# A SAMPLER OF FORMAL DEFINITIONS

By

Michael Marcotty*
Henry F. Ledgard*
Gregor V. Bochmann†

COINS Technical Report 75A-2
June 1975

Keywords: 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

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

† Department d'Informatique
Universite de Montreal
Montreal 101, Canada

Part of this work was supported by the U.S. Army Research Office

## ABSTRACT

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

We present four well-known formal definition techniques, W-grammars, Attribute Grammars, Production Systems with Hoare Axioms, and the Vienna Definition Language. Each technique is then applied to define the same small programming language. These definitions provide a basis for a discussion of the relative clarity of the different methods and for a review of some of the debatable issues of formal definition. Among these issues are the advantages, if any, to the definition 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.

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

"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. [S4]

## 1. INTRODUCTION

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

Sometimes a mixture of methods is used. The formalism is then 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 completely to full definitions.

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 provides rules for selecting the set $P \subset S^*$ of "legal" programs of the language being defined, and for each legal program $p \in P$, the definition also specifies its "meaning".

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

1. Should the definition model be based on the notion of an underlying machine?

2. 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?

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 forsee all such restrictions?

Indeed, we, the authors, have differing views on 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 most 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.

Several characteristics are important to the successful use of any method. Among these are:

1. Completeness. There must be no gaps in the definition. In particular, there should be no questions about the syntax or semantics of the language that cannot be answered from the definition.

2. Clarity. The user of the definition must be able to understand the definition and find answers to his questions easily. While it is obvious that some facility with the notation is essential before being able to understand it fully, the amount of effort required should be small.

3. Naturalness. The naturalness of a notation has a very large effect on the ability to use a definition approach. The naturalness of a notation is more important than its conciseness, although there is some relation between the two. We have, therefore, used notational abbreviations only where there was a real gain in clarity, and have chosen mnemonic names wherever possible

4. Realism. Although the designer of a language may like to think in a universe free from mundane restrictions like finite ranges for number values and bounded storage, these are the realities of the implementor's world. The definition given him by the design-

> er, his manufacturing specification, must tell him exactly where he can make restrictions or choices, and where the designer's unobstructed landscape must be modelled exactly.

We present here a prose description and four very different formal definitions of the same language. We then examine the ease with which answers to typical questions about the language can be obtained. The language used in the analysis is ASPLE [C1], where it is defined by a W-grammar, an extension of the method developed by van Wijngaarden [W2] to define ALGOL 68. Our first formal definition of ASPLE is derived from the definition 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 productions, the meta-productions and the hyper-rules. 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 of 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 [L6,L7,L4,W1]. In this method, a procedure is defined that takes a program string and transforms it 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 last formal definition technique is that of Attribute Grammars [K1,L5,B1]. In this approach, a context-free grammar is augmented 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 one to specify the context-sensitive requirements of a language and the meaning of a program by translating it into a separately defined sequence of actions.

One other major definition approach, that of Scott and Strachey [S2], is not considered in this paper. For more detail and a complete bibliography on this method, the reader is referred to a recent paper by Tennant [T1].

We make no attempt at a formal proof of the equivalence of the four definitions of ASPLE, but have relied on our own careful checking of the definitions. To assist the reader, we have included comments in the bodies of the actual definitions. These are separated from the formal part by being enclosed in brackets.

## 2. AN INFORMAL DESCRIPTION OF ASPLE

ASPLE is a very small language derived from ALGOL 68. Its context-free syntax is defined in Table 2.1 using Backus-Naur Form.

An ASPLE program consists 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: (a) the type of the value, integer or boolean, to which it may refer, and (b) whether the reference is direct 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 familiar syntax.

As 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 into X from the input file and setting the values of Y and Z both 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.

Table 2.1 BNF DESCRIPTION OF ASPLE

| | | |
|---|---|---|
| [B01] | <program> | ::= *begin* <dcl train> ; <stm train> *end* |
| [B02] | <dcl train> | ::= <declaration><br>\| <declaration> ; <dcl train> |
| [B03] | <stm train> | ::= <statement><br>\| <statement> ; <stm train> |
| [B04] | <declaration> | ::= <mode>, <idlist> |
| [B05] | <mode> | ::= *bool*<br>\| *int*<br>\| *ref* <mode> |
| [B06] | <idlist> | ::= <id><br>\| <id> , <idlist> |
| [B07] | <statement> | ::= <asgt stm><br>\| <cond stm><br>\| <loop stm><br>\| <transput stm> |
| [B08] | <asgt stm> | ::= <id> := <<exp> |
| [B09] | <cond stm> | ::= *if* <exp> *then* <stm train> *fi*<br>\| *if* <exp> *then* <stm train><br>*else* <stm train> *fi* |
| [B10] | <loop stm> | ::= *while* <exp> *do* <stm train> *end* |
| [B11] | <transput stm> | ::= *input* <id><br>\| *output* <exp> |
| [B12] | <exp> | ::= <factor><br>\| <exp> + <factor> |
| [B13] | <factor> | ::= <primary><br>\| <factor> * <primary> |
| [B14] | <primary> | ::= <id><br>\| <constant><br>\| (<exp>)<br>\| (<compare>) |
| [B15] | <compare> | ::= <exp> = <exp><br>\| <exp> ≠ <exp> |
| [B16] | <constant> | ::= <bool constant><br>\| <int constant> |
| [B17] | <bool constant> | ::= *true*<br>\| *false* |
| [B18] | <int constant> | ::= <number> |
| [B19] | <number> | ::= <digit><br>\| <number> <digit> |
| [B20] | <digit> | ::= *0* \| *1* \| ... \| *9* |
| [B21] | <id> | ::= <letter><br>\| <id> <letter> |
| [B22] | <letter> | ::= *A* \| *B* \| ... \| *Z* |

This sample ASPLE program uses only identifiers that refer directