ρ> & n_1 ≥ PROGRAM LENGTH(*).

[The statement train must be legal in ρ*;*
n_1 is the maximum program length]

[PS02] dt DECLARE TRAIN <dcl_1 , ..., dcl_n>

← n_2 ≥ NUM DECLARED IDS(*).

[n_2 is the maximum number of declared identifiers]

[PS03] dcl DECLARATION <m id_1, ..., id_n>

← DIFF IDLIST<id_1, ..., id_n>.

[PS04] m MODE <*int* | *bool* | *ref* m>.

[PS05] st STM TRAIN <stm_1, ...; stm_n> & LEGAL <*:ρ>

← LEGAL<stm_1·ρ> & ... & LEGAL <stm_n ρ>.

[A statement train is legal in ρ *only if all contained statements are legal in* ρ*]*

[PS06] stm STATEMENT <stm>

← (ASGT STM<stm> | COND STM<stm> | LOOP STM<stm> | IO STM<stm>).

[PS07] stm ASGT STM <id := exp> & LEGAL <*·ρ>

← LEGAL<id:ρ> & LEGAL<exp:ρ> &

dm_ℓ ≡ DERIVED EXP MODE(id:ρ) & dm_r ≡ DERIVED EXP MODE(exp ρ) &

PRIM MODE(dm_ℓ) = PRIM MODE(dm_r) &

[The primitive modes of id and exp in ρ *must be identical]*

n_ℓ ≡ NUM REFS(dm_ℓ) & n_r ≡ NUM REFS(dm_r) & $n_\ell \leq n_r + 1$.

[The mode of id must be obtainable from the mode of exp by deferencing exp]

[PS08] stm COND STM <*if* exp *then* st *fi*> & LEGAL <*:ρ>

← LEGAL<exp:ρ> & LEGAL<st:ρ> &

DERIVED PRIM MODE(exp.ρ) = BOOLEAN.

[The mode of exp in ρ *must be boolean]*

[PS09] stm COND STM <*if* exp *then* st_1 *else* st_2 *fi*> & LEGAL <*.ρ>

← LEGAL<exp·ρ> & LEGAL<st_1·ρ> & LEGAL<st_2:ρ> &

DERIVED PRIM MODE(exp·ρ) = BOOLEAN.

TABLE 3.1. PRODUCTION SYSTEM SPECIFYING THE COMPLETE SYNTAX OF ASPLE

```
[PS10]  stm   LOOP STM <while exp do st end>  &  LEGAL <*.ρ>
                ← LEGAL<exp·ρ>  &  LEGAL<st:ρ>  &
                  DERIVED PRIM MODE(exp·ρ) = BOOLEAN.

[PS11]  stm   IO STM <input id>  &  LEGAL <* ρ>
                ← LEGAL<id:ρ> .

[PS12]  stm   IO STM <output exp>  &  LEGAL <*:ρ>,
                ← LEGAL<exp.ρ>.

[PS13]  exp   EXPRESSION <fac>  &  LEGAL <*·ρ>
                ← LEGAL<fac·ρ>.

[PS14]  exp   EXPRESSION <fac + exp>  &  LEGAL <* ρ>
                ← LEGAL<fac:ρ>  &  LEGAL<exp·ρ>  &
                  DERIVED PRIM MODE(fac.ρ) = DERIVED PRIM MODE(exp ρ).
                     [The derived primitive modes of fac and exp
                      must be identical]

[PS15]  fac   FACTOR <prim>  &  LEGAL <* ρ>
                ← LEGAL<prim·ρ>.

[PS16]  fac   FACTOR <prim * fac>  &  LEGAL <*·ρ>
                ← LEGAL<prim:ρ>  &  LEGAL<fac·ρ>  &
                  DERIVED PRIM MODE(prim:ρ) = DERIVED PRIM MODE(fac:ρ).

[PS17]  prim  PRIMARY <(exp1 = exp2) | (exp1 ≠ exp2)>  &  LEGAL <*·ρ>
                ← LEGAL<exp1·ρ>  &  LEGAL<exp2:ρ>  &
                  DERIVED PRIM MODE(exp1 ρ) = INTEGER  &
                  DERIVED PRIM MODE(exp2:ρ) = INTEGER.

[PS18]  prim  PRIMARY <(exp)>  &  LEGAL <*·ρ>
                ← LEGAL<exp·ρ>.

[PS19]  prim  PRIMARY <id>  &  LEGAL <*·ρ>
                ← LEGAL<id:ρ>.

[PS20]  prim  PRIMARY <true | false | int>  &  LEGAL <*:ρ>.

[PS21]  int   INTEGER <d1...dn>
                ← n3 ≥ n. [n3 is the maximum length of integers]
```

TABLE 3.1.—*Continued*

[PS22] id IDENTIFIER $<\ell_1 \ldots \ell_n>$ & LEGAL $<* \cdot \rho>$

$\leftarrow \ell_1 \ldots \ell_n \in$ DOMAIN(ρ) & $n_4 > n$.

[*Each identifier must be declared in* ρ; n_4 *is the maximum length of an identifier*]

[PS23] d DIGIT <*0* | *1* | ... | *9*>.

[PS24] ℓ LETTER <*A* | *B* | ... | *Z*>.

[PS25] dm DERIVED ASPLE MODE <INTEGER | BOOLEAN | REF dm>.

[*Auxiliary Functions*]

[PS26] DERIVED ENV(dcl_1; ... ; dcl_n)

≡ { DERIVED ENV(dcl_1), ... ,DERIVED ENV(dcl_n) }.

[PS27] DERIVED ENV(m id_1, ... , id_n) ≡ id_1→dm, ... , id_n→dm

← dm ≡ DERIVED MODE(m).

[PS28] DERIVED EXP MODE(exp + fac :ρ) ≡ INTEGER

← DERIVED PRIM MODE(exp ρ) = INTEGER &

DERIVED PRIM MODE(fac.ρ) = INTEGER.

[PS29] DERIVED EXP MODE(exp + fac ρ) ≡ BOOLEAN

← DERIVED PRIM MODE(exp ρ) = BOOLEAN &

DERIVED PRIM MODE(fac·ρ) = BOOLEAN.

[PS30] DERIVED EXP MODE(fac * prim ρ) ≡ DERIVED EXP MODE(fac + prim .ρ).

[PS31] DERIVED EXP MODE((exp_1 = exp_2) ρ) ≡ BOOLEAN

← DERIVED PRIM MODE(exp_1 .ρ) = INTEGER &

DERIVED PRIM MODE(exp_2 ρ) = INTEGER.

[PS32] DERIVED EXP MODE((exp_1 ≠ exp_2) ρ) ≡ DERIVED EXP MODE((exp_1 = exp_2) ρ).

[PS33] DERIVED EXP MODE((exp) ρ) ≡ DERIVED PRIM MODE(exp·ρ).

[PS34] DERIVED EXP MODE(id:ρ) ≡ dm

← id→dm ∈ ρ. [*id→dm must occur in* ρ]

[PS35] DERIVED EXP MODE(*true*·ρ) ≡ BOOLEAN.

[PS36] DERIVED EXP MODE(*false*:ρ) ≡ BOOLEAN.

[PS37] DERIVED EXP MODE(int ρ) ≡ INTEGER.

[PS38] DERIVED PRIM MODE(exp·ρ) ≡ dm'

← dm ≡ DERIVED EXP MODE(exp·ρ) & dm' ≡ PRIM MODE(dm).

TABLE 3.1.—*Continued*

```
[PS39]   PRIM MODE(INTEGER) ≡ INTEGER.

[PS40]   PRIM MODE(BOOLEAN) ≡ BOOLEAN.

[PS41]   PRIM MODE(REF dm)  ≡ PRIM MODE(dm).

[PS42]   DERIVED MODE(int)   ≡ REF INTEGER.

[PS43]   DERIVED MODE(bool)  ≡ REF BOOLEAN.

[PS44]   DERIVED MODE(ref m) ≡ REF dm
           ← dm ≡ DERIVED MODE(m).

[PS45]   NUM REFS(INTEGER) ≡ 0.

[PS46]   NUM REFS(BOOLEAN) ≡ 0.

[PS47]   NUM REFS(REF dm)  ≡ 1 + NUM REFS(dm).

[PS48]   DOMAIN( {id_1→dm_1, ... ,id_n→dm_n} )
           ≡ id_1, ... ,id_n.

[PS49]   DIFF IDLIST<Λ | id>.
                 [The symbol "Λ" denotes the empty list]

[PS50]   DIFF IDLIST<ℓ,id>
           ← ℓ ≠ Λ   &   id ∉ ℓ.

[Functions for Implementation Dependent Requirements]

[PS51]   NUM DECLARED IDS(dcl_1; ... , dcl_n)
           ≡ NUM DECLARED IDS(dcl_1) +  ...  + NUM DECLARED IDS(dcl_n).

[PS52]   NUM DECLARED IDS(m id_1, ... , id_n)
           ≡ n.

[PS53]   PROGRAM LENGTH(prog) ≡   ..

                 [Implementation defined function to compute the length
                  of a program n_1]
```

TABLE 3.1.—*Continued*

[PS07] **stm** $\textbf{ASGT STM}\ \langle \mathbf{id}\ \underline{:=}\ \mathbf{exp}\rangle\ \ \&\ \ \textbf{LEGAL}\ \langle * : \rho\rangle$
$\leftarrow \textbf{LEGAL}\ \langle \mathbf{id} : \rho\rangle\ \ \&\ \ \textbf{LEGAL}\ \langle \mathbf{exp} : \rho\rangle\ \ \&$
$\mathbf{dm}_\ell \equiv \underline{\textbf{DERIVED EXP MODE}}(\mathbf{id} : \rho)\ \ \&$
$\mathbf{dm}_r \equiv \underline{\textbf{DERIVED EXP MODE}}(\mathbf{exp} : \rho)\ \ \&$
$\underline{\textbf{PRIM MODE}}(\mathbf{dm}_\ell) = \underline{\textbf{PRIM MODE}}(\mathbf{dm}_r)\ \ \&$
[*The primitive modes of id and exp in ρ must be identical*]
$\mathbf{n}_\ell \equiv \underline{\textbf{NUM REFS}}(\mathbf{dm}_\ell)\ \ \&\ \ \mathbf{n}_r \equiv \underline{\textbf{NUM REFS}}(\mathbf{dm}_r)\ \ \&\ \ \mathbf{n}_\ell \leq \mathbf{n}_r + 1.$
[*The mode of id must be obtainable from the mode of exp by deferencing exp*]

In detail, this production may be read: A string of the form

$$\mathbf{id} := \mathbf{exp}$$

is an assignment statement, and the pair

$$\langle \mathbf{id}\ \underline{:=}\ \mathbf{exp} : \rho\rangle$$

is a member of the set **LEGAL**, if
1) **id** is an identifier that is legal in ρ, and
2) **exp** is an expression that is legal in ρ, and
3) $\mathbf{dm}_\ell$ is the derived mode obtained by applying the function **DERIVED EXP MODE** to the **id** on the left side in ρ, and
4) $\mathbf{dm}_r$ is the derived mode obtained by applying the function **DERIVED EXP MODE** to the **exp** on the right side in ρ, and
5) the function **PRIM MODE** maps $\mathbf{dm}_\ell$ and $\mathbf{dm}_r$ into identical primitive modes, and
6) $\mathbf{n}_\ell$ is the integer obtained by applying the function **NUM REFS** to $\mathbf{dm}_\ell$, and
7) $\mathbf{n}_r$ is the integer obtained by applying the function **NUM REFS** to $\mathbf{dm}_r$; and
8) $\mathbf{n}_\ell$ is less than or equal to $\mathbf{n}_r + 1$.

Conditions (3) through (5) indicate that the primitive modes of **id** and **exp** must be identical, and conditions (6) through (8) indicate that the mode of **id** must be obtainable by sufficiently dereferencing **exp.**

In production [PS07], the symbol, "*" in the conclusion for **LEGAL** is used in place of the string:

$$\mathbf{id} := \mathbf{exp}$$

being defined, and the production system variables **id**, **exp**, ρ, **dm**, and **n** (possibly with subscripts) are defined in subsequent productions. The underline on the symbol "$\underline{:=}$" is used to specify that the ":" is an object symbol, and not a Production System punctuation mark separating items in an n-tuple.

More briefly, we shall read several productions from Table 3.1.

[PS01]
prog $\textbf{PROGRAM}\ \langle \textit{begin}\ \mathbf{dt}\ ;\ \mathbf{st}\ \textit{end}\rangle$
$\leftarrow \rho \equiv \underline{\textbf{DERIVED ENV}}(\mathbf{dt})\ \ \&\ \ \textbf{DIFF IDLIST}\ \langle \underline{\textbf{DOMAIN}}(\rho)\rangle\ \ \&$
[*All declared identifiers must be different*]
$\textbf{LEGAL}\langle \mathbf{st}\ \rho\rangle\ \ \&\ \ \eta_1 \geq \underline{\textbf{PROGRAM LENGTH}}(*).$
[*The statement train must be legal in ρ; η₁ is the maximum program length*]

A string of the form

$$\textit{begin}\ \mathbf{dt}\ ;\ \mathbf{st}\ \textit{end}$$

is a valid program if

1) ρ is the environment derived from the declare train **dt**, and
2) the domain of ρ is a list of different identifiers, and
3) **st** is a statement train that is legal in ρ, and
4) η_1 is greater than or equal to the (implementation defined) length of the program.

[PS05]

$$\textbf{st} \quad \textbf{STM TRAIN} \langle \textbf{stm}_1; \ldots ; \textbf{stm}_n \rangle \;\; \& \;\; \textbf{LEGAL} \langle * : \rho \rangle$$
$$\leftarrow \textbf{LEGAL} \langle \textbf{stm}_1 : \rho \rangle \;\; \& \;\; \ldots \;\; \& \;\; \textbf{LEGAL} \langle \textbf{stm}_n : \rho \rangle .$$

[*A statement train is legal in* ρ *only if all contained statements are legal in* ρ]

A sequence of statements of the form

$$\textbf{stm}_1 \; ; \cdots ; \textbf{stm}_n$$

is a statement train, and the statement train is legal in ρ if $\textbf{stm}_1$ through $\textbf{stm}_n$ are statements that are legal in ρ.

[PS14]

$$\textbf{exp} \quad \textbf{EXPRESSION} \langle \textbf{fac} + \textbf{exp} \rangle \;\; \& \;\; \textbf{LEGAL} \langle * : \rho \rangle$$
$$\leftarrow \textbf{LEGAL} \langle \textbf{fac} : \rho \rangle \;\; \& \;\; \textbf{LEGAL} \langle \textbf{exp} : \rho \rangle \;\; \&$$
$$\underline{\textbf{DERIVED PRIM MODE}}(\textbf{fac} : \rho) = \underline{\textbf{DERIVED PRIM MODE}}(\textbf{exp} : \rho).$$

[*The derived primitive modes of fac and exp must be identical*]

A string of the form

$$\textbf{fac} + \textbf{exp}$$

is an expression, and the expression is legal in ρ, if
1) **fac** is a factor that is legal in ρ and
2) **exp** is an expression that is legal in ρ and
3) the derived primitive mode of **fac** in ρ is identical to the derived primitive mode of **exp** in ρ.

Examples of Production Systems

We now consider two ASPLE programs, the first of which is syntactically legal, and the second of which is not. The two programs differ only in the declared modes of B.

program 1	program 2
begin	*begin*
int A;	*int A*;
ref int B;	*int B*;
ref ref int C;	*ref ref int C*;
A := 100;	*A := 100*;
B := A;	*B := A*;
C := B;	*C := B*;
input C;	*input C*;
output A	*output A*
end	*end*

Using the productions for <u>**DERIVED ENV**</u> [PS26]–[PS27], the environments for the two programs are:

$\underline{\rho_1}$	$\underline{\rho_2}$
$\{A \to$ **REF INTEGER**,	$\{A \to$ **REF INTEGER**,
$B \to$ **REF REF INTEGER**,	$B \to$ **REF INTEGER**,
$C \to$ **REF REF REF INTEGER**$\}$	$C \to$ **REF REF REF INTEGER**$\}$

From the premise **LEGAL**$\langle$**st**$:\rho\rangle$ in the production for **PROGRAM** [PS01], the statement trains are legal only if the statement trains are legal using ρ_1 and ρ_2, respectively. Using the production for **STM TRAIN** [PS05], each statement in a statement train is legal only if each individual statement is legal using ρ_1 and ρ_2, respectively.

Using the production for **ASGT STM** [PS07], a statement of the form:

$$\mathbf{id} := \mathbf{exp}$$

is legal in ρ if

1) $\mathbf{dm}_\ell$ is the derived mode of **id** in ρ, and
2) $\mathbf{dm}_r$ is the derived mode of **exp** in ρ, and
3) the primitive modes obtained from $\mathbf{dm}_\ell$ and $\mathbf{dm}_r$ are identical, and
4) the number of references in $\mathbf{dm}_\ell$ is less than or equal to 1 plus the number of references in $\mathbf{dm}_r$.

For programs 1) and 2), the statement "A := *100*" is legal, since for both ρ_1 and ρ_2:

$$\begin{aligned}\mathbf{dm}_\ell &\equiv \underline{\textbf{DERIVED EXP MODE}}\ (A : \rho)\\ &\equiv \textbf{REF INTEGER}\\ \mathbf{dm}_r &\equiv \underline{\textbf{DERIVED EXP MODE}}\ (\mathit{100} : \rho)\\ &\equiv \textbf{INTEGER}\end{aligned}$$

$$\underline{\textbf{PRIM MODE}}(\mathbf{dm}_\ell) \equiv \textbf{INTEGER}$$

$$\underline{\textbf{PRIM MODE}}(\mathbf{dm}\ ) \equiv \textbf{INTEGER}$$

$$\begin{aligned}\mathbf{n}_\ell &\equiv \underline{\textbf{NUM REFS}}(\mathbf{dm}_\ell)\\ &\equiv 1\\ \mathbf{n}_r &\equiv \underline{\textbf{NUM REFS}}(\mathbf{dm}_r)\\ &\equiv 0\\ \mathbf{n}_\ell &= \mathbf{n}_r + 1\end{aligned}$$

On the other hand, the assignment "$C := B$" is legal in ρ_1, but *not* in ρ_2, since:

$\underline{\text{for } \rho_1}$	$\underline{\text{for } \rho_2}$
$\mathbf{n}_\ell \equiv 3$	$\mathbf{n}_\ell \equiv 3$
$\mathbf{n}_r \equiv 2$	$\mathbf{n}_r \equiv 1$
$\mathbf{n}_\ell = \mathbf{n}_r + 1$	$\mathbf{n}_\ell > \mathbf{n}_r + 1$

The productions given in Table 3.1 should now be clear. Foi more detail on the Production Systems notation, see [L3].

Semantics Using the Axiomatic Approach

The Production Systems approach given here relies on another language for defining semantics. The only role of Production Systems in defining "semantics" is the specification of a mapping from legal programs into a target language that expresses the meaning of a program. In this subsection, we use the axiomatic approach of Hoare [H1] as the basis for such a target language. A mapping of syntactically legal ASPLE programs into this target language is given in Table 3.2. Production Systems could be used directly to define semantics by specifying a mapping:

$$\text{program : input file} \;\longrightarrow\; \text{output file}$$

giving the corresponding output file for each input file and each legal program. This approach has not been tried.

The axiomatic approach differs significantly from most semantic approaches in that the method is entirely "synthetic" and thus does not rely on any execution model. To define semantics using an axiomatic approach, the following question is addressed: Upon termination of a program, what assertions can be made? The axiomatic approach of Hoare [H1] is based on the first-order predicate calculus [M3] which permits assertions about the membership of objects in sets and the results of applying operations to objects; for example, the kinds of objects stored on some external medium and the values of expressions. To define the semantics of "programs," a correspondence between programs and the relevant assertions must be defined.

This correspondence has two basic parts: a specification of assertions that can be generated *directly* from the program text, and a specification of points where the *user* must derive new assertions based on those already generated. In the paper by Hoare [H1], this issue is only lightly touched upon. We believe this separation to be important, for it shows the user when to proceed automatically and when to make "mental leaps" in the attempt to prove a program correct. However, there is some research being done on the automatic generation of such deductions.

In the specification of ASPLE semantics here, we adopt the following conventions:

1) <u>**SEM PROG**</u>, <u>**SEM STM**</u>, and <u>**SEM EXP**</u> are the names of Production System functions that map legal ASPLE constructs into assertions.
2) **a**, $\mathbf{a}_1$, $\mathbf{a}_2$, etc., are Production System variables denoting members of the set of assertions. The class of well-formed assertions is not defined here, but may be obtained from [M3].
3) <u>**PROVABLE**</u> is a Production System predicate naming a set of ordered pairs $\langle \mathbf{a}_1 : \mathbf{a}_2 \rangle$, where $\mathbf{a}_1$ and $\mathbf{a}_2$ are assertions. This predicate is true only if $\mathbf{a}_2$ can be derived from $\mathbf{a}_1$ by the *user*. The rules used to derive $\mathbf{a}_2$ from $\mathbf{a}_1$ are those of the predicate calculus.

The first production of Table 3.2 specifies the assertions for programs:

[PT01]

$$\underline{\textbf{SEM PROG}}(\textit{begin}\ \mathbf{dt}\ ;\ \mathbf{st}\ \textit{end}) \equiv \underline{\textbf{true}}\ \{*\}\ \mathbf{a}''$$
$$\leftarrow \mathbf{a} \equiv \underline{\textbf{DERIVED ASSERTIONS}}(\mathbf{dt}) \quad \&$$
$$\mathbf{a}' \equiv \mathbf{a} \wedge \mathbf{a}_{\mathbf{prim}} \wedge (\mathbf{F}_{\mathbf{in}} = \beta) \wedge (\mathbf{F}_{\mathbf{out}} = \underline{\textbf{empty file}}) \quad \&$$
$$\rho \equiv \underline{\textbf{DERIVED ENV}}(\mathbf{dt}) \quad \& \quad \underline{\textbf{SEM STM}}(\mathbf{st}:\rho) = \mathbf{a}'\ \{\mathbf{st}\}\ \mathbf{a}''.$$

[$a_{prim} \equiv a_{int} \wedge a_{bool} \wedge a_{ref} \wedge a_{file}$ *are the respective assertions for integers, booleans, reference and files*]
[β *is the user supplied input file*]

This production may be read as follows. If:

a is the assertion derived from the declare train *dt*, and
$\mathbf{a}_{\mathbf{prim}}$ is the assertion for primitive objects: integers, booleans, references, and files, and
$\mathbf{F}_{\mathbf{in}}$ is the user-supplied input file β, and
$\mathbf{F}_{\mathbf{out}}$ is the empty file, and
$\mathbf{a}'$ is the assertion $\mathbf{a} \wedge \mathbf{a}_{\mathbf{prim}} \wedge (\mathbf{F}_{\mathbf{in}} = \beta) \wedge (\mathbf{F}_{\mathbf{out}} = \underline{\textbf{empty file}})$, and
$\mathbf{a}''$ is the assertion obtained after execution of the statement train, given that $\mathbf{a}'$ is true before execution of the statement train;

then

$\mathbf{a}''$ is the assertion upon termination of the program.

[*Assertions for Programs*]

[PT01] $\underline{\text{SEM PROG}}(\textit{begin}\ dt\ ,\ st\ \textit{end}) \equiv \underline{\text{true}}\ \{*\}\ a''$

$\leftarrow a \equiv \underline{\text{DERIVED ASSERTIONS}}(dt)$ &

$a' \equiv a \wedge a_{prim} \wedge (F_{in} = \beta) \wedge (F_{out} = \underline{\text{empty file}})$ &

$\rho \equiv \underline{\text{DERIVED ENV}}(dt)$ & $\underline{\text{SEM STM}}(st\ \rho) = a'\ \{st\}\ a''$.

[$a_{prim} \equiv a_{int} \wedge a_{bool} \wedge a_{ref} \wedge a_{file}$ *are the respective assertions for integers, booleans, reference and files*]

[β *is the user supplied input file*]

[*Assertions for Declarations*]

[PT02] $\underline{\text{DERIVED ASSERTIONS}}(dcl_1, \ldots, dcl_n) \equiv (a_1 \wedge \ldots \wedge a_n)$

$\leftarrow a_1 \equiv \underline{\text{DERIVED ASSERTIONS}}(dcl_1)$ & . . & $a_n \equiv \underline{\text{DERIVED ASSERTIONS}}(dcl_n)$.

[PT03] $\underline{\text{DERIVED ASSERTIONS}}(m\ id_1, \ldots, id_n) \equiv (id_1 \varepsilon dm) \wedge \ldots \wedge (id_n \varepsilon dm)$

$\leftarrow dm \equiv \underline{\text{DERIVED MODE}}(m)$.

[*Assertions for Statements*]

[PT04] $\underline{\text{SEM STM}}(stm_1, stm_2 \ldots ; stm_n\ \ \rho) \equiv a_1\ \{*\}\ a'_{n+1}$

$\leftarrow$ PROVABLE$\langle a_1\ a'_1 \rangle$ & $\underline{\text{SEM STM}}(stm_1\ \rho) = a'_1\ \{stm_1\}\ a_2$ &

PROVABLE$\langle a_2\ a'_2 \rangle$ & $\underline{\text{SEM STM}}(stm_2\ \rho) = a'_2\ \{stm_2\}\ a_3$ &

. . &

PROVABLE$\langle a_n\ a'_n \rangle$ & $\underline{\text{SEM STM}}(stm_n \cdot \rho) = a'_n\ \{stm_n\}\ a_{n+1}$ &

PROVABLE$\langle a_{n+1} \cdot a'_{n+1} \rangle$.

[*Before or after statements, a new assertion* a'_i *may need to be created and derived from* a_i.]

[PT05] $\underline{\text{SEM STM}}(id_\ell\ \underline{:=}\ id_r : \rho) \equiv a^{id_\ell \downarrow}_{\underline{deref}(id_r, n)} \wedge (\underline{deref}(id_r, n)\ \varepsilon\ dm_r)\ \{*\}\ a$

$\leftarrow dm_\ell \equiv \underline{\text{DERIVED EXP MODE}}(id_\ell\ \rho)$ & $dm_r \equiv \underline{\text{DERIVED EXP MODE}}(id_r\ \rho)$ &

$n_\ell \equiv \underline{\text{NUM REFS}}(dm_\ell)$ & $n_r \equiv \underline{\text{NUM REFS}}(dm_r)$ & $n \equiv (n_r - n_\ell) + 1$.

[*The assignment of an* ***identifier*** *on the right side requires dereferencing to obtain a mode compatible with the identifier on the left side*]

[*Checking that the dereferenced value of* id_r *is contained in* dm_r *ensures that* id_r *is not undefined*]

[PT06] $\underline{\text{SEM STM}}(id\ \underline{:=}\ exp \cdot \rho) \equiv a^{id\downarrow}_{exp'} \wedge (exp'\ \varepsilon\ dm_r)\ \{*\}\ a$

$\leftarrow exp' \equiv \underline{\text{SEM EXP}}(exp . \rho)$ &

$dm_r \equiv \underline{\text{DERIVED EXP MODE}}(exp . \rho)$ & $\underline{\text{NUM REFS}}(dm_r) = 0$.

[*The assignment of an* ***expression*** *that is not an identifier simply changes the value of the target identifier on the left side*]

[*Checking that* exp' *is contained in* dm_r ***ensures*** *that* exp' *is not undefined.*]

TABLE 3.2. PRODUCTION SYSTEM MAPPING LEGAL ASPLE PROGRAMS INTO VERIFICATION RULES

[PT07] $\underline{\text{SEM STM}}(\textit{if}\ \text{exp}\ \textit{then}\ \text{st}\ \textit{fi}\quad \rho) \equiv a\ \{*\}\ a'$

$\leftarrow \text{exp}' \equiv \underline{\text{SEM EXP}}(\text{exp}\ \rho)\ \ \&\ \ \underline{\text{SEM STM}}(\text{st}\ \rho) = a \wedge \text{exp}'\ \{\text{st}\}\ a'\ \ \&$

$\text{PROVABLE}\langle a \wedge \underline{\text{not}}(\text{exp}')\quad a'\rangle.$

[PT08] $\underline{\text{SEM STM}}(\textit{if}\ \text{exp}\ \textit{then}\ \text{st}_1\ \textit{else}\ \text{st}_2\ \textit{fi}\quad \rho) \equiv a\ \{*\}\ a'$

$\leftarrow \text{exp}' \equiv \underline{\text{SEM EXP}}(\text{exp}\ \rho)\ \ \&\ \ \underline{\text{SEM STM}}(\text{st}_1\ \rho) = a \wedge \text{exp}'\{\text{st}_1\}\ a'\ \ \&$

$\underline{\text{SEM STM}}(\text{st}_2\ \rho) = a \wedge \underline{\text{not}}(\text{exp}')\ \{\text{st}_2\}\ a'.$

[PT09] $\underline{\text{SEM STM}}(\textit{while}\ \text{exp}\ \textit{do}\ \text{st}\ \textit{end}\quad \rho) \equiv a\ \{*\}\ a_{inv} \wedge \underline{\text{not}}(\text{exp}')$

$\leftarrow \text{exp}' \equiv \underline{\text{SEM EXP}}(\text{exp}\ \rho)\ \ \&\ \ \text{PROVABLE}\langle a \cdot a_{inv}\rangle\ \ \&$

$\underline{\text{SEM STM}}(\text{st}\ \rho) = a_{inv} \wedge \text{exp}'\{\text{st}\}\ a_{inv}.$

[a_{inv} *is the invariant for the loop*]

[PT10] $\underline{\text{SEM STM}}(\textit{input}\ \text{id}\ .\ \rho) \equiv \left(\underline{\text{not}}(\underline{\text{eof}}(F_{in})) \wedge (\underline{\text{first}}(F_{in}) \in \text{dm}) \wedge \text{id}' = \underline{\text{deref}}(\text{id},n) \wedge a^{\text{id}'\downarrow\ \ F_{in}}_{\underline{\text{first}}(F_{in})\ \underline{\text{rest}}(F_{in})}\right)\ \{*\}\ a.$

$\leftarrow \text{dm} \equiv \underline{\text{DERIVED PRIM MODE}}(\text{id}\quad \rho)\ \ \&\ \ n \equiv \underline{\text{NUM REFS}}(\text{dm}) - 1.$

[*The first value in* F_{in} *must be compatible with the mode of id*]

[*Dereferencing id by n refs must yield an identifier*]

[PT11] $\underline{\text{SEM STM}}(\textit{output}\ \text{exp}\quad \rho) \equiv a^{F_{out}}_{\underline{\text{cat}}(F_{out},\ \text{exp}')}\ \{*\}\ a$

$\leftarrow \text{exp}' \equiv \underline{\text{SEM EXP}}(\text{exp}\ \rho).$

[*Assertions for Expressions*]

[PT12] $\underline{\text{SEM EXP}}(\text{exp} + \text{fac}\quad \rho) \equiv \underline{\text{sum}}(\text{exp}',\text{fac}')$

$\leftarrow \text{exp}' \equiv \underline{\text{SEM EXP}}(\text{exp}.\rho)\ \ \&\ \ \text{fac}' \equiv \underline{\text{SEM EXP}}(\text{fac}\cdot\rho)\ \ \&$

$\underline{\text{DERIVED PRIM MODE}}(\text{exp}\ \rho) = \text{INTEGER}.$

[PT13] $\underline{\text{SEM EXP}}(\text{exp} + \text{fac}\quad \rho) \equiv \underline{\text{or}}(\text{exp}',\text{fac}')$

$\leftarrow \text{exp}' \equiv \underline{\text{SEM EXP}}(\text{exp}\ \rho)\ \ \&\ \ \text{fac}' \equiv \underline{\text{SEM EXP}}(\text{fac}\ \rho)\ \ \&$

$\underline{\text{DERIVED PRIM MODE}}(\text{exp}.\rho) = \text{BOOLEAN}.$

[PT14] $\underline{\text{SEM EXP}}(\text{fac}\ \underline{*}\ \text{prim}:\rho) \equiv \underline{\text{product}}(\text{fac}',\text{prim}')$

$\leftarrow \text{fac}' \equiv \underline{\text{SEM EXP}}(\text{fac}\cdot\rho)\ \ \&\ \ \text{prim}' \equiv \underline{\text{SEM EXP}}(\text{prim}\cdot\rho)\ \ \&$

$\underline{\text{DERIVED PRIM MODE}}(\text{prim}.\rho) = \text{INTEGER}.$

[PT15] $\underline{\text{SEM EXP}}(\text{fac}\ \underline{*}\ \text{prim}\cdot\rho) \equiv \underline{\text{and}}(\text{exp}',\text{prim}')$

$\leftarrow \text{exp}' \equiv \underline{\text{SEM EXP}}(\text{fac}\ \rho)\ \ \&\ \ \text{prim}' \equiv \underline{\text{SEM EXP}}(\text{prim}\ \rho)\ \ \&$

$\underline{\text{DERIVED PRIM MODE}}(\text{fac}.\rho) = \text{BOOLEAN}.$

TABLE 3.2.—*Continued*

[PT16] SEM EXP((exp_1 = exp_2) ρ) ≡ equal(exp'_1, exp'_2)
  ← exp'_1 ≡ SEM EXP(exp_1 ρ) & exp'_2 ≡ SEM EXP(exp_2.ρ).

[PT17] SEM EXP((exp_1 ≠ exp_2) ρ) ≡ not(equal(exp'_1, exp'_2))
  ← exp'_1 ≡ SEM EXP(exp_1 ρ) & exp'_2 ≡ SEM EXP(exp_2 ρ).

[PT18] SEM EXP((exp) . ρ) ≡ exp'
  ← exp' ≡ SEM EXP(exp ρ).

[PT19] SEM EXP(id ρ) ≡ deref(id,n)↓
  ← dm ≡ DERIVED EXP MODE(id·ρ) & n = NUM REFS(dm) - 1
  [*Identifiers in expressions must be fully dereferenced*]

[PT20] SEM EXP(int·ρ) ≡ int.

[PT21] SEM EXP(*true* ρ) ≡ true.

[PT22] SEM EXP(*false*·ρ) ≡ false.

[*The assertions* a_{prim} *for primitive values are* $a_{int} \wedge a_{bool} \wedge a_{ref} \wedge a_{file}$]

[*Assertions* a_{int} *for integers*; IMAX *is the implementation defined quantity* n_5]

[PT23] 0, IMAX ε INTEGER

[PT24] (int ≠ IMAX) ⊃ succ(int) ε INTEGER.

[PT25] (int = IMAX) ⊃ . . [*Implementation defined result upon arithmetic overflow*]

[PT26] (int ≠ 0) ⊃ pred(int) ε INTEGER.

[PT27] sum(int,0) = int.

[PT28] (int_1 ≠ IMAX) ∧ (int_2 ≠ 0) ⊃ sum(int_1,int_2)
  = sum(succ(int_1),pred(int_2))

  ... [*The conventional axioms for non-negative integers*]

[*Assertions* a_{bool} *for booleans*]

[PT29] true,false ε BOOLEAN.

[PT30] and(true,true) = true.

[PT31] and(true,false) = false.

  ... [*The conventional axioms for booleans*]

[*Assertions* a_{ref} *for dereferencing*, v ε IDENTIFIER ∪ INTEGER ∪ BOOLEAN]

[PT32] deref(v,0) = v

[PT33] (id↓ = v) ∧ (n ≥ 1) ⊃ (deref(id,n) = deref(v,n-1))

TABLE 3.2.—*Continued*

[*Assertions* a_{file} *for input and output files*]

[ib ε INTEGER ∪ BOOLEAN, f ε FILE]

[PT34] <u>empty file</u> ε FILE.

[PT35] n_6 > <u>FILELENGTH</u>(f) ⊃ <u>cat</u>(ib,f) ε FILE.

[*<u>FILELENGTH</u> is the implementation defined function for computing the file length* n_6]

[PT36] <u>first</u>(<u>cat</u>(ib,f)) = ib.

[PT37] <u>rest</u>(<u>cat</u>(ib,f)) = f.

[PT38] <u>eof</u>(<u>empty file</u>) = <u>true</u>.

[PT39] <u>eof</u>(<u>cat</u>(ib,f)) = <u>false</u>.

TABLE 3.2.—*Continued*

The assertions derived from ASPLE declare trains [PT03] are simply the assertions of set membership for each declared identifier. For example, the declare train:

ref int A;
ref bool B

yields the assertion

$$(A \in \textbf{REF REF INTEGER}) \;\wedge\; (B \in \textbf{REF REF BOOLEAN})$$

Each statement in a statement train gives rise to a production of the form:

$$\underline{\textbf{SEM STM}}(\textbf{stm}) \;\equiv\; \mathbf{a}_1 \,\{*\}\, \mathbf{a}_2 \;\leftarrow \mathbf{p}_1, \mathbf{p}_2, \cdots, \mathbf{p}_n.$$

Here $\mathbf{a}_1$ is any assertion that is true before execution of the statement; $\mathbf{a}_2$ is the assertion derived from $\mathbf{a}_1$ after execution of the statement; and $\mathbf{p}_1$ through $\mathbf{p}_n$ are Production System predicates that must be true in order to generate $\mathbf{a}_2$ from $\mathbf{a}_1$.

The semantics of assignment statements and *while-do* statements are particularly important. For assignment of *identifiers*, we have:

[PT05]

$$\textbf{SEM STM}(\mathbf{id}_\ell := \mathbf{id}_r{:}\rho) \;\equiv\; \mathbf{a}^{\mathbf{id}_\ell\downarrow}_{\underline{\mathbf{deref}}(\mathbf{id}_r,\,\mathbf{n})} \wedge (\underline{\mathbf{deref}}(\mathbf{id}_r, \mathbf{n}) \in \mathbf{dm}_r) \,\{*\}\, \mathbf{a}$$
$$\leftarrow \mathbf{dm}_\ell \equiv \underline{\textbf{DERIVED EXP MODE}}(\mathbf{id}_\ell{:}\rho) \;\&\; \mathbf{dm}_r \equiv \underline{\textbf{DERIVED EXP MODE}}(\mathbf{id}_r{:}\rho) \;\&$$
$$\mathbf{n}_\ell \equiv \underline{\textbf{NUM REFS}}(\mathbf{dm}_\ell) \;\&\; \mathbf{n}_r \equiv \underline{\textbf{NUM REFS}}(\mathbf{dm}_r) \;\&\; \mathbf{n} \equiv (\mathbf{n}_r - \mathbf{n}_\ell) + 1.$$

[*The assertion of an <u>identifier</u> requires dereferencing the identifier to obtain a mode <u>compatible</u> with the identifier on the left side*]
[*Checking that the dereferenced value of* id_r *is contained in* dm_r *insures that* id_r *is not undefined.*]

This production may be read as follows: The assertion **a** may be derived from the assertion:

$$\mathbf{a}^{\mathbf{id}_\ell\downarrow}_{\underline{\mathbf{deref}}(\mathbf{id}_r,\,\mathbf{n})} \;\wedge\; (\underline{\mathbf{deref}}(\mathbf{id}_r, \mathbf{n}) \in \mathbf{dm}_r)$$

if:

$\mathbf{dm}_\ell$ and $\mathbf{dm}_r$ are the derived modes of $\mathbf{id}_\ell$ and $\mathbf{id}_r$ in ρ, and
$\mathbf{n}_\ell$ and $\mathbf{n}_r$ are the number of refs in $\mathbf{dm}_\ell$ and $\mathbf{dm}_r$, and
$\mathbf{n}$ equals $(\mathbf{n}_r - \mathbf{n}_\ell) + 1$.

The arrow pointing downward, "↓," denotes a reference to a value. In general, the notation $\mathbf{a}^{\mathbf{x}}_{\mathbf{y}}$ denotes the assertion obtained from **a** by replacing occurrences of **x** by **y**. In the preceding production, **y** is <u>**deref**</u>($\mathbf{id}_r$, **n**), that is, the dereferenced value of $\mathbf{id}_r$. The assertion that $\mathbf{y} \in \mathbf{dm_r}$ insures that this value must be well defined, that is, not undefined. In a sense, the proof rule for assignment appears to be the wrong way around, for the assertion replacing $\mathbf{id}_\ell \downarrow$ by a value must be derivable before the statement. This initially counter-intuitive definition reflects two facts:

- the dereferenced value of $\mathbf{id_r}$ must be obtained before the statement is executed;
- any invariant derived after execution of the statement must be true when $\mathbf{id}_\ell \downarrow$ is replaced by the deferenced value of $\mathbf{id_r}$ before execution of the statement.

For assignment of *expressions* (that are not identifiers), we have

[PT06]

$$\underline{\textbf{SEM STM}}(\mathbf{id} \underline{:=} \mathbf{exp} : \rho) \equiv \mathbf{a}_{\mathbf{exp}'} \wedge (\mathbf{exp}' \in \mathbf{dm_r}) \{*\} \mathbf{a}$$
$$\leftarrow \mathbf{exp}' \equiv \underline{\textbf{SEM EXP}}(\mathbf{exp} \cdot \rho) \;\&$$
$$\mathbf{dm_r} \equiv \underline{\textbf{DERIVED EXP MODE}}(\mathbf{exp} \cdot \rho) \;\&\; \underline{\textbf{NUM REFS}}(\mathbf{dm_r}) = 0.$$

[*The assignment of an* <u>*expression*</u> *that is not an identifier simply changes the value of the target identifier on the left side*]
[*Checking that exp′ is contained in* dm_r *ensures that exp′ is not undefined.*]

The assignment of expressions can only be made to identifiers with one syntactically declared reference. Since the rules for expression semantics result in primitive values that are integers or Booleans (with zero references), generation of the new assertion results from a simple replacement.

For *while-do* loops, the rule is:

[PT09]

$$\underline{\textbf{SEM STM}}(\textit{while } \mathbf{exp} \textit{ do } \mathbf{st} \textit{ end} : \rho) \equiv \mathbf{a} \{*\} \mathbf{a_{inv}} \wedge \underline{\textbf{not}}(\mathbf{exp}')$$
$$\leftarrow \mathbf{exp}' \equiv \underline{\textbf{SEM EXP}}(\mathbf{exp} : \rho) \;\&\; \textbf{PROVABLE}\langle \mathbf{a} : \mathbf{a_{inv}} \rangle \;\&$$
$$\underline{\textbf{SEM STM}}(\mathbf{st} : \rho) = \mathbf{a_{inv}} \wedge \mathbf{exp}' \{\mathbf{st}\} \mathbf{a_{inv}}.$$

[a_{inv} *is the invariant for the loop*]

Here the predicate **PROVABLE** must be used to derive the loop invariant $\mathbf{a_{inv}}$ from any assertion **a** that is true before the loop, and $\underline{\textbf{SEM STM}}(\mathbf{st} : \rho)$ must be shown to not alter the truth of $\mathbf{a_{inv}}$ when the value of $\mathbf{exp}'$ is true. The invariant $\mathbf{a_{inv}}$ must be devised by the user. The creation of this invariant is the major mental leap required by the user in the correctness proofs of ASPLE programs.

For statement trains [PT04], the generation of a terminal assertion involves two steps:

- the generation of an assertion $\mathbf{a_i}'$ obtained from the assertion $\mathbf{a_i}$ from the previous statement or declaration;
- a proof that the assertion $\mathbf{a_{i+1}}$ after each statement is provable from the assertion $\mathbf{a_i}'$ obtained from execution of the previous statement.

In particular, the semantics of a statement train is specified in [PT04]

[PT04]

$$\underline{\textbf{SEM STM}}(\mathbf{stm_1}; \mathbf{stm_2} \ldots ; \mathbf{stm_n} : \rho) \equiv \mathbf{a_1} \{*\} \mathbf{a'_{n+1}}$$
$$\leftarrow \textbf{PROVABLE}\langle \mathbf{a_1} : \mathbf{a_1}' \rangle \;\&\; \underline{\textbf{SEM STM}}(\mathbf{stm_1} : \rho) = \mathbf{a_1}' \{\mathbf{stm_1}\} \mathbf{a_2} \;\&$$
$$\textbf{PROVABLE}\langle \mathbf{a_2} : \mathbf{a_2}' \rangle \;\&\; \underline{\textbf{SEM STM}}(\mathbf{stm_2} : \rho) = \mathbf{a_2}' \{\mathbf{stm_2}\} \mathbf{a_3} \;\&$$
$$\ldots \;\&$$
$$\textbf{PROVABLE}\langle \mathbf{a_n} : \mathbf{a_n}' \rangle \;\&\; \underline{\textbf{SEM STM}}(\mathbf{stm_n} : \rho) = \mathbf{a_n}' \{\mathbf{stm_n}\} \mathbf{a_{n+1}} \;\&$$
$$\textbf{PROVABLE}\langle \mathbf{a_{n+1}} : \mathbf{a'_{n+1}} \rangle.$$

[*Before or after statements, a new assertion* a_i' *may need to be created and derived from* a_i]

The creation of new assertions $\mathbf{a_1}', \mathbf{a_2}', \cdots, \mathbf{a_n}'$, and $\mathbf{a'_{n+1}}$ that are provable from $\mathbf{a_1}, \mathbf{a_2}, \cdots, \mathbf{a_n}, \mathbf{a_{n+1}}$ reflect the mental leaps required by the user regarding proofs about subsequent statements.

The semantics for ASPLE expressions are quite straightforward. For numeric expressions, for example:

[PT12] $\underline{\mathbf{SEM\ EXP}}(\mathbf{exp} + \mathbf{fac} : \rho) \equiv \underline{\mathbf{sum}}(\mathbf{exp'}, \mathbf{fac'})$
$\leftarrow \mathbf{exp'} \equiv \underline{\mathbf{SEM\ EXP}}(\mathbf{exp} : \rho) \ \ \& \ \ \mathbf{fac'} \equiv \underline{\mathbf{SEM\ EXP}}(\mathbf{fac} : \rho) \ \ \&$
$\underline{\mathbf{DERIVED\ PRIM\ MODE}}(\mathbf{exp} : \rho) = \mathbf{INTEGER}.$

[PT14] $\underline{\mathbf{SEM\ EXP}}(\mathbf{fac} \ \underline{*} \ \mathbf{prim} : \rho) \equiv \underline{\mathbf{product}}(\mathbf{fac'}, \mathbf{prim'})$
$\leftarrow \mathbf{fac'} \equiv \underline{\mathbf{SEM\ EXP}}(\mathbf{fac} : \rho) \ \ \& \ \ \mathbf{prim'} \equiv \underline{\mathbf{SEM\ EXP}}(\mathbf{prim} : \rho) \ \ \&$
$\underline{\mathbf{DERIVED\ PRIM\ MODE}}(\mathbf{prim} : \rho) = \mathbf{INTEGER}.$

the basic axioms for "**sum**" and "**product**" over positive integers follow the usual rules for finite arithmetic:

[PT23] $0, \mathrm{IMAX} \in \mathbf{INTEGER}.$

[PT24] $(\mathbf{int} \neq \mathrm{IMAX}) \supset \underline{\mathbf{succ}}(\mathbf{int}) \in \mathbf{INTEGER}.$

[PT26] $(\mathbf{int} \neq 0) \supset \underline{\mathbf{pred}}(\mathbf{int}) \in \mathbf{INTEGER}.$

and so forth. The number IMAX is the implementation defined maximum integer η_5.

For deferencing identifiers we have:

[PT32] $$\underline{\mathbf{deref}}(\mathbf{v}, 0) = \mathbf{v}$$

that is, dereferencing a value by zero refs yields the value itself, and

[PT33] $$(\mathbf{id} \downarrow = \mathbf{v}) \wedge (\mathbf{n} \geq 1) \supset (\underline{\mathbf{deref}}(\mathbf{id}, \mathbf{n}) = \underline{\mathbf{deref}}(\mathbf{v}, \mathbf{n} - 1))$$

that is, dereferencing a value by **n** refs results in removal of **n** refs.

The axioms for ASPLE files are straightforward, and are given in Table 3.2.

As a final note observe that in the axiomatic approach a program may have many "semantics" in the sense that several mutually consistent final assertions are derivable from a given program.

Examples of the Axiomatic Approach

Consider the following simple ASPLE program:

```
begin                          [01]
  int N, I, SUM;               [02]
  N := 10;                     [03]
  I := 0;                      [04]
  SUM := 0;                    [05]
  while (I ≠ N) do             [06]
    I := I + 1;                [07]
    SUM := SUM + I             [08]
  end;                         [09]
  output SUM                   [10]
end                            [11]
```

For an empty input file β, productions [PT01] through [PT03] specify that

$$\rho \equiv \{N \rightarrow \mathbf{REF\ INTEGER}, I \rightarrow \mathbf{REF\ INTEGER}, SUM \rightarrow \mathbf{REF\ INTEGER}\}$$

$$\mathbf{a} \equiv (N \in \mathbf{REF\ INTEGER}) \wedge (I \in \mathbf{REF\ INTEGER}) \wedge (SUM \in \mathbf{REF\ INTEGER})$$

$$\mathbf{a'} \equiv \mathbf{a} \wedge \mathbf{a_{prim}} \wedge (\mathbf{F_{in}} = \underline{\mathbf{empty\ file}}) \wedge (\mathbf{F_{out}} = \underline{\mathbf{empty\ file}})$$

$$\underline{\mathbf{SEM\ STM}}(\mathbf{st} : \rho) \equiv \mathbf{a'}\{\mathbf{st}\}\mathbf{a''}$$

The semantics of the program are specified by deriving $\mathbf{a''}$, where **st** is the statement train in lines [03] through [10].

The semantics of statement trains allow the creation and derivation of new assertions before using the semantic rules for each contained statement. From $\mathbf{a}'$ we may create and (trivially) derive the assertion

$$\mathbf{a}'_{03} \equiv (\mathbf{a}' \wedge (N\downarrow = 10))^{N\downarrow}_{10}$$

Using production [PT05] for assignment after line [03], we may immediately derive

$$\mathbf{a}_{04} \equiv \mathbf{a}' \wedge (N\downarrow = 10)$$

Similarly, we may derive

$$\mathbf{a}_{06} \equiv \mathbf{a}_{04} \wedge (I\downarrow = 0) \wedge (SUM\downarrow = 0)$$

Before execution of the *while* loop, the loop invariant must be created. This invariant is

$$\mathbf{a}_{\mathrm{inv}} \equiv \mathbf{a}_{04} \wedge (I\downarrow \leq N\downarrow) \wedge \left(SUM\downarrow = \left(\sum_{k=0}^{k=I\downarrow} k\right)\right)$$

This major mental leap is based on a proper abstraction from the *while* loop, that is, that the assertion $\mathbf{a}_{04}$ remains unchanged, that $I\downarrow$ is always less than or equal to $N\downarrow$, and that $SUM\downarrow$ represents the sum of integers up to $I\downarrow$. This invariant is easily provable from $\mathbf{a}_{06}$, where $I\downarrow = 0$.

From production [PT09] we must now prove that the statement train in the body of the loop preserves the invariant $\mathbf{a}_{\mathrm{inv}}$, that is,

$$\mathbf{a}_{\mathrm{inv}} \wedge \underline{\mathbf{not}}(\underline{\mathbf{equal}}\ (I\downarrow, N\downarrow)) \quad \{\mathbf{st}\} \quad \mathbf{a}_{\mathrm{inv}}$$

Since from

$$\mathbf{a}_{\mathrm{inv}} \wedge \underline{\mathbf{not}}(\underline{\mathbf{equal}}\ (I\downarrow, N\downarrow))$$

we can readily make a mental leap to the assertion

$$\mathbf{a}'_{07} \equiv \left(\mathbf{a}_{04} \wedge (I\downarrow < N\downarrow + 1) \wedge \left(SUM\downarrow = \left(\sum_{k=0}^{k=I\downarrow-1} k\right)\right)\right)^{I\downarrow}_{I\downarrow+1}$$

after execution of statement [07] we have

$$\mathbf{a}_{08} \equiv \mathbf{a}_{04} \wedge (I\downarrow < N\downarrow + 1) \wedge \left(SUM\downarrow = \left(\sum_{k=0}^{k=I\downarrow-1} k\right)\right)$$

Similarly, after execution of statement [08], we have

$$\mathbf{a}_{09} \equiv \mathbf{a}_{04} \wedge (I\downarrow < N\downarrow + 1) \wedge \left(SUM\downarrow = \left(\sum_{k=0}^{k=I\downarrow-1} k\right) + I\downarrow\right)$$

from which we can create and derive the assertion

$$\mathbf{a}'_{09} \equiv \mathbf{a}_{04} \wedge (I\downarrow \leq N\downarrow) \wedge \left(SUM\downarrow = \left(\sum_{k=0}^{k=I\downarrow} k\right)\right.$$

which is precisely the loop invariant $\mathbf{a}_{\mathrm{inv}}$.

Accordingly, the semantics of the entire loop is specified as

$$\mathbf{a}_{\mathrm{inv}} \wedge \underline{\mathbf{equal}}\ (I\downarrow, N\downarrow)$$

from which we may assert

$$SUM\downarrow = \left(\sum_{k=0}^{k=N\downarrow} k\right) = \left(\sum_{k=0}^{k=10} k\right) = 55$$

Production [PT11] thus specifies that

$$\mathbf{F}_{\mathrm{out}} = \underline{\mathbf{cat}}(55, \underline{\mathbf{empty\ file}})$$

which is the desired result.

The major issue left in the semantics of ASPLE is that of indirect addressing. Consider the program given in Section 1, Informal Description of ASPLE [see page 197]:

```
begin                                          [01]
  int INTA, INTB:                              [02]
  ref int REFINTA, REFINTB;                    [03]
  ref ref int REFREFINTA, REFREFINTB;          [04]
    INTA := 100;                               [05]
    INTB := 200;                               [06]
    REFINTA := INTA;                           [07]
    REFINTB := INTB;                           [08]
    REFREFINTA := REFINTA;                     [09]
    REFINTA := INTB;                           [10]
    INTB := REFREFINTA;                        [11]
    input REFREFINTA;                          [12]
    output REFINTB                             [13]
end                                            [14]
```

For β, the user-supplied input file, equal to **cat**(300, **empty file**) after the statement [13] we have the (partial) assertion

$$(INTA \downarrow\ = 100) \quad \wedge \quad (INTB \downarrow\ = 300) \quad \wedge \quad (REFINTA \downarrow\ = INTB) \quad \wedge$$
$$(REFINTB \downarrow\ = INTB) \wedge (REFREFINTA \downarrow\ = REFINTA) \quad \wedge$$
$$(\mathbf{F}_{\mathbf{in}} = \underline{\mathbf{empty\ file}}) \wedge (\mathbf{F}_{\mathbf{out}} = \underline{\mathbf{cat}}(300, \underline{\mathbf{empty\ file}})$$

The generation of this assertion from Table 3.2 is left to the reader.

Finally, we discuss one important point. In the Production System given in Table 3.2, no explicit mention is made of cases where syntactically legal programs result in semantic errors. Like BNF and Production Systems with regard to the specification of syntax, semantic errors in the axiomatic approach can be deduced only by the impossibility of deriving a valid result. For example, in the semantic definition of assignment statements, the attempt to evaluate an arithmetic expression containing an undefined identifier results in an execution error. This error can only be deduced by observing that no assertions can be derived from an identifier whose dereferenced value is not defined.

4. VIENNA DEFINITION LANGUAGE

One of the earliest proposals for the rigorous definition of a programming language was Garwick's suggestion that an actual implementation be used [G1]. Two major objections to this technique are: 1) the inevitable encroachment of the host hardware into the language being defined; and 2) the restricted availability of the definition. To escape these objections, the IBM Vienna Laboratories developed the idea of a hypothetical machine, as proposed by McCarthy [M1, M2], Landin [L1], and Elgot [E1], on which to make an implementation. This work led to the Vienna Definition Language (VDL) and was used originally for a formal definition of PL/I [L6].

Overview of VDL

In VDL a formal definition is based on the concept of an abstract machine (see Figure 4). The meaning of a program is defined by the sequence of changes in the state of the abstract machine as the program is executed. The rules of execution are defined by an algorithm, the Interpreter. To make a distinction between those properties of a program that can be determined statically and those that are intrinsically connected to the dynamics of the program's execution, the original program is transformed into an abstracted form before execution. This transformation is performed by another algorithm, the Translator, which corresponds to the early phases of a compiler in a real computer system. During the transformation, the context-sensitive requirements on syntax can be checked.

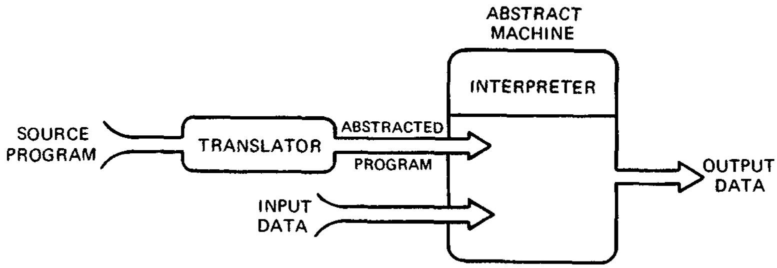

Figure 4. Schematic of a programming language definition in VDL.

The notation of VDL is fully defined in [L4, L6, L7, W1]. In this section we give a brief description of notation, introducing only those parts that are needed for the definition of ASPLE.

In VDL both the abstract machine and the program are *objects*. An object can be represented as a tree. There are two classes of objects: *elementary objects*, with no components; and *composite objects*, with a finite number of immediate components that are also objects. Thus, in the tree representation, an elementary object is a terminal node and a composite object is a nonterminal or branch node.

Figure 5 shows a representation of a composite object named **A**. This object has three immediate components, each uniquely named by its *selector*, χ_1, χ_2, or χ_3. We denote the immediate component χ_1 of **A** by $\chi_1(\mathbf{A})$. This is the elementary object **B**. Similarly, we denote the elementary object **D** by $\chi_4 \cdot \chi_3(\mathbf{A})$ since **D** is the χ_4 component of $\chi_3(\mathbf{A})$. The selector $\chi_4 \cdot \chi_3$ is a *composite selector*. The application of a selector to an object that has no selector of that name yields the null object, denoted by Ω. For example, $\chi_7(\mathbf{A}) = \Omega$ and $\chi_3 \cdot \chi_2(\mathbf{A}) = \Omega$.

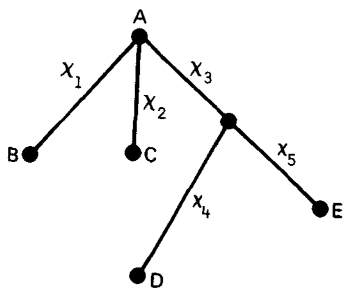

Figure 5. Composite VDL object.

The composite object $\chi_3(\mathbf{A})$ has two components named χ_4 and χ_5. These components are the elementary objects **D** and **E**, respectively. We may describe the construction of $\chi_3(\mathbf{A})$ by showing it as a set of two selector-object pairs:

$$\chi_3(\mathbf{A}) \equiv (\langle \chi_4 : \mathbf{D} \rangle, \langle \chi_5 : \mathbf{E} \rangle)$$

Similarly, we can show the construction of the composite object **A** by a set of three selector-object pairs:

$$\mathbf{A} \equiv (\langle\chi_1 : \mathbf{B}\rangle, \langle\chi_2 : \mathbf{C}\rangle, \langle\chi_3 : (\langle\chi_4 : \mathbf{D}\rangle, \langle\chi_5 : \mathbf{E}\rangle)\rangle)$$

The object in the third of these pairs is the composite object $\chi_3(\mathbf{A})$ whose composition was shown earlier.

To specify subclasses of the class of objects, VDL uses predicates that are true for members of the subclass and are false for all other objects. All such predicates have the prefix **is-**, for example, **is-**$\Omega(\mathbf{Z})$ will be true if and only if **Z** is a null object.

Objects can be modified by using the μ operator. The result of $\mu(\mathbf{A}: \langle\chi_1 : \mathbf{F}\rangle)$ is an object constructed from a copy of **A** by:
1) deleting the component $\chi_1(\mathbf{A})$, if it exists;
2) adding a component $\langle\chi_1 : \mathbf{F}\rangle$.

The result of $\mu(\mathbf{A}: \langle\chi_1 : \mathbf{F}\rangle)$, where **A** is the object **A** of Figure 5, is a copy of **A** with the elementary object **B** replaced by the elementary object **F**.

A special case of the μ operation is the μ_0 operator which constructs a new object from a set of selector-object pairs. For example, the object **A** can be constructed by:

$$\mu_0(\langle\chi_1 : \mathbf{B}\rangle, \langle\chi_2 : \mathbf{C}\rangle, \langle\chi_3 : (\langle\chi_4 : \mathbf{D}\rangle, \langle\chi_5 : \mathbf{E}\rangle)\rangle)$$

Objects that represent lists are often used in VDL. If **L** is an object that represents a list of $\boldsymbol{n}$ objects, *none of them null*, then the elements of **L** are named by the selectors, **elem**$(\boldsymbol{i})$, $1 \leq \boldsymbol{i} \leq \boldsymbol{n}$. For such a list **L**, VDL also makes use of the elementary functions:

length (L)	value $\boldsymbol{n}$;
head (L)	object selected by **elem**(1);
tail (L)	objects **elem**$(\boldsymbol{i})$ (L), $2 \leq \boldsymbol{i} \leq \boldsymbol{n}$, in the form of a list with selectors **elem**(j), $1 \leq \mathbf{j} \leq \boldsymbol{n}-1$, respectively;
$\mathbf{L}_1 \frown \mathbf{L}_2$	concatenation of the two lists $\mathbf{L}_1$ and $\mathbf{L}_2$ to form a single list.

By convention, all objects that satisfy the predicate **is-x-list** are lists each one of whose components satisfies the predicate **is-x**. The empty list is denoted by $\langle\ \rangle$ and is different from the null object Ω.

Abstract Machine

The abstract machine used to define ASPLE, the ASPLE Machine, is specified by its machine state ξ. This is an object satisfying the predicate **is-state** which is defined by four predicate definitions in Table 4.1. Rule [M10] in this table:

```
[M01]        is-state  = (<program: is-abs-program>,
                                    [abstraction of concrete program]
                          <control: is-abs-control>,
                                    [control of abstract machine]
                          <store:   is-abs-storage>,
                          <input:   is-abs-const-list>,
                                    [input file]
                          <output:  is-abs-const-list>
                                    [output file]
```

shows that a composite object ξ satisfying the predicate **is-state** has five components:

1) **program**: the abstracted program to be interpreted. This component will be described in the Subsection, VDL Representation of Programs [page 237].
2) **control**: the control part of the Machine.
3) **store**: the storage part of the Machine.
4) **input**: the input file.
5) **output**: the output file.

The **control** part determines the action of the ASPLE Machine as the abstracted program is interpreted. The object selected by **control** is an object that satisfies the predicate **is-control**. This is a stack of machine operations that will be described in the Subsection, VDL Interpreter [page 248].

The storage part of the ASPLE Machine is defined by the predicate definition **[M02]**:

[M02] **is-storage** = ({<**id**: **is-abs-value**> || **is-abs-identifier(id)**})
[*each element of the set of components of the storage part is selected by an identifier and is an object satisfying is-abs-value*]

The notation here is similar to set notation and defines the storage part as a finite set of selector-object pairs of the form ⟨**id: is-abs-value**⟩; a selector **id** and an object that satisfies the predicate **is-abs-value**. The "**-abs-**" indicates that the object is part of the abstract machine. The latter part of the definition states that the selector **id** satisfies the predicate **is-abs-identifier**. The value part of the pair represents an object that can be obtained by applying an identifier as a selector to the storage component of the ASPLE Machine.

The predicate **is-abs-value** is defined by the predicate definition **[M03]**:

[M03] **is-abs-value** = **is-abs-const** $\vee$ **is-abs-identifier**

By this rule, an ASPLE value is either a constant or an identifier. The input and output files, **input**(ξ) and **output**(ξ) respectively, are objects satisfying the predicate **is-abs-const-list**. These objects are therefore lists of objects each element of which satisfies **is-abs-const**. By rule **[M04]**:

[M04] **is-abs-const** = **is-abs-boolean** $\vee$ **is-abs-integer**

an object that satisfies **is-abs-const** will be one that satisfies either **is-abs-boolean** or **is-abs-integer**.

The second part of Table 4.1 defines the initial state of the ASPLE Machine, ξ_0:

ξ_0 = μ_0 (<**program**: **translate**(PROG)>,
[*initialized by performing translate function on the concrete program PROG*]
<**control**: **interpret-program**>,
<**store**: Ω >
<**input**: [*input file for program, obtained from a source outside this definition*]>
<**output**: **is**-<>>) [*output file is initially empty*]

The program part of the ASPLE Machine is an abstracted ASPLE program, described in the Subsection, VDL Representation of Programs [page 237]. The program part is initialized by attaching with the selector **program** the object obtained by evaluating the func-

```
[M01]  is-state      =  (<program   is-abs-program>,     [abstraction of concrete program]
                         <control·  is-abs-control>,     [control of abstract machine]
                         <store     is-abs-storage>,
                         <input     is-abs-const-list>,  [input file]
                         <output    is-abs-const-list>)  [output file]

[M02]  is-storage    =  ({<id  is-abs-value>  ||   is-abs-identifier(id)})
                           [each element of the set of components of the storage
                            part is selected by an identifier and is an object
                            satisfying is-abs-value]

[M03]  is-abs-value  =  is-abs-const  v is-abs-identifier

[M04]  is-abs-const  =  is-abs-boolean v  is-abs-integer


                        [INITIAL STATE OF THE ASPLE MACHINE]

ξ0 =  μ0(<program. translate(PROG)>,     [initialized by performing translate function on the
                                          concrete program PROG]
         <control· interpret-program>,
         <store    Ω ,
         <input    [input file for program. obtained from a source outside this definition]>
         <output:  is-<>>)               [The output file is initially empty]
```

TABLE 4.1. DEFINITION OF THE ASPLE MACHINE STATE

tion **translate** with PROG, the VDL representation of the original source program. The Translator is described in the Subsection, VDL Translator [page 241]. The control part of the ASPLE Machine is initialized to the machine operation **interpret-program**, which is described in the Subsection, VDL Interpreter [page 248]. The storage part of the ASPLE Machine is initially empty, reflecting the ASPLE rule that the values of all variables are undefined at the start of execution. The input file is initialized to the input data for the program and the output file is initialized to an empty list.

VDL Representation of Programs

The input to the ASPLE Translator is a class of objects, *concrete-programs*, that satisfy the predicate **is-c-program** defined in Table 4.2. The "**-c-**" indicates that the object is part of the concrete program. This definition is derived directly from the BNF syntax of ASPLE shown in Table 1.1. There is a one-to-one correspondence between concrete programs and the character-string representation of well-formed ASPLE programs.

The definition of concrete programs makes use of certain standard selectors, $\mathbf{s}_1$, $\mathbf{s}_2$, $\cdots$ assumed to be mutually distinguishable. Objects with these selectors are objects whose structure differs from VDL lists only in that some of the components may be null. These objects are referred to as "slists." A function, **length**, that corresponds to the **length** function for VDL lists, gives the minimum value n such that for all $i > n$, $\mathbf{s}_i$ selects the null object.

Informally, the correspondence between the predicate **is-c-program** and the context-free syntax of ASPLE can be seen by comparing production **[B01]** of Table 1.1:

[B01] <**program**> ::= *begin* <**dcl train**> ;
<**stm train**> *end*

```
[C01] is-c-program          = (<s1: is-begin>, <s2: is-c-dcl-train>, <s3: is-;>,
                                                     <s4: is-c-stm-train>, <s5: is-end>)
[C02] is-c-dcl-train        = (<s-del: is-;>, <s1: is-c-declaration>, ... )
[C03] is-c-stm-train        = (<s-del: is-;>, <s1: is-c-statement>, ... )
[C04] is-c-declaration      = (<s1: is-c-mode>, <s2: is-c-idlist>)
[C05] is-c-statement        = is-c-asgt-stm v is-c-cond-stm v is-c-loop-stm v is-c-input-stm
                                                                       v is-c-output-stm
[C06] is-c-mode             = (<s1: is-Ω v (<s1: is-ref>, ... )>, <s2: is-bool v is-int>)
[C07] is-c-idlist           = (<s-del: is-,>, <s1: is-c-id>, ... )
[C08] is-c-asgt-stm         = (<s1: is-c-id>, <s2: is-:=>, <s3: is-c-exp>)
[C09] is-c-cond-stm         = (<s1: is-if>, <s2: is-c-exp>, <s3: is-then>, <s4: is-c-stm-train>,
                                   <s5: is-Ω v is-c-else-part>, <s6: is-fi>)
[C10] is-c-loop-stm         = (<s1: is-while>, <s2: is-c-exp>, <s3: is-do>, <s4: is-c-stm-train>,
                                                                  <s5: is-end>)
[C11] is-c-input-stm        = (<s1: is-input>, <s2: is-c-id>)
[C12] is-c-output-stm       = (<s1: is-output>, <s2: is-c-exp>)
[C13] is-c-else-part        = (<s1: is-else>, <s2: is-c-stm-train>)
[C14] is-c-exp              = is-c-factor v (<s1: is-c-exp>, <s2: is-+>, <s3: is-c-factor>)
[C15] is-c-factor           = is-c-primary v (<s1: is-c-factor>, <s2: is-*>, <s3: is-c-primary>)
[C16] is-c-primary          = is-c-id v is-c-bool-const v is-c-int-const v is-c-parenthesized-exp
[C17] is-c-parenthesized-exp= (<s1: is-(>, <s2: is-c-exp v is-c-compare>, <s3: is-)>)
[C18] is-c-compare          = (<s1: is-c-exp>, <s2: is-= v is-≠>, <s3: is-c-exp>)
[C19] is-c-bool-const       = is-true v is-false
[C20] is-c-int-const        = (<s1: is-c-digit>, ... )
[C21] is-c-id               = (<s1: is-c-letter>, ... )
[C22] is-c-digit            = is-0 v is-1 v ... v is-9
[C23] is-c-letter           = is-A v is-B v ... v is-Z
```

TABLE 4.2. DEFINITION OF PREDICATE IS-C-PROGRAM

with definition [C01] of Table 4.2:

[C01] **is-c-program** = (⟨$\mathbf{s_1}$: **is**-*begin*⟩, ⟨$\mathbf{s_2}$: **is-c-dcl-train**⟩, ⟨$\mathbf{s_3}$: **is**-;⟩,
⟨$\mathbf{s_4}$: **is-c-stm-train**⟩, ⟨$\mathbf{s_5}$: **is**-*end*⟩)

Definition [C01] specifies that an object satisfying **is-c-program** has five immediate components with names $\mathbf{s_1}, \cdots, \mathbf{s_5}$. Three of these components, those selected by $\mathbf{s_1}$, $\mathbf{s_3}$, and $\mathbf{s_5}$, are elementary objects satisfying the predicates **is**-*begin*, **is**-;, and **is**-*end*, respectively. These elementary objects correspond to the *begin*, ; , and *end* shown in [B01]. The other two components are composite objects corresponding to the **⟨dcl train⟩** and **⟨stm train⟩** of Table 1.1.

The component selected by $\mathbf{s_2}$, is an object satisfying **is-c-dcl-train**, defined in rule [C02]:

[C02] **is-c-dcl-train** = (⟨**s-del**: **is**-;⟩, ⟨$\mathbf{s_1}$: **is-c-declaration**⟩, . . .)

This predicate definition shows the VDL convention for representing a sequence of items separated by a delimiter. The special selector **s-del** selects an elementary object representing the delimiter, and $\mathbf{s_1}$, $\mathbf{s_2}$, $\cdots$ select the successive items of the sequence. Thus an object satisfying **is-c-dcl-train** represents a sequence of declarations separated by semicolons. The declarations are represented by objects satisfying **is-c-declaration.** For example, a declare train consisting of three declarations could be represented by the tree shown in Figure 6. Each of the objects $\mathbf{d}_1$, $\mathbf{d}_2$, and $\mathbf{d}_3$ satisfies the predicate **is-c-declaration.**

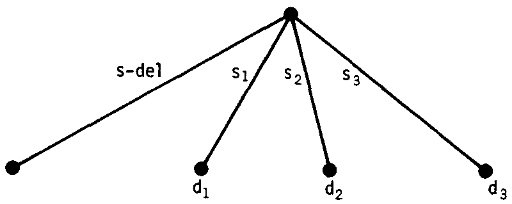

Figure 6. Tree representation of declaration train.

The objects that satisfy **is-c-declaration** have two components defined by the predicates **is-c-mode** and **is-c-idlist**. The first of these predicates is defined in rule [C06]:

[C06] **is-c-mode** = (<s_1 : is-Ω $\vee$ (<s_1 : is-*ref*>, . . .)>, <s_2 : is-*bool* $\vee$ is-*int*>)

An object that satisfies **is-c-mode** has two components. The first is either Ω, the null object, or is a list of elementary objects defined by **is**-*ref*. The second component is an elementary object that satisfies either **is**-*bool* or **is**-*int*. A tree representation of the object that corresponds to the mode declaration:

ref ref ref int

is shown in Figure 7.

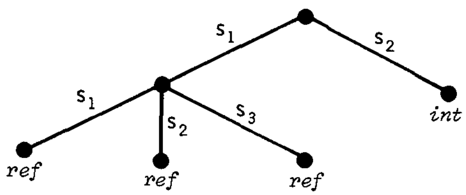

Figure 7. Object satisfying **is-c-mode**

The remainder of Table 4.2 completes the definition of the predicate **is-c-program.** The algorithm that converts the character-string representation of an ASPLE program into the corresponding VDL object is not specified in this definition. Because of the one-to-one correspondence between syntactically correct ASPLE programs and objects satisfying **is-c-program**, Table 4.2 defines the context-free syntax of ASPLE.

The ASPLE program executed by the ASPLE Machine is obtained from concrete programs by removing the syntactic devices that were associated with their character-string representations. These abstracted programs are the essence of the corresponding ASPLE programs. Abstracted programs are objects that satisfy the predicate **is-abs-program** defined in Table 4.3. The definition of the elementary objects has been left somewhat informal, indicated by the use of italic type. Some of these predicates and elementary objects are used in the Machine-state.

The degree of abstraction between concrete and abstracted programs is, to a certain extent, a matter of the definer's choice. In this definition, the aim has been to define an abstraction that leaves only those parts of an ASPLE program that are essential for execution.

Abstracted ASPLE programs are simpler than the corresponding concrete programs. There are no declarations and the only explicit type information is contained in the abstraction of the input statement. The type information is needed there to check that the types of the input value and target value match. There is, however, some implicit type information contained in the representation of operators. For example, either *plus* or *or*, depending on the type of the operands, are used to represent the + operator of the original program. This is a similar situation to that which exists in compiled machine code for real computers where there is implicit type information contained in the choice of the operation codes.

```
[A01]  is-abs-program        =  is-abs-statement-list

[A02]  is-abs-statement      =  is-abs-assignment v is-abs-conditional v is-abs-loop v is-abs-input
                                                                                   v is-abs-output

[A03]  is-abs-assignment     =  (<target: is-abs-identifier>, <source: is-abs-expr>)

[A04]  is-abs-conditional    =  (<condition: is-abs-expr>, <true-part: is-abs-statement-list>,
                                                  <false-part: is-abs-statement-list>)

                                [the false part may be an empty list]

[A05]  is-abs-loop           =  (<condition: is-abs-expr>, <body: is-abs-statement-list>)

[A06]  is-abs-input          =  (<target: is-abs-loc>, <mode: is-abs-mode>)

                                [the mode part gives the mode of the target so that the
                                 mode of the input value can be checked]

[A07]  is-abs-output         =  (<source: is-abs-expr>)

[A08]  is-abs-loc            =  (<name: is-abs-identifier>, <deref: is-abs-integer>)

                                [a location is an identifier to which the deref operation
                                 is to be applied a specified number of times]

[A09]  is-abs-expr           =  is-abs-value v is-abs-infix-op v is-abs-loc

[A10]  is-abs-value          =  is-abs-const v is-abs-identifier

[A11]  is-abs-const          =  is-abs-boolean v is-abs-integer

[A12]  is-abs-infix-op       =  (<operand-1: is-abs-expr>, <operand-2: is-abs-expr>,
                                                            <action: is-abs-operator>)

[A13]  is-abs-mode           =  [a set of two elementary objects represented by {int, bool}]

[A14]  is-abs-identifier     =  [an infinite set of distinguishable elementary objects]

[A15]  is-abs-integer        =  [an infinite set of elementary objects denoting the integer values.
                                The subset that denotes the integer values less than 10 is
                                represented by {0, 1, ..., 9}]

[A16]  is-abs-boolean        =  [a set of two elementary objects denoting truth values and
                                represented by {true, false}]

[A17]  is-abs-operator       =  [a set of elementary objects represented by {plus, mult, or, and,
                                                                     equal, notequal}]

[The sets of elementary objects satisfying the predicates is-abs-identifier, is-abs-integer,
 is-abs-boolean, is-abs-mode, and is-abs-operator are mutually exclusive.]
```

TABLE 4.3. DEFINITION OF THE PREDICATE IS-ABS-PROGRAM

VDL Translator

The construction of an abstracted program from its corresponding concrete program is defined by an algorithm, the Translator. This algorithm checks that the concrete program satisfies the context-sensitive requirements of ASPLE and, if so, constructs the corresponding abstracted program. The Translator is defined by the set of functions, many of them recursive, specified in Table 4.4. To explain the notation of this table, we describe the working of some of these functions.

Generally, the functions consist of conditional expressions of the form

$$p_1 \rightarrow e_1\,,\quad p_2 \rightarrow e_2\,, \cdots, p_n \rightarrow e_n$$

where p_i is a predicate expression and e_i is an expression defining an action to be taken. The value of this conditional is the value of the first evaluated expression e_i for which p_i is true. In this definition, the conditional expressions are all written so that at least one predicate is true.

The top-level function **translate** is defined in **[T01]**:

[T01] **translate(t)** =
  program-length(t) $\leq n_1 \rightarrow$ **trans-program(t)**
  true $\rightarrow$ **error** [*program too long*]

This function has a single parameter, **t**, corresponding to an object satisfying the predicate **is-c-program**. The function is evaluated in the process of initializing the Machine-state with PROG, the VDL representation of the source program.

The function **translate** checks that the length of the program, calculated by **program-length**, is less than the implementation defined limit, n_1. If this condition is satisfied, the abstracted program obtained by evaluating **trans-program(t)** is returned as the value of **translate**. This abstracted program will be attached as part of the initial Machine-state ξ_0 by the selector **program** (see Table 4.1). If the length of the program is too great, the Translator terminates in an error. This is typical of the checks that the Translator makes. If the program being translated fails a test, the process is stopped and the program is left undefined.

The function **trans-program [T02]**:

[T02] **trans-program(t)** =
  number-of-identifiers(s_2(**t**)) $< n_2$ [*n_2 is an implementation defined maximum*]
    $\rightarrow$ **trans-stm-train**(s_4(**t**))
  true $\rightarrow$ **error** [*too many variables declared*]
  [*where*: **is-c-dcl-train**(s_2(**t**)) *and* **is-c-stm-train**(s_4(**t**))]

first checks that the number of variables declared is less than the implementation defined maximum, n_2. The number of variables declared is obtained by evaluating **number-of-identifiers** with the argument s_2(**t**). This function also checks that no identifier is declared more than once. The parameter **t** of **trans-program** is PROG that was passed on by **translate**. Applying the selector s_2 using rule **[C01]** selects the object representing the declare train. While counting the number of variables declared, **number-of-identifiers** also checks that no identifier is declared more than once. If there are not too many declared identifiers, **trans-stm-train** is evaluated with the s_4(**t**), which is the statement train.

The translation of the statement train is specified in statement **[T03]**:

[T03]
trans-stm-train(t) =
  slength(t) $= 0 \rightarrow \langle\rangle$ [*if the statement train contains no statement, return an empty list; this can arise when translating the else part of a conditional*]
  true $\rightarrow \mu_0$ ({$\langle$**elem(i)**: **trans-stmt**(s_i(**t**)) $\rangle \;\|\; 1 \leq$ **i** $\leq$ **slength(t)**})
  [*where*: **is-c-statement**(s_i(**t**)), $1 \leq$ **i** $\leq$ **slength(t)**)]

```
[T01] translate(t)=
        program-length(t) ≤ n_1 → trans-program(t)
        true                    → error [program too long]

[T02] trans-program(t)=
        number-of-identifiers(s_2(t)) < n_2    [n_2 is an implementation defined maximum]
                                              → trans-stm-train(s_4(t))
        true                                  → error [too many variables declared]

        [where is-c-dcl-train(s_2(t)) and is-c-stm-train(s_4(t))]

[T03] trans-stm-train(t)=
        slength(t) = 0  → <> [if the statement train contains no statement, return an
                               empty list, this can arise when translating the else
                               part of a conditional]
        true            → μ_0({<elem(i)  trans-stmt(s_i(t))> || 1 ≤ i ≤ slength(t)})
        [where  is-c-statement(s_i(t)), 1 ≤ i ≤ slength(t))]

[T04] trans-stmt(t)=
        is-c-asgt-stm(t)    → trans-asgt-stm(t)
        is-c-cond-stm(t)    → trans-cond-stm(t)
        is-c-loop-stm(t)    → trans-loop-stm(t)
        is-c-input-stm(t)   → trans-input-stm(t)
        is-c-output-stm(t)  → trans-output-stm(t)

[T05] trans-asgt-stm(t)=
        valid-mode-for-assignment(t)  → translate-assignment(t)
        true                          → error [modes not compatible for assignment]

[T06] translate-assignment(t)= [if the reference chain length of the target is 1 then
                                the righthand side is treated as an expression, other-
                                wise the right side is a reference and the appropriate
                                amount of de-referencing must be calculated]
        ref-chain-length(s_1(t)) = 1  → μ_0(<target: make-id(s_1(t))>,
                                                <source: trans-exp(s_3(t))>)
        true                          → μ_0(<target: make-id(s_1(t))>,
                                                <source: trans-ref(s_3(t),
                                                 ref-chain-length(s_1(t))-1>)
        [where  is-c-id(s_1(t))and is-c-exp(s_3(t))]

[T07] trans-cond-stm(t)=
        primitive-mode(s_2(t)) = bool  →
          μ_0(<condition. trans-expr(s_2(t))>, <true-part: trans-stmt-train(s_4(t))>,
                                              <false-part. trans-stmt-train(s_2•s_5(t))>)
        true                           → error [mode of conditional expression not
                                                 boolean]
        [where. is-c-exp(s_2(t)), is-c-stm-train(s_4(t)), and is-c-stm-train(s_2•s_5(t))]

[T08] trans-loop-stm(t)=
        primitive-mode(s_2(t)) = bool  → μ_0(<condition  trans-expr(s_2(t))>,
                                                <body: trans-stm-train(s_4(t))>)
        true                           → error  [mode of conditional expression not
                                                  boolean]
        [where  is-c-exp(s_2(t)), and is-c-stm-train(s_4(t))]

[T09] trans-input-stm(t)=
        μ_0(<target. trans-ref(s_2(t), 1)>,  <mode: primitive-mode(s_2(t))>)
        [where  is-c-id(s_2(t))]
```

TABLE 4.4. ASPLE TRANSLATOR

```
[T10] trans-output-stm(t)=
        valid-exp(t)     →  μ0(<source: trans-exp(s2(t))>)
        true             →  error  [invalid expression]
        [where  is-c-exp(s2(t))]

[T11]  trans-exp(t)=
        is-c-bool-const(t)          →  make-bool-const(t)
        is-c-int-const(t)           →  make-int-const(t)
        is-c-id(t)                  →  trans-ref(t,0)    [dereference sufficiently to get value]
        is-c-parenthesized-exp      →  trans-exp(s2(t))  [t is a parenthesized expression]
        true                        →  μ0(<operand-1: trans-exp(s1(t))>, <operand-2: trans-exp(s3(t))>,
                                                    <action: make-operator(t)>)
                                        [if t is not a constant, identifier, or parenthesized
                                         expression then t consists of two operands and an operator]

[T12]  trans-ref(t,n)=   [construct a reference to a variable such that the length of the
                           reference chain of the value is n]
        μ0(<name: make-id(t)>, <deref: ref-chain-length(t)-n>

[T13]  make-id(t)=
        slength(t) < n4   [implementation defined maximum]
                        →[an elementary object satisfying is-identifier such that
                           (∀t1,t2) (is-c-id(t1) & is-c-id(t2) & (make-id(t1) =
                           make-id(t2) ⊃ t1= t2) ) that is, there is a one-to-one mapping
                           between t and the result of this operation]
        true            →error  [identifier longer than implementation defined length]

[T14]  make-bool-const(t)=
        is-true(t)     →   true
        is-false(t)    →   false    [there can be no other possibility]

[T15]  make-int-const(t)=
        value-of-int-const(t) ≤ n5   → value-of-int-const(t)
        true                         → error  [integer constant too big for
                                                implementation]

[T16]  value-of-int-constant(t)=
        is-0(t)    →   0
        is-1(t)    →   1
        ⋮
        is-9(t)    →   9
                           slength(t)
        slength(t) < n3  →    Σ   value-of-int-const(si(t)) • 10 ↑ (slength(t)-i)
                             i=1
        true             →   [too many digits in integer constant]

        [where  is-c-digit(si(t)), 1 ≤ i ≤ slength(t)]

[T17]  make-operator(t)=
        primitive-mode(s1(t)) = bool  &  is-+(s2(t))   →   or
        primitive-mode(s1(t)) = bool  &  is-*(s2(t))   →   and
        primitive-mode(s1(t)) = int   &  is-+(s2(t))   →   plus
        primitive-mode(s1(t)) = int   &  is-*(s2(t))   →   mult
        primitive-mode(s1(t)) = int   &  is-=(s2(t))   →   equal
        primitive-mode(s1(t)) = int   &  is-≠(s2(t))   →   notequal
        [where  is-c-exp(s1(t))]
```

TABLE 4.4—*Continued*

```
[T18] primitive-mode(t)=   [check validity of expression and obtain its primitive mode]
        is-c-id(t)                          →  primitive-mode-of-id(t)
        is-c-bool-const                     →  bool
        is-c-int-const                      →  int
        is-c-parenthesized-expression(t)    →  primitive-mode(s2(t))
        valid-compare(t)                    →  bool
        valid-exp(t)                        →  primitive-mode(s1(t))
                                               [primitive mode of valid expression is
                                                primitive mode of either operand]
        true                                →  error [invalid expression]

[T19] ref-chain-length(t)=
        is-c-id(t)   →  slength(s1•mode-of-id(t))+1
                        [this is an elementary object satisfying is-integer]
        true         →  1
        [where   s1•mode-of-id(t) is the list of ref's in the declaration of the
                 identifier t]

[T20] primitive-mode-of-id(t)=
        is-bool(s2(mode-of-id(t)))    →  bool
        is-int(s2(mode-of-id(t)))     →  int

[T21] mode-of-id(t)= [find declaration that contains identifier equal to t and select
                      mode part of declaration]
        (∃x1) (x1•s2(PROG)=t)  →  s1•((ιx2) (is-c-declaration(x2(PROG)) &
                                                   (∃i)(s1•s2•x2(PROG)=t)))(PROG)
        true                   →  error [identifier was not declared]
        [where  is-c-dcl-train(s2(PROG))]

[T22] program-length(t)=
        ¬is-slist(t)   →  1
                          slength(t)
         true          →     Σ     program-length(s1(t))
                            i=1

[T23] number-of-identifiers(t)=
                                     slength(t)
        valid-declare-train(t)   →      Σ     slength(s2•s1(t))
                                       i=1
        true                     →   error  [duplicate declarations in declare train]
        [where   is-c-idlist(s2•s1(t)), 1 ≤ i ≤ slength(t)]

[T24] valid-declare-tr`in(t)=
        ¬(∃x1,x2) (x1≠x2  &  is-c-id(x1(t))  &  is-c-id(x2(t))  &  x1(t) = x2(t))
        [this is only true of the declare train t if there do not exist two different
         selectors that select equal identifiers, i.e., if there are no duplicate
         declarations]

[T25] valid-mode-for-assignment(t)=
        (primitive-mode(s1(t)) = primitive-mode(s3(t))  &
                              (ref-chain-length(s1(t))-1 ≤ ref-chain-length(s3(t)))
        [true if the mode of the right side of an assignment statement is valid for
         assignment to the left side]
        [where  is-c-id(s1(t)) and  is-c-exp(s3(t))]

[T26] valid-compare(t)=
        is-c-compare(t)  & (primitive-mode(s1(t)) = int) & (primitive-mode(s3(t)) = int)
        [where  is-c-exp(s1(t))  &  is-c-exp(s3(t))]

[T27] valid-exp(t)=
        ¬is-=(s2(t)) &  ¬is-≠(s2(t)) & (primitive-mode(s1(t)) = primitive-mode(s3(t)))
        [where  is-c-exp(s1(t))  &  is-c-exp(s2(t))]
```

TABLE 4.4.—*Continued*

In this function, if the parameter is the null object, then an empty list is returned. This could happen while translating the conditional statement if the else-part is empty. Otherwise, **trans-stm-train** returns a list constructed by applying the μ_0 operation to the result of evaluating **trans-stmt** for each statement of the train.

The Translator often uses predicate functions for making context-sensitive checks. For example, the predicate **valid-mode-for-assignment**, defined in statement [T25]:

[T25]
valid-mode-for-assignment(t) =
  (**primitive-mode**(s_1(**t**)) = **primitive-mode**(s_3(**t**)) &
    (**ref-chain-length**(s_1(**t**)) − 1 ≤ **ref-chain-length**(s_3(**t**)))
  [*true if the mode of the right side of an assignment statement is valid for assignment to the left side*]
  [*where*: **is-c-id**(s_1(**t**)) *and* **is-c-exp**(s_3(**t**))]

is true if the modes of the expression and the target of the assignment statement **t** are compatible. This predicate specifies the rule needed for legal modes in assignment; that is, the primitive modes of both sides must be identical, and the number of levels of indirection of the source and target must be compatible. The functions **primitive-mode** and **ref-chain-length** are defined in statements [T18] and [T19], respectively.

The function **mode-of-id** [T21]:

[T21]
mode-of-id(t) = [*find declaration that contains identifier equal to* **t** *and select mode part of declaration*]
  $(\exists\chi_1)(\chi_1\cdot s_2(\mathrm{PROG}) = \mathbf{t}) \rightarrow s_1\cdot((\iota\chi_2)(\textbf{is-c-declaration}(\chi_2(\mathrm{PROG}))\ \&$
  $(\exists i)(s_i\cdot s_2\cdot\chi_2(\mathrm{PROG}) = \mathbf{t})))(\mathrm{PROG})$
  true → **error** [*identifier was not declared*]
  [*where*: **is-c-dcl-train**(s_2(PROG))]

checks that there exists a declaration for the identifier **t** and, if so, selects the mode part of the declaration of **t**. The existence of the declaration is verified by the predicate

$$(\exists\chi_1)(\chi_1\cdot s_2(\mathrm{PROG}) = \mathbf{t})$$

which is true if and only if there exists a composite selector χ_1 which, when applied to s_2(PROG), yields the identifier **t**. The object s_2(PROG) is the declare train from the concrete program; see rule [C01]. If the selector χ_1 exists, then there must be an occurrence of the identifier **t** in the declare train, and **t** must have been declared. If χ_1 does not exist, then **t** has not been declared and the program is in error.

If there is a declaration of **t**, then the value of **mode-of-id(t)** is

$$s_1\cdot((\iota\chi_2)\ (\textbf{is-c-declaration}(\chi_2(\mathbf{PROG}))\ \&\ (\exists i)(s_i\cdot s_2\cdot\chi_2(\mathbf{PROG}) = \mathbf{t})))(\mathbf{PROG})$$

The iota function, ι, applied here yields the composite selector χ_2, which satisfies two conditions. The object χ_2(PROG) must be a declaration and there must exist an i such that $s_i\cdot s_2\cdot\chi_2$(PROG) is equal to **t**. If χ_2(PROG) is a declaration, then $s_2\cdot\chi_2$(PROG) is the list of identifiers being declared; see rule [C04]. Applying s_i to this list of identifiers yields an identifier. This condition requires that χ_2 select the declaration that contains **t** in the identifier list. We know that χ_2 must be unique; otherwise, the function declare train [T23] would have detected an error. The iota function thus yields the unique composite selector χ_2 such that χ_2(PROG) is the declaration of **t**. Applying the selector s_1 to this declaration yields the mode of **t**.

The translation process thus consists of executing a sequence of operations that pass back a value to the caller. The final result, provided all the validity checks are passed, is the translated program. This is attached as a component of the initial Machine-state.

```
[I01]  interpret-program  = interpret-statement-list(program(ξ))

[I02]  interpret-statement-list(t)  =

          is-<>(t)  →  Ω    [if t is empty list, do nothing]
          true      →  interpret-statement-list(tail(t));  [defines interpretation sequence
                          interpret-statement(head(t))        of statements in program]

[I03]  interpret-statement(t)=

          is-abs-assignment(t)      →  interpret-assignment(t)
          is-abs-conditional(t)     →  interpret-conditional(t)
          is-abs-loop(t)            →  interpret-loop(t)
          is-abs-input(t)           →  interpret-input(t)
          is-abs-output(t)          →  interpret-output(t)

[I04]  interpret-assignment(t)=
          assign(target(t), value), [evaluate the right side then pass value to assign
                                     operation]
               value    eval-exp (source(t))
          [where   is-abs-identifier(target(t)) and is-abs-exp(source(t))]

[I05]  interpret-conditional(t)=
          eval-exp(condition(t)) = true   →  interpret-statement-list(true-part(t))
          true                            →  interpret-statement-list(false-part(t))

          [where   is-abs-statement-list(true-part(t)), is-abs-statement-list(false-part(t)),
                   and is-abs- expr(condition(t))]

[I06]  interpret-loop(t)=
         eval-exp(condition(t)) = true       →   interpret-loop(t),
                                                   interpret-statement-list(body(t))
         true                                →   Ω
         [where   is-abs-expr(condition(t)) and is-abs-statement-list(body(t))

[I07]  interpret-input(t)=
         assign(destination, value),
            destination   eval-ref(name•target(t), deref•target(t));
            value         read(mode(t))
         [where   is-abs-loc(target(t)), is-abs-mode(mode(t))]

[I08]  interpret-output(t)=
         write(value),
            value   eval-exp(source(t))

[I09]  eval-exp(t)=
         is-abs-loc(t)              →    eval-ref(name(t), deref(t))
         is-abs-infix-op(t)         →    operate(value1, value2, action(t));
                                                value1· eval-exp(operand-1(t)),
                                                value2· eval-exp(operand-2(t))
         is-abs-value(t)            →    PASS  t

[I10]  operate(v1, v2, op)=
         op =  plus          →  add(v1, v2)
         op =  mult          →  multiply(v1, v2)
         op =  or            →  logical-or(v1, v2)
         op =  and           →  logical-and(v1, v2)
         op =  equal         →  compare-equal(v1, v2)
         op =  notequal      →  compare-notequal(v1, v2)
```

TABLE 4.5. DEFINITION OF THE ASPLE INTERPRETER

```
[I11]  add(a,b)=
         a + b < n5[an implementation-defined maximum]   →  PASS  a + b
         true                                            →  PASS. implementation-defined result

[I12]  multiply(a,b)=
         a   b < n5[an implementation-defined maximum]   →  PASS  a   b
         true                                            →  PASS: implementation-defined result

[I13]  logical-or(a,b)=
         a =  true                →    PASS· true
         true                     →    PASS  b [if a is false, the value is the value of b]

[I14]  logical-and(a,b)
         a =  false               →    PASS  false
         true                     →    PASS  b [if a is true, the value is the value of b]

[I15]  compare-equal(a,b)=
         a = b                    →    PASS  true
         a ≠ b                    →    PASS. false

[I16]  compare-not-equal(a,b)=
         a = b                    →    PASS  false
         a ≠ b                    →    PASS  true

[I17]  assign(target, value)=   [perform the actual assignment of a value to storage]
           =  μ(store(ξ):  <target  value>)

[I18]  eval-ref(id, n)=
         n = 0                    →    PASS· id
         true                     →    eval-ref(ref, n-1),
                                           ref  dereference(id)

[I19]  dereference(id)=
         ¬is-Ω(id•store(ξ))       →    PASS. id•store(ξ) [obtain value of variable id from store]
         true                     →    error  [reference to value that has not been set]

[I20]  write(v)=
         length(output(ξ)) < n6     [an implementation-defined maximum]
                                        →  μ(ξ'<output  output(ξ) v>) [concatenate value v
                                           on end of output file]
         true                           →  error  [number of items on output file greater than
                                                    implementation defined maximum]

[I21]  read(t)=  [read and check value from input file
         is-<>(input(ξ))                     →    error  [end of file]
         mode-of-const(head(input(ξ))=t      →    μ(ξ:< input. tail(input(ξ))>)
                                                  PASS· head(input(ξ))
         true                                →    error  [mode of input incompatible]

[I22]  mode-of-const(v)=  [obtain mode of value in the input file]
         is-abs-boolean(v)        →    PASS. bool
         is-abs-integer(v)        →    PASS· int
```

TABLE 4.5.—*Continued*

For example, the tree representation of ξ_0, corresponding to the ASPLE program:

begin
int A;
input A;
$A := A + 5$;
output A
end

and an input file containing the value 7 is shown in Figure 8.

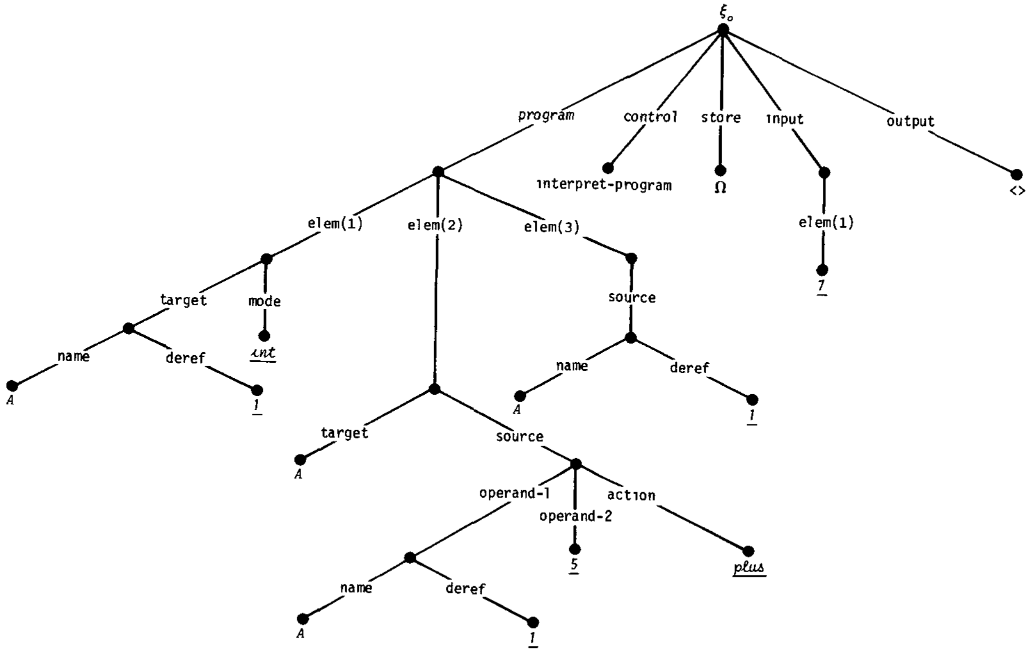

Figure 8. Initial machine-state.

VDL Interpreter

In the previous subsections we described the construction of the abstracted program and its attachment as part of the initial Machine-state of the abstract machine. The control part of ξ contains the operation **interpret-program**. Execution of this operation begins the interpretation of the abstracted program. Interpretation will continue until the control part becomes empty or until an error is detected.

The control part of ξ is a composite object with operations at the nodes. Since ASPLE is a language without side effects, we can simplify this and treat the control part informally as a stack of machine operations, some with arguments. The operation most recently added to the stack is the next one to be executed. When execution is complete, the operation is removed from the stack. The execution of an operation causes one of the following:

- the addition of new operations to the instruction stack;
- the insertion of a value into the argument list of an operation already on the stack, possibly accompanied by a change to some other components of ξ.

The machine operations of the ASPLE Machine are defined in Table 4.5. The **interpret-program** operation is defined in statement [101]:

[101] **interpret-program** = **interpret-statement-list(program**(ξ))

Its effect is to cause **interpret-statement-list** to be put on the operation stack with the abstracted program from ξ as argument. The abstracted program, defined in Table 4.3, consists of a statement list.

The operation **interpret-statement-list** is defined in [102]:

```
[102]        interpret-statement-list(t) =
               is-<>(t) → Ω [if t is empty list, do nothing]
               true     → interpret-statement-list(tail(t));
                            interpret-statement(head(t))
                            [defines interpretation sequence of statements in program]
```

and uses the same type of conditional expression as is used in the Translator. Here the expression to the right of the arrow specifies the operation that is to be added to the stack. If the statement list **t** is empty, then nothing, signified by Ω, is added to the operation stack. This is the way that the control part of ξ will become empty at the end of the interpretation. If the statement list **t** is not empty, the pair of operations:

interpret-statement-list (tail (t));
interpret-statement (head (t))

replaces the current operation on the stack in the order shown. The semicolon after **interpret-statement-list** separates the two operations. The **interpret-statement** operation is thus executed next. The argument of this operation is the first statement from the statement-list **t**, making this statement the next ASPLE statement to be interpreted. When this is completed, **interpret-statement-list** will become the next operation in the stack to be executed. Its argument is the statement list **t** with its first element deleted. This mechanism defines the sequence of execution of the statements of the ASPLE program.

As an example of the way statements are interpreted, consider **interpret-assignment**, defined in [104]:

```
[104]          interpret-assignment(t) =
          assign(target(t), value); [evaluate the right side then pass value to assign operation]
            value:  eval-exp (source(t))
              [where: is-abs-identifier(target(t)) and is-abs-exp(source(t))]
```

Execution of **interpret-assignment** causes it to be replaced on the stack by:

assign (target (t), value);
value: eval-exp (source (t))

The term "**value**·" denotes that the execution of **eval-exp** will return a value, which will be known locally as **value**. This value will be substituted into the argument list of an, as yet, unexecuted instruction on the operation stack. The value replaces the argument denoted by **value** in **assign**. In this way, the value computed by **eval-exp** is passed to **assign** for assignment to storage.

The definition of **eval-exp**:

```
[109]            eval-exp(t) =
                   is-abs-loc(t)       → eval-ref(name(t), deref(t))
                   is-abs-infix-op(t)  → operate(value1, value2, action(t));
                                           value1: eval-exp(operand-1(t));
                                           value2: eval-exp(operand-2(t))
                   is-abs-value(t)     → PASS: t
```

shows how the return of a value is expressed. This operation has a three-way conditional, and the action depends on the kind of expression passed to the operation as an argument. If the expression **t** is a reference, that is, if **is-abs-loc(t)** is true, then a single new operation, **eval-ref**, is put on the stack. If the expression is an infix operation, then three new operations are added to the stack. However, if it is a value, that is, if it corresponds to a constant in the original program, the actual value of the constant is returned. This is signified by **PASS**:, followed by the value to be returned.

Value-returning operations can also make changes to other parts of ξ through the use of the μ operator. For example, **read**, defined in statement [I21]:

```
[I21]   read(t) = [read and check value from input file]
            is-<>(input(ξ))                    → error [end of file]
            mode-of-const(head(input(ξ)) = t   → μ(ξ  :  <input: tail(input(ξ))>)
                                                 PASS: head(input(ξ))
            true                               → error
                                                   [mode of input incompatible]
```

first checks that the end of file has not been reached. If it has, this is an error and interpretation stops at this point. The next check is that the mode of the value to be read is the same as that of the variable to which it is to be assigned. This latter mode was determined by the Translator and inserted into the abstracted program. If the modes are compatible, two things take place simultaneously: the μ operator replaces the input component of ξ by its tail, and the head of the input component ξ is returned with the **PASS**: mechanism.

Discussion

The VDL definition of ASPLE specifically indicates the points at which errors can be detected in programs that are in accordance with the context-free syntax. These points are marked explicitly in the Translator and Interpreter algorithms. For example, if the modes of two operands in an expression are not compatible, then an error will be detected by **primitive-mode** [T18]. If a reference is made to a variable that has not been assigned a value, the error will be detected by **dereference** [I19].

By making a distinction between the Translator and the Interpreter, this VDL definition shows the difference between the static and dynamic aspects of ASPLE. While some errors can be detected statically, others seem to require interpretation. The dividing line between the two is a matter of judgment by the writer of the definition. Here, we have left to the Interpreter the detection of any error that required the manipulation of some data. In the VDL technique of definition, errors will only be detected by the Interpreter if the part of the program containing the error is actually executed.

This technique of language definition continues to be developed. Later work aimed at proving the correctness of implementations has shown a need to make the definitions even more abstract. These developments are described by Békic et al. [B0].

5. ATTRIBUTE GRAMMARS

We now discuss the definition technique of attribute grammars originally due to Knuth [K1]. Attribute grammars and related concepts have been described in different places [B1, B2, K1, K2, L5]. The notation used here is closely related to that used in [B1, B2, L5].

Overview

A context-free grammar of a language defines a derivation tree for each syntactically correct program of the language. An attribute grammar is based on a context-free grammar and associates attributes with the nodes of a derivation tree. Attribute evaluation rules are associated with the context-free productions of an attribute. The evaluation rule associated with a given production is applied for all instances of this production in the derivation tree.

Attributes can be of two kinds: the *inherited* attributes, whose values are obtained from the immediate parent node and its production in the derivation tree, and the *synthesized* attributes, whose values are obtained from the immediate descendants in the tree and the productions generating these. The inherited attributes of the left side of a production and the synthesized attributes of the right side represent values *obtained* from the surrounding nodes in the derivation tree. The evaluation rules of a production specify the computation of the other attributes, that is, the inherited attributes of the right side and the synthesized attributes of the left. The values of these attributes are passed to the surrounding nodes. More generally, one can say that the synthesized attributes of a node represent information which is synthesized in the subtree of the node and *passed* up toward the root node of the derivation tree, whereas the inherited attributes represent information which is passed down, from the root nodes towards the leaves. Inherited attributes indicate the context in which the node and its subtree are found.

The context-sensitive constraints of a language are expressed by conditions included in its attribute grammar. These conditions specify relations between the attribute values that must be satisfied in the derivation tree of a valid program.

Different methods can be used for specifying the attribute evaluation rules. The concept of attribute grammars is not a complete method for making formal definitions of programming languages. For general use, it must be combined with a method for the specification of its evaluation rules. In the attribute grammar for ASPLE, we use *action symbols* [L5] to specify evaluation rules other than simple value transfers.

Several approaches can be used with attribute grammars for the specification of the semantics of a program. Knuth [K1] proposed that the "meaning" of a program be given by the value of a special attribute at the root node of the derivation tree. For the specification of the ASPLE semantics, we have chosen a different approach, which corresponds to the practice of implementing programs in two phases: translation into a lower level target language, followed by the execution of the translated program. Therefore we distinguish two kinds of action symbols: 1) those symbols that are executed during a translation phase and that evaluate attribute values in the derivation tree, and 2) those that are executed later during an execution phase [B2]. The meaning of a program is specified by the sequence of action symbols and certain attribute values obtained during the translation of the source text of the program. Rather than choosing a rigidly defined set of actions for the execution phase (for example, a particular machine language) we have, as is customary, left the meaning of the action symbols informally defined.

Attribute Grammar for ASPLE

An attribute grammar for ASPLE is shown in Table 5.1. The production for the starting symbol **<program>** is shown in [AG01].

[AG01]
<program> ↑**memory** ::= *begin*
  <dcl train> ↓ *empty-env* ↓ *zero-ids* ↓ *empty-memory*
    ↑ **env** ↑ **num-ids** ↑ **memory**
  ;
  <stm train> ↓ **env**
  end
  condition: num-ids $< n_2$
    [*number of declared identifiers must be less than the implementation defined number*]
  condition: prog-length $< n_1$
    [*prog-length is an implementation defined attribute whose evaluation rules must be added to the grammar*]

```
[AG01]  <program> ↑memory   = begin
                              <dcl train>  ↓empty-env  ↓zero-ids  ↓empty-memory
                                           ↑env  ↑num-ids  ↑memory
                              ;
                              <stm train>  ↓env
                              end
                              condition  num-ids < n2
                                [number of declared identifiers must be less than
                                 the implementation defined number]
                              condition  prog-length < n1
                                [prog-length is an implementation defined attribute
                                 whose evaluation rules must be added to the grammar]

[AG02]  <dcl train>  ↓env1  ↓num-ids1  ↓memory1  ↑env2  ↑num-ids2  ↑memory2
                            = <declaration>  ↓env1  ↓num-ids1  ↓memory1  ↑env2
                                             ↑num-ids2  ↑memory2
                            | <declaration>  ↓env1  ↓num-ids1  ↓memory1  ↑env3
                                             ↑num-ids3  ↑memory3
                              ;
                              <dcl train>    ↓env3  ↓num-ids3  ↓memory3  ↑env2
                                             ↑num-ids2  ↑memory2

[AG03]  <stm train>  ↓env   = <statement>  ↓env
                            | <statement>  ↓env
                              ;
                              <stm train>  ↓env

[AG04]  <declaration>  ↓env1  ↓num-ids1  ↓memory1  ↑env2  ↑num-ids2  ↑memory2
                            = <mode>  ↑prim-mode  ↑refs
                              <id-list>  ↓env1  ↓num-ids1  ↓prim-mode  ↓refs
                                         ↓memory1  ↑env2  ↑num-ids2  ↑memory2

[AG05]  <mode>  ↑prim-mode  ↑refs1
                            = bool
                              give value to attribute ↓bool  ↑prim-mode
                              give value to attribute ↓one-ref  ↑refs1
                            | int
                              give value to attribute ↓int  ↑prim-mode
                              give value to attribute ↓one-ref  ↑refs1
                            | ref
                              <mode>  ↑prim-mode  ↑refs2
                              add one ref  ↓refs2  ↑refs1

[AG06]  <id-list>  ↓env1  ↓num-ids1  ↓prim-mode  ↓refs  ↓memory1  ↑env2  ↑num-ids2  ↑memory2
                            = <declared id>  ↓env1  ↓prim-mode  ↓refs  ↓memory1
                                             ↑env2  ↑memory2
                              add one id  ↓num-ids1  ↑num-ids2
                            | <declared id>  ↓env1  ↓prim-mode  ↓refs  ↓memory1
                                             ↑env3  ↑memory3
                              add one id  ↓num-ids1  ↑num-ids3
                              ;
                              <id-list>  ↓env3  ↓num-ids3  ↓prim-mode  ↓refs  ↓memory3
                                         ↑env2  ↑num-ids2  ↑memory2
```

TABLE 5.1. ATTRIBUTE GRAMMAR OF ASPLE

```
[AG07]  <declared id>  ↓env1   ↓prim-mode1  ↓refs1  ↓memory1  ↑env2  ↑memory2
                       =  <id>  ↑name1
                            insert declaration  ↓env1  ↓name1  ↓prim-mode1  ↓refs1  ↑env2
                            include variable  ↓memory1  ↓name1  ↑memory2
                              [the name is added to memory and its value initialised to
                                                                              undefined]
                            condition   ¬(∃t)(t=(name2,prim-mode2, refs2)   &
                                               t ∈ env1  &  name2=name1)
                              [duplicate declarations are not allowed]

[AG08]  <statement>  ↓env      =  <asgt stm>  ↓env
                               |  <cond stm>  ↓env
                               |  <loop stm>  ↓env
                               |  <input stm> ↓env
                               |  <output stm> ↓env

[AG09]  <asgt stm>  ↓env       =  <used id>  ↓env  ↑prim-mode1  ↑refs1  ↑name
                                  =
                                  subtract one ref  ↓refs1  ↑refs2
                                 <exp>  ↓env  ↓refs2  ↑prim-mode2  ↑VALUE
                                  STORE  ↓name  ↓VALUE
                                  condition   prim-mode1 = prim-mode2
                                    [primitive modes must be compatible for assignment]

[AG10]  <cond stm>  ↓env       =   if
                                  <exp>  ↓env  ↓zero-refs  ↑prim-mode  ↑VALUE
                                   BRANCH ON FALSE  ↓VALUE  ↓label1
                                   then
                                  <stm train>  ↓env
                                   BRANCH  ↓label2
                                   else
                                   LOCATE  ↑label1
                                  <stm train>  ↓env
                                   fi
                                   LOCATE  ↑label2
                                   condition   prim-mode = bool
                               |   if
                                  <exp>  ↓env   ↓zero-refs  ↑prim-mode  ↑VALUE
                                   BRANCH ON FALSE  ↓VALUE  ↓label
                                   then
                                  <stm train>  ↓env
                                   fi
                                   LOCATE  ↑label
                                   condition   prim-mode = bool

[AG11]  <loop stm>  ↓env       =   while
                                   LOCATE  ↑label1
                                  <exp>  ↓env  ↓zero-refs  ↑prim-mode  ↑VALUE
                                   BRANCH ON FALSE  ↓VALUE  ↓label2
                                   do
                                  <stm train>  ↓env
                                   end
                                   BRANCH  ↓label1
                                   LOCATE  ↑label2
                                   condition   prim-mode = bool
```

TABLE 5.1.—*Continued*

```
[AG12]  <input stm>  ↓env         =  input
                                     <used id>  ↓env  ↑prim-mode  ↑refs  ↑name1
                                     <deref action>  ↓name1  ↓refs  ↓one-ref  ↑NAME2
                                     <input value>  ↓prim-mode  ↑VALUE
                                     STORE  ↓NAME2  ↓VALUE

[AG13]  <input value>  ↓prim-mode  ↑VALUE
                                  =  READ INTEGRAL  ↑VALUE
                                     condition   prim-mode = int
                                  |  READ BOOLEAN  ↑VALUE
                                     condition   prim-mode = bool
                                       [value input must be compatible with target]

[AG14]  <output stm>  ↓env        =  output
                                     <exp>  ↓env  ↓zero-refs  ↑prim-mode  ↑VALUE
                                     <output action>  ↓prim-mode  ↓VALUE

[AG15]  <output action>  ↓prim-mode  ↓VALUE
                                 .=  WRITE INTEGRAL  ↓VALUE
                                     condition.  prim-mode = int
                                  |  WRITE BOOLEAN  ↓VALUE
                                     condition   prim-mode = bool

[AG16]  <exp>  ↓env  ↓refs  ↑prim-mode1  ↑VALUE1
                                  =  <factor>  ↓env  ↓refs  ↑prim-mode1  ↑VALUE1
                                  |  <exp>  ↓env  ↓zero-refs  ↑prim-mode1  ↑VALUE2
                                     +
                                     <factor>  ↓env  ↓zero-refs  ↑prim-mode2  ↑VALUE3
                                     <+ action>  ↓prim-mode1  ↓VALUE2  ↓VALUE3  ↑VALUE1
                                     condition   prim-mode1 = prim-mode2
                                       [primitive modes must correspond]
                                     condition   refs = zero-refs
                                       [the mode of the factor is without any references]

[AG17]  <+ action>  ↓prim-mode  ↓VALUE1  ↓VALUE2  ↑VALUE3
                                  =  ADD  ↓VALUE1  ↓VALUE2  ↑VALUE3
                                     condition   prim-mode = int
                                  |  OR  ↓VALUE1  ↓VALUE2  ↑VALUE3
                                     condition   prim-mode = bool

[AG18]  <factor>  ↓env  ↓refs  ↑prim-mode1  ↑VALUE1
                                  =  <primary>  ↓env  ↓refs  ↑prim-mode1  ↑VALUE1
                                  |  <factor>  ↓env  ↓ zero-refs ↑prim-mode1  ↑VALUE2
                                     *
                                     <primary>  ↓env  ↓ zero-refs ↑prim-mode2  ↑VALUE3
                                     <* action>  ↓prim-mode1  ↓VALUE2  ↓VALUE3  ↑VALUE1
                                     condition   prim-mode1 = prim-mode2
                                       [primitive modes must correspond]
                                     condition   refs = zero-refs
                                       [the mode of the factor is without any references]
```

TABLE 5.1.—*Continued*

```
[AG19]  <* action>  ↓prim-mode  ↓VALUE_1  ↓VALUE_2  ↑VALUE_3
                        =  MULTIPLY  ↓VALUE_1  ↓VALUE_2  ↑VALUE_3
                           condition  prim-mode = int
                        |  AND  ↓VALUE_1  ↓VALUE_2  ↑VALUE_3
                           condition  prim-mode = bool

[AG20]  <primary>  ↓env  ↓refs_1  ↑prim-mode  ↑VALUE
                        =  <used id>  ↓env  ↑prim-mode  ↑refs_2  ↑name_1
                           <deref action>  ↓name_1  ↓refs_2  ↓refs_1  ↑VALUE
                             [some dereferencing may possibly be done]
                        |  <constant>  ↑prim-mode  ↑VALUE
                           condition  refs_1 = zero-refs
                        |  (
                           <exp>  ↓env  ↓zero-refs ↑prim-mode  ↑VALUE
                           )
                           condition  refs_1 = zero-refs
                        |  (
                           <compare>  ↓env  ↑VALUE
                           )
                           give value to attribute ↓bool  ↑prim-mode
                           condition  refs_1 = zero-refs

[AG21]  <used id>  ↓env  ↑prim-mode  ↑refs  ↑name
                        =  <id>  ↑name
                           condition  (name, prim-mode, refs) ε env

[AG22]  <deref action>  ↓name  ↓refs_1  ↓refs_2  ↑VALUE_1
                        =  give value to attribute  ↓name  ↑VALUE_1
                           condition  refs_1 = refs_2
                             [no dereferencing is necessary]
                        |  LOAD  ↓name  ↑VALUE_2
                             [an undefined stored value gives rise to an error condition]
                             [one level of dereferencing is done]
                           subtract one ref  ↓refs_1  ↑refs_3
                             [refs_1 will always be greater than zero]
                           <deref action>  ↓VALUE_2  ↓refs_3  ↓refs_2  ↑VALUE_1
                           condition  refs_1 > refs_2
                             [several levels of dereferencing can be done recursively
                              The number of times the recursion is invoked depends on
                              the difference of the values of refs_1 and refs_2]

[AG23]  <compare>  ↓env  ↑VALUE_1
                        =  <exp>  ↓env  ↓zero-refs  ↑prim-mode_1  ↑VALUE_2
                           =
                           <exp>  ↓env  ↓zero-refs  ↑prim-mode_2  ↑VALUE_3
                           COMPARE EQUAL  ↓VALUE_2  ↓VALUE_3  ↑VALUE_1
                           condition  prim-mode_1 = int
                           condition  prim-mode_2 = int
                        |  <exp>  ↓env  ↓zero-refs  ↑prim-mode_1  ↑VALUE_2
                           ≠
                           <exp>  ↓env  ↓zero-refs  ↑prim-mode_2  ↑VALUE_3
                           COMPARE NOT EQUAL  ↓VALUE_2  ↓VALUE_3  ↑VALUE_1
                           condition  prim-mode_1 = int
                           condition  prim-mode_2 = int
```

TABLE 5.1.—*Continued*

```
[AG24]  <constant> ↑prim-mode ↑value
                           = <bool constant> ↑value
                             give value to attribute ↓bool ↑prim-mode
                           | <int constant> ↑value
                             give value to attribute ↓int ↑prim-mode

[AG25]  <bool constant> ↑value
                           = true
                             give value to attribute ↓true ↑value
                           | false
                             give value to attribute ↓false ↑value

[AG26]  <int constant> ↑value = <number> ↑num-digits ↑value
                             condition num-digits < n_3
                               [number of digits in an integer constant must be
                                less than the implementation defined maximum n_3]

[AG27]  <number> ↑num-digits_1 ↑value_1
                           = <digit> ↑value_1
                             give value to attribute ↓one-digit ↑num-digits_1
                           | <number> ↑num-digits_2 ↑value_2
                             <digit> ↑value_3
                             multiply ↓value_2 ↓10 ↑value_4
                             add ↓value_4 ↓value_3 ↑value_1
                             add one digit ↓num-digits_2 ↑num-digits_1

[AG28]  <digit> ↑value     = 0
                             give value to attribute ↓0 ↑value
                           | 1
                             give value to attribute ↓1 ↑value

                           | 9
                             give value to attribute ↓9 ↑value

[AG29]  <id> ↑name         = <identifier> ↑num-letters ↑name
                             condition num-letters < n_4
                               [number of letters in an identifier must be less than
                                the implementation defined maximum n_4]

[AG30]  <identifier> ↑num-letters_1 ↑name_1
                           = <letter> ↑name_1
                             give value to attribute ↓one-letter ↑num-letters_1
                           | <identifier> ↑num-letters_2 ↑name_2
                             <letter> ↑name_3
                             concatenate ↓name_2 ↓name_3 ↑name_1
                             add one letter ↓num-letters_2 ↑num-letters_1

[AG31]  <letter> ↑name     = A
                             give value to attribute ↓A ↑name
                           | B
                             give value to attribute ↓B ↑name
                              . .
                           | Z
                             give value to attribute ↓Z ↑name
```

TABLE 5.1.—*Continued*

This production specifies the context-free rule [B01]:

<**program**> ::= *begin* <**dcl train**> ; <**stm train**> *end*

The terminal symbols are written in italic characters. In the attribute grammar, each symbol of the right side of a production starts a new line. Attributes are represented by names which are written on the same line, following the syntactic symbol to which they apply. Synthesized attributes are prefixed by an arrow pointing upward; inherited ones by an arrow pointing downward. The attributes of a given symbol are always written in the same order.

The synthesized attribute ↑ **memory** of the root node <**program**> represents the initial state of the program variables, each one being initialized to the **undefined** value. The value of this attribute is given by the sixth attribute of the <**dcl train**>. Since the latter is a synthesized attribute of a symbol on the right side of the production, its value is obtained from the lower productions in the derivation tree. In this case, the value is synthesized in the subtree of the node <**dcl train**>. The transfer of the attribute value from the right-side symbol <**dcl train**> to the left-side symbol <**program**> is indicated by the use of same name ↑ **memory** at both places. In general, attribute evaluation rules that are simple value transfers are specified by the use of identical names.

The value of the attribute **env** represents the environment of the program and is a set of triples associating identifiers, primitive modes, and reference chain lengths. The value of **env** is synthesized in the subtree of the node <**dcl train**> and is transferred to the inherited attribute of the node <**stm train**>.

The names *empty-env*, *zero-ids*, and *empty-memory* represent constant attribute values, written in script characters. These are the values taken by the first three attributes of the node <**dcl train**> and passed down to its subtree. Table 5.2 lists the set of possible values for each attribute type, the names of constant values used in the attribute grammar, and the action symbols associated with the attribute type.

There are two conditions imposed on the values of the attributes in [AG01]. The value of the attribute **num-ids**, which represents the number of declared identifiers in the program, must be less than the implementation defined constant n_2 and the attribute **prog-length**, which serves to represent the length of the program, must be less than the constant n_1. The attribute **prog-length** is not associated with any symbol, since its computation is left for implementation definition.

The attribute evaluation rules of production [AG01] can be summarized as follows: the values of the attributes **memory** of <**program**> and **env** of <**stm train**> are defined by simple value transfers, indicated by the use of identical names, and the first three attributes of <**dcl train**> are defined to have constant values. The values of all other attributes are determined by surrounding productions within the derivation tree, in this case by the production for <**dcl train**>. In addition, certain conditions must be satisfied by the obtained attribute values. A program that does not satisfy these conditions is invalid.

The production of <**dcl train**> is given in [AG02]:

[AG02] <**dcl train**> ↓ $\mathbf{env}_1$ ↓ $\mathbf{num\text{-}ids}_1$ ↓ $\mathbf{memory}_1$ ↑ $\mathbf{env}_2$ ↑ $\mathbf{num\text{-}ids}_2$ ↑ $\mathbf{memory}_2$
    : := <**declaration**> ↓ $\mathbf{env}_1$ ↓ $\mathbf{num\text{-}ids}_1$ ↓ $\mathbf{memory}_1$ ↑ $\mathbf{env}_2$
        ↑ $\mathbf{num\text{-}ids}_2$ ↑ $\mathbf{memory}_2$
      | <**declaration**> ↓ $\mathbf{env}_1$ ↓ $\mathbf{num\text{-}ids}_1$ ↓ $\mathbf{memory}_1$ ↑ $\mathbf{env}_3$
        ↑ $\mathbf{num\text{-}ids}_3$ ↑ $\mathbf{memory}_3$
      ;
      <**dcl train**> ↓ $\mathbf{env}_3$ ↓ $\mathbf{num\text{-}ids}_3$ ↓ $\mathbf{memory}_3$ ↑ $\mathbf{env}_2$
        ↑ $\mathbf{num\text{-}ids}_2$ ↑ $\mathbf{memory}_2$

In this production, the subscripts on the attribute names distinguish different instances of attributes of the same type. Attributes distinguished in this way can have different values.

[AT01] prim-mode [*primary mode*]
values: *bool* and *int*

[AT02] refs [*length of reference chain*]
values *positive integers*
constants *zero-refs* = 0
one-ref = 1
action symbols <u>add one ref</u> $\downarrow$refs$_1$ $\uparrow$refs$_2$
implies refs$_2$ = refs$_1$ + 1
<u>subtract one ref</u> $\downarrow$refs$_1$ $\uparrow$refs$_2$
implies refs$_2$ = refs$_1$ - 1

[AT03] name [*name of variable*]
values *arbitrary length character strings*
constants A, B, ..., Z
action symbols: <u>concatenate</u> $\downarrow$name$_1$ $\downarrow$name$_2$ $\uparrow$name$_3$
implies name$_3$ *is the concatenation of* name$_1$ *with* name$_2$

[AT04] env [*environment, i.e "symbol table"*]
values: *sets of triples of the form* (name, prim-mode, refs)
constants *empty-env* = ∅
action symbol. <u>insert declaration</u> $\downarrow$env$_1$ $\downarrow$name $\downarrow$prim-mode $\downarrow$refs $\uparrow$env$_2$
implies env$_2$ = env$_1$ ∪ {(name,prim-mode,refs)}

[AT05] value [*boolean, integral, or reference value*]
values: *union of* boolean, integral, *and* name

[AT06] stored-value
values: *union of* value, *and* {*undefined*}

[AT07] boolean
values *true and false*
action symbols. <u>or</u> $\downarrow$boolean$_1$ $\downarrow$boolean$_2$ $\uparrow$boolean$_3$
implies boolean$_3$ = boolean$_1$ ∨ boolean$_2$
<u>and</u> $\downarrow$boolean$_1$ $\downarrow$boolean$_2$ $\uparrow$boolean$_3$
implies boolean$_3$ = boolean$_1$ ∧ boolean$_2$

[AT08] integral
values. *positive integers*
constants: *0, 1, ..., 9, 10*
n_5 = *implementation defined maximum value for integral values*
action symbols. <u>add</u> $\downarrow$*integral*$_1$ $\downarrow$*integral*$_2$ $\uparrow$*integral*$_3$
implies if integral$_1$ + *integral*$_2$ < n_5
then *integral*$_3$ = *integral*$_1$ + *integral*$_2$
otherwise implementation defined result
<u>multiply</u> $\downarrow$*integral*$_1$ $\downarrow$*integral*$_2$ $\uparrow$*integral*$_3$
implies if integral$_1$ × *integral*$_2$ < n_5
then *integral*$_3$ = *integral*$_1$ × *integral*$_2$
otherwise implementation defined result

TABLE 5.2. Definition of Attributes and Action Symbols

[AT08] continued

<u>compare equal</u> ↓integral$_1$ ↓integral$_2$ ↑boolean
implies if integral$_1$ = integral$_2$
then boolean = *true*
otherwise boolean = *false*

<u>compare not equal</u> ↓integral$_1$ ↓integral$_2$ ↑boolean
implies if integral$_1$ = integral$_2$
then boolean = *true*
otherwise boolean = *false*

[AT09] memory *[memory state]*
values *sets of pairs of the form* (name, stored-value)
constant *empty-memory* = ∅
action symbol. <u>include variable</u> ↓memory$_1$ ↓name ↑memory$_2$
implies memory$_1$ = memory$_2$ ∪ {(name, *undefined*)}

[AT10] prog-length *[program length (implementation defined)]*
constant· η_1 = *implementation defined maximum*

[AT11] num-ids *[number of identifiers declared]*
values *positive integers*
constants *zero-ids* = *0*
η_2 = *implementation defined maximum*
action symbol. <u>add one id</u> ↓num-ids$_1$ ↑num-ids$_2$ η-ids$_2$
implies num-ids$_2$ = num-ids$_1$ + 1

[AT12] num-digits *[number of digits in a constant denotation]*
values *positive integers*
constants *one-digit* = *1*
η_3 = *implementation defined maximum*
action symbol <u>add one digit</u> ↓num-digits$_1$ ↑num-digits$_2$
implies num-digits$_2$ = num-digits$_1$ + 1

[AT13] num-letters *[number of letters in an identifier]*
values *positive integers*
constants *one-letter* = *1*
η_4 = *implementation defined maximum*
action symbol <u>add one letter</u> ↓num-letters$_1$ ↑num-letters$_2$
implies num-letters$_1$ = num-letters$_2$ + 1

<u>Additional Data Type for the Execution Phase</u>

The global state space is a product of a memory *state and two* file *states for the input and output.*

[AT14] file *[content of the input or output file]*
values *sequences* $(v_1, v_2, \ldots, v_n)$ *where* $n \geq 0$ *and the* v_i $(1 \leq i \leq n)$ *are of type* integral *or* boolean
constants ( ) = *the empty sequence [file containing only an end of file mark]*

TABLE 5.2.—*Continued*

Additional Actions for the Execution Phase

(a) *Interaction with the global* memory *state*

Execution of LOAD ↓name ↑stored-value$_1$:

if $(\exists t)$ (t=(name, stored-value$_2$)) & t ε *global* memory & stored value$_2$ ≠ *undefined*)

then stored-value$_1$ = stored value$_2$

otherwise execution error [*undefined variable reference*]

execution of STORE ↓name ↓stored-value$_1$

(name, stored-value$_2$) ε *global* memory before

implies *global* memory$_{after}$ = (*global* memory$_{before}$ - {(name, stored-value$_2$)}) ∪ {(name, stored-value$_1$)}

[*the stored value of the variable* name *is replaced by* stored-value$_1$]

(b) *Interaction with global input* file *state*

Execution of READ INTEGRAL ↑value

if input file$_{before}$ = $(v_1, v_2, \ldots, v_n)$ & $n \geq 1$

then if v_1 *is of type* integral

then value = v_1 *and input* file$_{after}$ = $(v_2, \ldots, v_n)$

otherwise execution error [*incompatible input data type*]

otherwise execution error [*attempt to read beyond end of input file*]

Execution of READ BOOLEAN ↑value

if input file$_{before}$ = $(v_1, v_2, \ldots v_n)$ & $n \geq 1$

then if v_1 *is of type* boolean

then value = v_1 *and input* file$_{after}$ = $(v_2, \ldots, v_n)$

otherwise execution error [*incompatible input data type*]

otherwise execution error [*attempt to read beyond end of input file*]

(c) *Interaction with global output* file *state*

Execution of WRITE INTEGRAL ↓value

if output file$_{before}$ = $(v_1, \ldots v_n)$ & $n < \eta_6$

then output file$_{after}$ = $(v_1, \ldots, v_n$, *value*)

otherwise execution error [*implementation defined size of output file exceeded*]

Execution of WRITE BOOLEAN ↓value

if output file$_{before}$ = $(v_1, \ldots, v_n)$ & $n < \eta_6$

then output file$_{after}$ = $(v_1, \ldots, v_n$, *value*)

otherwise execution error [*implementation defined size of output file exceeded*]

(d) *Action symbols for specifying non-sequential execution*

LOCATE ↑*label* [*global state unchanged, locates a unique label to which the branching action symbol can be connected; the next action symbol to be executed is the next one in sequence*]

BRANCH ↓*label* [*global state unchanged, the next action symbol to be executed is the* locate *symbol of the same label*]

BRANCH ON FALSE ↓*value* ↓*label*

[*global state unchanged, if value = false then the next action symbol to be execution is the locate symbol of the same label, otherwise the next action symbol in sequence will be executed*]

TABLE 5.2.—*Continued*

Any order of evaluation of the attributes that leads to well-defined values in the derivation tree is allowed. If we take the second alternative in [AG02], the following sequence of evaluation will be followed for the **env** attribute:

1) the value **env**$_1$ is inherited by **<dcl train>** on the left side of the production;
2) this value is passed down to **<declaration>** and the synthesized attribute value of **env**$_3$ is obtained from the subtree of **<declaration>**;
3) the value of **env**$_3$ is then passed down to **<dcl train>** on the right side of the production, and the value of **env**$_2$ is obtained as a synthesized attribute of the right side, **<dcl train>**;
4) the value of **env**$_2$ is then passed up as a synthesized attribute of **<dcl train>** on the left side of the production.

This process is illustrated in Figure 9, where part of the derivation tree for a declare train with more than one declaration is represented. The solid lines indicate the syntactic structure of the derivation tree, and the broken lines show the transfer of values of the env attribute between the nodes as specified by the productions.

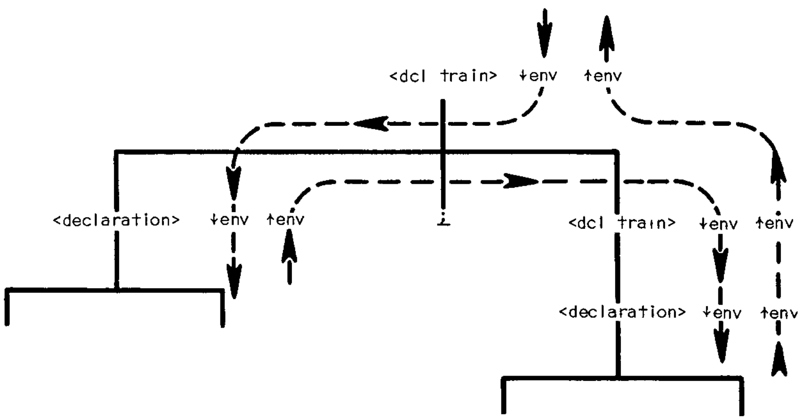

Figure 9. Partial derivation tree for <dcl train> showing evaluation of env attribute.

Action Symbols

In a production where the evaluation of an attribute value requires more than a simple value transfer, action symbols are used. In the attribute grammar, action symbols are always shown with underlined names. The meaning of the actions and their attribute values are defined informally in Table 5.2.

A simple example of the use of action symbols is shown in [AG05]:

[AG05] **<mode> ↑ prim-mode ↑ refs**$_1$

::= *bool*
    give value to attribute ↓*bool* ↑ **prim-mode**
    give value to attribute ↓*one-ref* ↑ **refs**$_1$
| *int*
    give value to attribute ↓*int* ↑ **prim-mode**
    give value to attribute ↓*one-ref* ↑ **refs**$_1$
| *ref*
    <mode> ↑ prim-mode ↑ refs$_2$
    add one ref ↓ **refs**$_2$ ↑ **refs**$_1$

The category <**mode**> has two synthesized attributes: **prim-mode** and **refs**$_1$. In the first two alternatives of [AG05], values are given to these attributes by means of the action symbol, <u>**give value to attribute**</u>, which denotes a function that takes a value, which is in this case a constant, and it returns an attribute with that same value. The attribute **refs** represents the length of the reference chain of a variable. The action symbol:

<u>**give value to attribute**</u> ↓ *one-ref* ↑ **refs**$_1$

defines the value of the attribute **refs**$_1$ to be a reference chain of length 1. In the third alternative, the action symbol:

<u>**add one ref**</u> ↓ **refs**$_2$ ↑ **refs**$_1$

defines the length of the reference chain represented by **refs**$_1$ to be 1 greater than that represented by **refs**$_2$.

An example of the use of action symbols and the value-passing mechanism is shown in Figure 10. This diagram depicts the sequence of attribute evaluations for obtaining the value of the **env** attribute in the derivation tree for the ASPLE program:

begin
int A;
. . .
end

As before, only those attributes that contribute to the evaluation of the **env** attribute are ncluded in the figure.

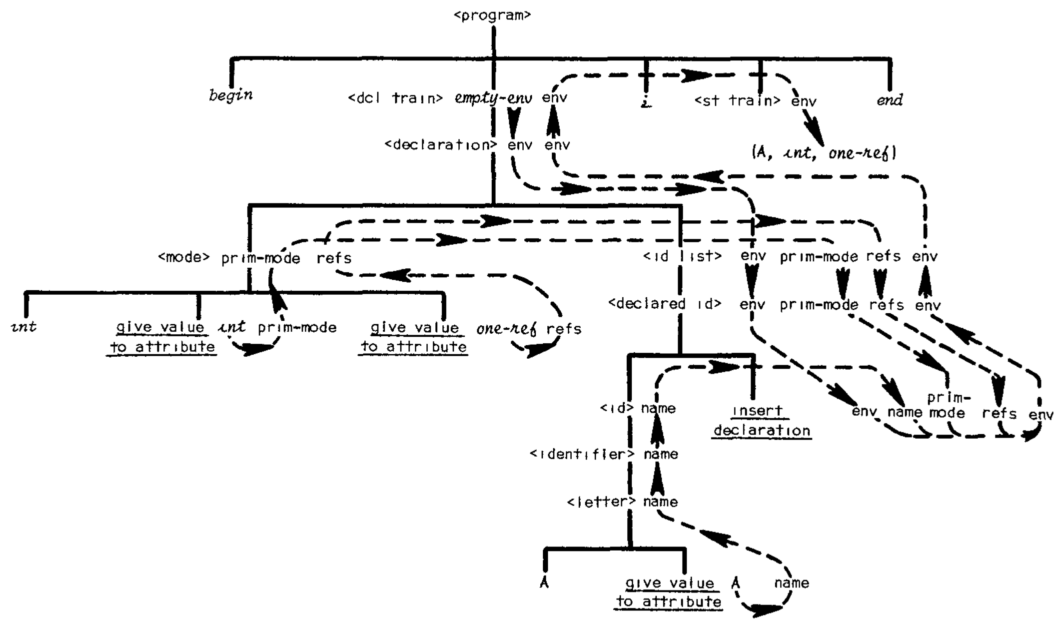

Figure 10. Derivation tree for attribute grammars.

So far we have only considered action symbols whose attributes can be evaluated during the translation phase. In the following examples, we encounter action symbols that are also part of the translation of the program and that are executed during the subsequent execution phase. Those attribute values and action symbols that in general can only be evaluated during the execution phase are written in upper case characters. An example occurs in the definition of the loop statement in [AG11]:

[AG11] **<loop stm>** ↓ **env** ::= *while*
LOCATE ↑ **label**$_1$
<exp> ↓ **env** ↓ *zero-refs* ↑ **prim-mode** ↑ **VALUE**
BRANCH ON FALSE ↓ **VALUE** ↓ **label**$_2$
do
<stm train> ↓ **env**
end
BRANCH ↓ **label**$_1$
LOCATE ↑ **label**$_2$
condition: **prim-mode** = *bool*

The last attribute of **<exp>** represents the value of the actual expression. This attribute is written in upper case characters to show that it can only be evaluated during the execution of the program. The action symbols written in upper case characters can be regarded as part of the translated program. The left-to-right order of the italic terminals in the derivation tree specifies the written form of the source program. Similarly, the left-to-right order of the upper case action symbols in the derivation tree specifies the translation of the source program. During execution of the program, these symbols are interpreted strictly according to their written sequence, except for deviations caused by the **BRANCH** actions. These actions change the execution sequence, making use of label attributes that are evaluated by the **LOCATE** action. In the case of the loop statement, the control flow during the execution phase is as indicated in Figure 11.

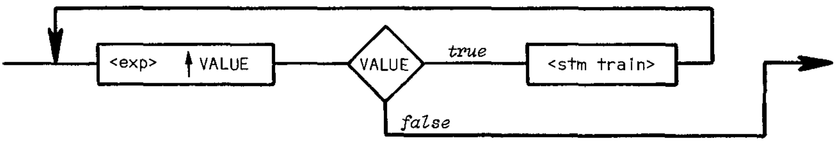

Figure 11. Control flow in loop statement.

The first three attributes of the category **<exp>** can be evaluated during the translation phase. The second attribute indicates the length of the reference chain in the mode of the expression value. If necessary, a sufficient number of dereferencing operations have to be performed. In the case of the loop statement, this attribute is set to *zero-refs*, since an actual primitive value is required. The value of the synthesized attribute **prim-mode** is the primitive mode, that is, integral or Boolean, of the expression value, which is determined in the subtree of the node **<exp>** according to the inherited attribute **env** and the program text. The condition specifies that this primitive mode must be Boolean.

Dereferencing is described by the production [AG22]:

[AG22]
<deref action> $\downarrow$ **name** $\downarrow$ **refs$_1$** $\downarrow$ **refs$_2$** $\uparrow$ **VALUE$_1$**
  ::= **give value to attribute** $\downarrow$ **name** $\uparrow$ **VALUE$_1$**
    condition: refs$_1$ = refs$_2$
      [*no dereferencing is necessary*]
  | **LOAD** $\downarrow$ **name** $\uparrow$ **VALUE$_2$**
      [*an undefined stored value gives rise to an error condition*]
      [*one level of dereferencing is done*]
    subtract one ref $\downarrow$ **refs$_1$** $\uparrow$ **refs$_3$**
      [**refs$_1$** *will always be greater than zero*]
    <deref action> $\downarrow$ **VALUE$_2$** $\downarrow$ **refs$_3$** $\downarrow$ **refs$_2$** $\uparrow$ **VALUE$_1$**
    condition: refs$_1$ > refs$_2$
      [*several levels of dereferencing can be done recursively. The number of times the recursion is invoked depends on the difference of the values of refs$_1$ and refs$_2$*]

The attributes **name** and **refs$_1$** represent an identifier name and the value of the length of its reference chain, respectively, **refs$_2$** represents the length of the reference chain required for the mode of the value to be obtained. The subtree of **<deref action>** performs the necessary dereference operations and returns the value as a synthesized attribute. This subtree does not generate any terminal source symbols. However, it generates **LOAD** actions for the execution phase. The structure of the subtree, and the number of **LOAD** actions generated, depend on the values of the attributes **refs$_1$** and **refs$_2$**. Similarly, the choice of the appropriate alternative of the production [AG15], and others, depends on the value of the inherited attribute, **prim-mode**. For example, consider the program:

```
begin
  int A;
  ref int B;
  A := 0;
  B := A;
  while (A ≠ 12) do
    A := A + 2
  end;
  output B
end
```

Using the productions of Table 5.1, after all possible attributes on the derivation tree of the program have been evaluated during the translation phase, the only action symbols that remain for the execution phase are:

STORE $\downarrow$ *'A'* $\downarrow$ *0*
STORE $\downarrow$ *'B'* $\downarrow$ *'A'*
LOCATE $\uparrow$ **label$_1$**
LOAD $\downarrow$ *'A'* $\uparrow$ **value$_1$**
COMPARE NOT EQUAL $\downarrow$ **value$_1$** $\downarrow$ *12* $\uparrow$ **value$_2$**
BRANCH ON FALSE $\downarrow$ **value$_2$** $\downarrow$ **label$_2$**
LOAD $\downarrow$ *'A'* $\uparrow$ **value$_3$**
ADD $\downarrow$ **value$_3$** $\downarrow$ *2* $\uparrow$ **value$_4$**
STORE $\downarrow$ *'A'* $\downarrow$ **value$_4$**
BRANCH $\downarrow$ **label$_1$**
LOCATE $\uparrow$ **label$_2$**
LOAD $\downarrow$ *'B'* $\uparrow$ **value$_5$**
LOAD $\downarrow$ **value$_5$** $\uparrow$ **value$_6$**
WRITE INTEGRAL $\downarrow$ **value$_6$**

This sequence of action symbols with the indicated attribute values is the translation of the source program and represents the meaning of the program.

As has been shown, the attribute grammar approach that uses action symbols relies on the existence of some other target language for specifying semantics. In the case of ASPLE, this target language consists of the action symbols informally described in Table 5.2. They operate over three global variables: the **memory** state, the input **file** state, and the output **file** state. These states are changed by a number of actions that take place during the execution of the program.

6. CRITIQUE OF THE DEFINITION TECHNIQUES

The four formal definitions of ASPLE illustrate a variety of models whose usability can be compared. Any full definition of a programming language must supply information to a range of users. Language designers need to review their work and to assess the full impact of their design decisions. Language implementors need a precise formulation of a language as part of their job description. Writers of textbooks and reference manuals need information at all levels, from the general to the particular. Serious programmers need to resolve detailed questions about facets of the language that are often omitted from informal language definitions.

To all these users, the formal definition must be a definitive source of answers to their questions. Beyond this essential minimum function, the quality of the definition is critically determined by the ease with which users can obtain the required information. As an illustration, Table 6.1 lists six questions that might be posed about ASPLE. To compare the four definition techniques we will consider Question 4:

In this example ASPLE program, is the assignment of an integer constant to the variable X valid?

```
begin
  ref int X;
  X := 2
end
```

and follow through the process of obtaining answers from each definition. We will also look at each definition in a critical light.

W-grammars

Since the question involves the assignment statement, we first look for a hyperrule for assignments. Hyperrule [HR10] contains the protonotion for assignment:

[HR10] **TABLE TAG becomes EXP val assignment:**
TABLE ref MODE TAG identifier,
$\underline{:=}$,
TABLE EXP MODE value.

This hyperule shows that the right side of an assignment statement must be derivable from

TABLE EXP MODE value

1) General question about the language:
   What data types are available in ASPLE?

2) More detailed question on the data types of the language:
   Are mixed mode expressions permitted in ASPLE?

3) Detailed question on the context-free syntax of the language:
   In this example ASPLE program, is the semicolon after the second input statement correct?

```
begin
  int X;
  input X;
  while (X ≠ 0) do
    output X;
    input X;
  end
end
```

4) Detailed question on the context-sensitive syntax of the language:
   In this example ASPLE program, is the assignment of an integer constant to the variable X valid?

```
begin
  ref int X;
  X := 2
end
```

5) Detailed question on the semantics of the language:
   In this example ASPLE program, is the disjunction between two variables, one of which has the value true and the other has an undefined value, legal?

```
begin
  bool A, B;
  A := true;
  if (A + B)
    then B := true
    else B := false
  fi
end
```

6) Detailed question on the implementation defined features of the language:
   In this example ASPLE program, is the value printed defined by the language or is it dependent on the implementation?

```
begin
  int X, Y;
  X := 1;
  Y := 1;
  while (X ≠ 1000) do
    output Y;
    X := X + 1;
    Y := Y * 2
  end
end
```

TABLE 6.1 SAMPLE QUESTIONS ON ASPLE

Following this form takes us through several hyperrules, **[HR17]**, **[HR19]**, and **[HR20]**:

[HR17] **TABLE EXP MODE value:**
  TABLE EXP MODE factor.

[HR19] **TABLE EXP MODE factor:**
  TABLE EXP MODE primary.

[HR20] **TABLE EXP MODE primary:**
  strong TABLE EXP MODE identifier;
  TABLE EXP MODE value pack;
  MODE EXP denotation,
    where MODE is INTBOOL;
  TABLE EXP compare pack,
    where MODE is bool.

Since the right side of the assignment is a constant, a "denotation" in the W-grammar, we choose the third alternative. The uniform replacement rule applied to hyperrule **[HR10]** causes MODE in hyperrule **[HR20]** to be replaced by the declared mode of the target of the assignment. The phrase

where MODE is INTBOOL

from hyperrule **[HR20]** specifies that this mode must be *int* or *bool*. Hence the mode *ref int* is not permitted for X and the assignment statement is illegal in the given program.

Conceptually, the W-grammar is the simplest of the formal systems presented here. All aspects of the definition are covered by a single formalism that is based on the familiar notion of context-free grammars. However, this one formalism has been pushed to an extreme. The reader must simultaneously follow protonotions down several branches of the tree keeping in mind many possible replacements and combinations.

The expression of a complete definition in a formalism based entirely on symbol manipulation leads to some unnatural constructions. For example, all arithmetic must be performed on sequences of **one**'s. This technique is at first difficult to understand. Only after considerable thought can the reader make the appropriate mental abstraction. However, it should be noted that the W-grammar definition is the only one of the four that defines the arithmetic operations fully. Once the reader has verified the way that the arithmetic works, **plus** and **times** serve as abstractions for that part of the derivation tree.

The use of a generative grammar for the definition of semantics is not followed exclusively. There are points at which this approach has been abandoned and the explicit detection of errors is used for clarity. For example, in hyperrule **[HR75]**:

[HR75] **where NUMBER matches INTBOOL:**
  where INTBOOL is int;
  where INTBOOL is bool,
  [*input error*] **abnormal termination.**

a mismatch of types during input is specifically trapped. The reasons for the distinction between explicit and implicit detection of errors is a property of the definition and is not concerned with the semantics of ASPLE.

Production Systems

Here, we go directly to the production that deals with assignment statements:

[PS07]
stm **ASGT STM** <**id** $\underline{:=}$ **exp**> & **LEGAL** <*:ρ>
← **LEGAL**<**id**:ρ> & **LEGAL**<**exp**:ρ> &
$\mathbf{dm}_\ell \equiv$ **DERIVED EXP MODE**(**id**:ρ) & $\mathbf{dm}_r \equiv$ **DERIVED EXP MODE**(**exp**:ρ) &
PRIM MODE($\mathbf{dm}_\ell$) = **PRIM MODE**($\mathbf{dm}_r$) &
[*The primitive modes of id and exp in ρ must be identical*]
$\mathbf{n}_\ell \equiv$ **NUM REFS**($\mathbf{dm}_\ell$) & $\mathbf{n}_r \equiv$ **NUM REFS**($\mathbf{dm}_r$) & $\mathbf{n}_\ell \leq \mathbf{n}_r + 1$.
[*The mode of id must be obtainable from the mode of exp by deferencing exp*]

From this we see that the primitive mode of the identifier must be identical to the primitive mode of the expression. We also see that $\mathbf{n}_\ell$, the value of **NUM REFS** of the declared mode of the identifier, must be less than or equal to $\mathbf{n}_r + 1$, the value of **NUM REFS** of the declared mode of the expression plus 1. The value of **NUM REFS** for the identifier X declared as *ref int* is derived from the following rules:

[PS42] **DERIVED MODE**(*int*) ≡ **REF INTEGER**.

[PS44] **DERIVED MODE**(*ref* **m**) ≡ **REF dm**
← **dm** ≡ **DERIVED MODE** (**m**).

[PS45] **NUM REFS**(**INTEGER**) ≡ 0.

[PS47] **NUM REFS**(**REF dm**) ≡ 1 + **NUM REFS**(**dm**).

From these we see that the value of **NUM REFS** for an identifier is one more than the number of occurrences of *ref* in the declaration for the identifier. For X, the value of **n** is 2. The value of **NUM REFS** for the expression, an integer constant, is obtained from:

[PS37] **DERIVED EXP MODE**(**int**:ρ) ≡ **INTEGER**.

[PS45] **NUM REFS**(**INTEGER**) ≡ 0.

The value of $\mathbf{n}_r$ is thus 0. Applying these values to the relation in production [PS07]:

$$\mathbf{n}_\ell = 2 \qquad \mathbf{n}_r = 0 \qquad \mathbf{n}_\ell > \mathbf{n}_r + 1$$

the assignment is shown to be illegal in the given context.

The notation for Production Systems is based on a combination of generative and analytic concepts. Sets are defined generatively and the properties are defined analytically. This interplay leads to definitions that are short and provide some degree of abstraction. Furthermore, the use of a static environment leads to a conceptually clear definition of the context-sensitive requirements. If the user is only concerned with the context-free syntax, only the left-most conclusion in each production need be considered and all premises and predicates involving an environment may be ignored. One debit with Production Systems is that they have not been used for the direct definition of semantics. The user is therefore required to learn another method.

The axiomatic approach to semantics is based primarily on generative concepts and does not rely on any machine model of execution. It concentrates on the essence of semantics by specifying only relevant assertions about objects and operations. The approach also has the advantage of giving the user tools for proving properties about programs. The major debit is the need to make mental leaps in order to select the relevant assertions. Since the process is generative, the detection of errors is implicit rather than explicit.

Vienna Definition Language

Since the legality of the statement can be determined statically, we start with the function **trans-asgt-stm** [T05] in the Translator:

```
[T05]           trans-asgt-stm(t) =
                  valid-mode-for-assignment(t) → translate-assignment(t)
                  true                         → error
                    [modes not compatible for assignment]
```

Here we see that the predicate function **valid-mode-for-assignment** is used to check the legality of the statement before translation. This predicate:

```
[T25]
   valid-mode-for-assignment(t) =
     (primitive-mode(s1(t)) = primitive-mode(s3(t)) &
                               (ref-chain-length(s1(t))- 1 ≤ ref-chain-length(s3(t)))
     [true if the mode of the right side of an assignment statement is valid for assignment to the left side]
     [where: is-c-id(s1(t)) and is-c-exp(s3(t))]
```

requires that the primitive-mode of the identifier on the left match the primitive-mode of the expression on the right. Also, the value of **ref-chain-length** for the identifier must not be greater than 1 plus the value of **ref-chain-length** for the expression. From the definition:

```
[T19]       ref-chain-length(t) =
              is-c-id(t) → slength(s1·mode-of-id(t)) + 1
                             [this is an elementary object satisfying is-integer]
              true       → 1
              [where: s1·mode-of-id(t) is the list of ref's in the declaration of the identifier t]
```

the value of **ref-chain-length** for an identifier is one more than the number of occurrences of *ref* in its declaration. For the variable X, this value will be 2. The value of **ref-chain-length** for any other type of expression, including constants, is 0. Thus the relationship in **valid mode-for-assignment** does not hold and the statement is rejected as being illegal in the context.

The VDL approach is based entirely on the model of a hypothetical machine. The concept of a computer is familiar to many users and an abstract machine provides a precise and readily grasped metaphor. Because of the resemblance between the hypothetical machine and real computers, implementation restrictions can be introduced naturally.

A VDL definition is split into two parts, the Translator and the Interpreter. For many languages there is no sharp distinction between the statically and dynamically applied rules, and the writer of the VDL definition is forced to superimpose this structure. The dividing line will generally be drawn in order to make both parts as clear as possible, and in a large language, there are bound to be some arbitrary decisions.

One debit of the approach is that the use of the hypothetical machine brings extraneous detail into the definition that tends to obscure its meaning. For example, the mechanism for passing values from one operation to another in the Interpreter has no direct connection with ASPLE semantics. The mechanistic nature of this definition technique provides little help in deriving general properties of language constructs. The user can only attempt to draw conclusions about the general behavior of these constructs from specific examples.

For most programming languages it is not too difficult to draw these conclusions. The explicit detection of errors by the interpreter helps the user, particularly the implementor, to understand the language more easily.

Attribute Grammars

The production for the assignment statement is [AG09]:

[AG09] **<asgt stm> ↓ env ::= <used id> ↓ env ↑ prim-mode$_1$ ↑ refs$_1$ ↑ name**
<u>:=</u>
<u>subtract one ref</u> ↓ refs$_1$ ↑ refs$_2$
<exp> ↓ env ↓ refs$_2$ ↑ prim-mode$_2$ ↑ VALUE
<u>STORE</u> ↓ name ↓ VALUE
condition: prim-mode$_1$ = prim-mode$_2$
[*primitive modes must be compatible for assignment*]

The syntactic category **<used id>** is specified in production [AG21]:

[AG21] **<used id> ↓ env ↑ prim-mode ↑ refs ↑ name**
::= <id> ↑ name
condition: (name, prim-mode, refs) ε env

This shows that the number of references, the value of the attribute **refs**, associated with the identifier is to be obtained from the environment. This value is specified in [AG05]:

[AG05] **<mode> ↑ prim-mode ↑ refs$_1$**
::= *bool*
<u>give value to attribute</u> ↓ *bool* **↑ prim-mode**
<u>give value to attribute</u> ↓ *one-refs* **↑ refs$_1$**
| *int*
<u>give value to attribute</u> ↓ *int* **↑ prim-mode**
<u>give value to attribute</u> ↓ *one-refs* **↑ refs$_1$**
| *ref*
<mode> ↑ prim-mode ↑ refs$_2$
<u>add one ref</u> ↓ refs$_2$ ↑ refs$_1$

The value of **refs** is one greater than the number of occurrences of *ref* in the declaration. Thus the value of the attribute **refs** associated with **<used id>** in production [AG09] is 2. This is reduced by 1 by the action symbol **<u>subtract one ref</u>** to give 1 as the value of the attribute **refs** passed to the production for **<exp>**. Following this attribute through the productions for **<exp>** and **<factor>**, we arrive at the production for **<primary>** [AG20]:

[AG20] **<primary> ↓ env ↓ refs$_1$ ↑ prim-mode ↑ VALUE**
::= <used id> ↓ env ↑ prim-mode ↑ refs$_2$ ↑ name$_1$
<deref action> ↓ name$_1$ ↓ refs$_2$ ↓ refs$_1$ ↑ VALUE
[*some dereferencing may possibly be done*]
| **<constant> ↑ prim-mode ↑ VALUE**
condition: refs$_1$ = *zero-refs*
| **<u>(</u>**
<exp> ↓ env ↓*zero-refs* **↑ prim-mode ↑ VALUE**
<u>)</u>
condition: refs$_1$ = *zero-refs*
| **<u>(</u>**
<compare> ↓ env ↑ VALUE
<u>)</u>
<u>give value to attribute</u> ↓ *bool* **↑ prim-mode**
condition: refs$_1$ = *zero-refs*

Since the right side of the assignment is a constant, the second alternative applies and the condition stipulates that **refs** =*zero-refs*. Since the value of **refs**$_1$ is 1, the assignment statement is illegal in the given context.

This method clearly shows the underlying context-free syntax of the language. By overlaying the evaluation of attributes on the parse tree, the interrelation between the various parts of the tree is seen. Clarity is helped by including the attributes in the productions, thus keeping the information localized.

Attribute Grammars are limited in the amount of attribute evaluation that can be performed directly and by the lack of a method for defining the semantics. These require further action symbols. While action symbols correspond most closely to an actual implementation and may appeal to writers of compilers, the formal definition of the action symbols is troublesome. There is no way that this can be done within the Attribute Grammar system, though it would be possible to replace the action symbols with some other more formal system. This would require the user to learn a second formalism to understand the definition.

Evaluation

A comparative evaluation of the four techniques is indeed subjective. One way of presenting such an evaluation is in a tabular form, similar to that used for computer system selections or for reports on cars, shavers, and other objects whose characteristics are mainly assessed subjectively. Table 6.2 was obtained by combining the views of the authors of this paper. Although there was some disparity between these views, the disagreement was not large, and no great feat of compromise was required in deriving the table.

Some remarks on this table are in order.

- *Completeness.* By this we mean the ability of a formal system to define the entire programming language. As we have presented the formal definitions here, only the Attribute Grammars are incomplete, though they could have been coupled with axioms in the same way that we have done for Production Systems.
- *Simplicity of model.* There are two aspects to this criterion: the initial difficulty of learning the model, and the effect of the model on the clarity of the definition itself. Here, we only evaluate the first of these. The second is subsumed in other criteria. It could be argued that the initial difficulty of learning the technique is of relatively minor importance since this is only a "one time expense."
- *Clarity of defined syntax.* In particular, this includes the definition of the context-sensitive requirements. We believe that isolation of these requirements from the context-free specification and semantics is important to clarity.
- *Clarity of defined semantics.* This is the category in which we had the greatest divergence of opinion. Each of us found the technique we knew the best to be the clearest.
- *Ability to show errors.* It is not clear how valuable it is for a definition to show errors. From the theoretical point of view, a definition need only define the class of *legal* programs and their meaning. From the practical point of view, however, many of the questions that a definition will have to answer will be of the form shown in Table 6.1 and the explicit indication of errors is helpful in providing replies. It is probably of assistance to compiler writers that the definition show errors in the source program explicitly.
- *Ability to show details.* This criterion measures the ease with which the user can find detailed information about the language.
- *Ease of modification.* This is of great importance during the design of the language, but much less so once the design is complete.

	W-GRAMMARS	PRODUCTION SYSTEMS	AXIOMATIC APPROACH	VIENNA DEFINITION LANGUAGE	ATTRIBUTE GRAMMARS
COMPLETENESS	+	NA	NA	+	–
SIMPLICITY OF MODEL	+	+	0	–	0
CLARITY OF DEFINED SYNTAX	–	+	NA	0	+
CLARITY OF DEFINED SEMANTICS	0	NA	0	0	0
ABILITY TO SHOW ERRORS IN PROGRAMS	–	+	0	+	0
ABILITY TO SHOW DETAILS	0	+	0	+	0
EASE OF MODIFICATION	0	0	0	0	0

RATINGS:
+ Positive
0 Neutral
– Negative
NA Not Applicable

TABLE 6.2 COMBINED AUTHOR RATINGS OF THE DEFINITION METHODS

7. FORMAL DEFINITIONS IN GENERAL

At present, most formal definitions are used exclusively by humans. The direct machine use of formal definitions is limited and is used primarily for the automatic construction of recognizers from context-free grammars. Even with great advances in compiler technology, humans will remain the major users of formal definitions.

While it may seem to be trite to remark on the importance of clarity in formal definitions for human use, the subject of clarity has hitherto received but scant attention. Completeness and conciseness have generally been considered to be of greater importance. Completeness is indeed important, so important that it must be assumed in any formal definition without special comment. Conciseness, while sometimes helpful to clarity, is a dangerous mistress. She is the siren that lures programmers onto shoals of octal coding and the APL one-liner.

A comparison of the way our four definitions answer the sample questions shows that clarity depends critically on the formal model being used, and on what the reader is used to. However, even with a given formal system, there is still room to exercise the care and talent of the writer. The method of presentation also plays a vital part in the formal mechanism.

In the preparation of the example definitions in this paper, we have taken care to promote clarity. Among the principles we have used are:

- introduction of the minimum amount of notation required for the definition of ASPLE;
- use of abbreviations only where there is a clear gain in readability;

- separation of context-free, context-sensitive, and the semantic parts of the language as much as possible;
- arrangement of the tables in a way that makes them easy to read, even at the expense of almost doubling the conventional space requirement;
- selection of mnemonic names that help the reader in making abstractions;
- use of different type styles to separate different types of objects;
- use of comments.

It is clear that for a language of any magnitude, the production of a formal definition without the aid of some text preparation system is almost impossible. The incidence of typographic errors will always be too high to produce reliable tables. Even with the small tables we have produced here, we have had problems of this sort. Had we had access to a document preparation system with output provided in a choice of type styles, we would certainly have used it.

It should be remembered that our four definitions describe a toy language only. Even so, the labor of producing the tables was considerable, requiring at least a week for a first draft and then a large number of iterations to remove errors and improve clarity. For real programming languages, the mass of detail required in any formal definition becomes immense. A complete understanding and checking of such a definition certainly approaches and may exceed human abilities.

While there can be little argument about the need for clarity in formal definitions, there are several topics where debate continues.

What Constitutes a "Valid" Program?

Since a definition provides rules for selecting the set of legal programs from the set of all possible strings in the language, it is important that the properties of a "valid" or "legal" program be defined. There are several possibilities; for example, a valid program may be defined as one with:

1) no context-free syntax errors;
2) no context-free or context-sensitive syntax errors;
3) no syntax errors and whose execution terminates when encountering a particular set of input data;
4) no syntax errors and whose execution terminates for all possible sets of input data;
5) no syntax errors and whose execution terminates for all possible sets of input data and produces a "correct" answer.

In our example definitions, Production Systems and Attribute Grammars go as far as level 2). VDL and W-grammars include level 3) and the axiomatic approach allows level 4). However, only the W-grammars, by a requirement for a finite tree, touch on the problem of termination. A final opinion on this issue is left open.

How Should a Formal Definition Show Errors?

There are two fundamentally different ways that formal definitions specify "errors." A definition may be *analytic*, rejecting erroneous programs explicitly, or the definition may be *generative*, making it impossible to generate an erroneous program. From the user's point of view, the generative method leaves the question of whether a program is really erroneous or whether the user has not been able to think of a way to use the grammar to generate the program. None of our sample definitions takes a pure position in this matter. For example, VDL rejects programs with context-sensitive or semantic errors explicitly, but uses a generative approach that prevents the construction of a program with a context-free syntax error. The W-grammar is mainly generative but detects some semantic errors explicitly. Of our four definitions, the VDL formalism shows errors the most clearly.

How Should Definitions Show Implementation Restrictions?

Two subsidiary questions are: 1) How should definitions attempt to indicate the places where an implementation may introduce restrictions?; and 2) Furthermore, is it possible to foresee all such restrictions?

The second question begs the prior question, whether a language definition should allow any implementation-defined restrictions. If the language is *completely* specified by the designer, the implementor may be forced to take uneconomic expedients to meet the specification exactly. It may be a contractual condition that the language definition be completely implemented.

With the technology available at this time, it seems that the implementor must be left with several points at which he is free to make decisions. We contend that these implementation-defined points, if any, should not be ignored, but explicitly shown in the formal definition. It is important to users of a language, as well as to implementors, to know what can be counted on in all implementations. The question whether it is possible to foresee all such restrictions is still open. Currently, the closest to a formal definition for an official language standard is the draft proposed standard for PL/I [E2]. This uses a VDL-like model of an abstract machine, but the algorithms are expressed more informally in a disciplined style of English prose. This specification has attempted to mark all the implementation-defined features by listing 40 of them. However, the definition permits a standard implementation to make quantitative restrictions that are not included in the list. Much of the reason for this is not connected with the technology of the definition but with the more practical legal question of restraint of trade.

8. IMPORTANCE OF FORMAL DEFINITIONS

Because BNF is clear and easy to use, most definitions of programming languages include a BNF description of the context-free syntax. This is generally as far as the formal content of the definitions go. As a result, there is an unfortunate tendency to believe that this is all that is required of a formal definition. There is an analogous confusion in many textbooks on compilers where the subject matter is limited to the theory of parsing. In formal definitions, as with compilers, the more difficult parts are the context-sensitive requirements and the semantics.

It is precisely in the context-sensitive and semantic areas that formalism is needed. There is generally little argument over the precise syntax of a statement even if there is no formal description of it. All too often, however, an intuitive understanding of the semantics turns out to be woefully superficial. It is only when an attempt at implementation (which is, after all, a kind of formal definition) is made that ramifications and discrepancies are laid bare. What was thought to have been fully understood is discovered to have been differently perceived by various readers of the same description. By then, it is frequently too late to change, and incompatibilities have been cast in actual code.

Our example definitions indicate that the technology for full definitions is available but that there is still much work to do before any notation achieves the level of general acceptance of BNF. This work must overcome considerable user resistance. For example, the definition of the proposed standard for PL/I [E2], the VDL definition of PL/I [L7], and the W-grammar definition of ALGOL 68 [W2] have all received mixed reactions. Resistance to formal definitions will only be overcome by great attention to the human engineering so that the general user feels that the definition is understandable by other than formal definition specialists.

Despite the urgent need for the development of readable formal definitions, formal definitions must never be thought of as self-contained arenas with no user contacts. The interface with users is the key area where most of the effort is needed. The metalanguage of a formal definition must not become a language known to only the high priests of the cult. Tempering science with magic is a sure way to return to the Dark Ages.

ACKNOWLEDGMENTS

We are very grateful to Elliott Organick, Frederic Richard, Randall Mercer, Robert Brown, C. H. A. Koster, and the four referees, who helped to clarify the formalisms presented here. Participants in the PASCAL Assistant project at University of Massachusetts sharpened our concern for human factors. Finally, we acknowledge the skills of Edwina Ledgard and Linda Strzegowski in the preparation of this paper.

REFERENCES

[B0] BÉKIC, H.; BJØRNER, D.; HENHAPL, W.; JONES, C. B.; AND LUCAS, P. *A formal definition of PL/I subset*, IBM Technical Report 25.139, IBM Lab. Vienna, Austria, 1974.

[B1] BOCHMANN, G. V. "Semantic evaluation from left to right," *Comm. ACM* **19**, 2 (Feb. 1976), 55–62.

[B2] BOCHMANN, G. V. *Attribute grammars and compilation: Program evaluation in several phases*, Technical Report #54, Dept. d'Informatique, Univ. de Montréal, Montréal, Canada, 1974.

[B3] BOSCH, R.; GRUNE, D.; AND MEERTENS, L. "ALEPH, a language encouraging program hierarchy," in *Proc. Internatl. Computing Symp.*, North-Holland Publ. Co., Amsterdam, The Netherlands, 1973.

[C1] CLEAVELAND, J.; AND UZGALIS, R. *What every programmer should know about grammar*, Dept. Computer Science, Univ. of California, Los Angeles, Calif., 1973.

[D1] DONAHUE JAMES E. *Complementary definitions of programming language semantics*, Springer-Verlag, New York, 1976.

[D2] DONOVAN, J.; AND LEDGARD, H. F., "A formal system for the specification of the syntax and translation of computer languages," in *Proc. AFIPS 1967 Fall Jt. Computer Conf.*, Vol. 31, Thompson Books, Washington, D.C., 1967, pp. 553–569.

[E1] ELGOT, C. C.; AND ROBINSON, A. "Random access stored-program machines. An approach to programming languages," *J. ACM* **11**, 4 (Oct. 1964), 365–399.

[E2] EUROPEAN COMPUTER MANUFACTURERS' ASSOC. AND AMERICAN NATIONAL STANDARDS INST., *PL/I BASIS/1-12*. Published as BSR X3.53, Computer and Business Equipment Manufacturers' Assoc., Washington, D.C., Feb. 1975.

[G1] GARWICK, J. V. "The definition of programming languages," in *Formal language description languages*, T. B. Steel, Jr., (Ed.), North-Holland Publ. Co., Amsterdam, The Netherlands, 1963, pp. 139–147.

[H1] HOARE, C. A. R.; AND WIRTH, N. "An axiomatic definition of the programming language PASCAL," *Acta Informatica* **2** (1973), 335–355.

[H2] HOARE, C. A. R. "Consistent and complementary formal theories of the semantics of programming languages," *Acta Informatica* **3** (1974), 135–153.

[K1] KNUTH, D. E. "Semantics of context-free languages," *Math. Systems Theory* **2**, (1968), 127–145 and *Math. Systems Theory* **5**, (1971), 95.

[K2] KOSTER, C. H. A. "Affix grammars," in *Algol 68 implementation*, North-Holland Publ. Co., Amsterdam, The Netherlands, 1971; see also CROWE, D. "Generating parsers for affix grammars," *Comm. ACM* **15**, 8 (August 1972), 728–734.

[L1] LANDIN, P. J. "Correspondence between ALGOL 60 and Church's lambda-notation," Part 1: *Comm. ACM* **8**, 2 (Feb. 1965), 89–101; Part 2: *Comm. ACM* **8**, 3 (March 1965), 158–165.

[L2] LEDGARD, H. F. "Production systems: Or can we do better than BNF?" *Comm. ACM* **17**, 2 (Feb. 1974), 94–102.

[L3] LEDGARD, H. F. "Production Systems: A notation for defining syntax and translation of programming languages." To appear in *IEEE Transactions on Software Engineering*, April 1977.

[L4] LEE, J. A. N. *Computer semantics*, Van Nostrand Co., New York, 1972.

[L5] LEWIS, P. M.; ROSENKRANTZ, D. J.; AND STERNS, F. E. "Attributed translations," presented at the *ACM Symp. on Theory of Computing*, Austin, Texas, April 1973.

[L6] LUCAS, P.; LAUER, P.; AND STIGLEITNER, H. *Method and notation for the formal definition of programming languages*, IBM Technical Report 25.087, IBM Lab., Vienna, Austria, 1968.

[L7] LUCAS, P.; AND WALK, K. "On the formal description of PL/I," *Annual Rev. Automatic Programming* **6**, 3 (1969), 105–182.

[M1] MCCARTHY, J. "Towards a mathematical science of computation," in *Information processing 1962*, C. M. Popplewell, (Ed.), North-Holland Publ. Co., Amsterdam, The Netherlands, 1963, pp. 21–28.

[M2] MCCARTHY, J. "A formal description of a subset of ALGOL," in *Formal language description languages*, T. B. Steel, Jr., (Ed.), North-Holland Publ. Co., Amsterdam, The Netherlands, 1963, 1–12.

[M3] MANNA, Z. "Properties of programs and the first-order predicate calculus," *J. ACM* **16**, 2 (April 1969), 244–255.

[P1] POST, E. L. "Formal reductions of the general combinatorial decision problem," *Amer. J. Math.* **65** (1943), 197–215.

[S1] SCHULER, P. F. "Weakly context-sensitive languages as a model for programming languages," *Acta Informatica* 3 (1974), 155–170

[S2] SCOTT, D.; AND STRACHEY, C. *Towards a mathematical semantics for computer languages*, PRG-6, Programming Research Group, Oxford Univ., Oxford, England, 1971.

[S3] SMULLYAN, R. M. "Theory of formal systems," in *Annals of mathematical studies*, No. 47, Princeton Univ. Press, Princeton, N.J., 1961.

[S4] STEEL, T. B., JR., "Standards for computers and information processing," in *Advances in computers*, Vol. 8, F. L. Alt and M. Rubinoff, (Eds.), Academic Press, New York, 1967.

[T1] TENNENT, R. "The denotational semantics of programming languages," *Comm. ACM* **19**, 8 (August 1976), 437–453.

[W1] WEGNER, P. "The Vienna definition language," *Computing Surveys* **4**, 1 (March 1972), 5–63.

[W2] VAN WIJNGAARDEN, A.; MAILLOUX, B. J.; PECK, J. E.; AND KOSTER, C. H. A. *Report on the algorithmic language Algol 68*, MR 101, Mathematisch Centrum, Amsterdam, The Netherlands, 1969.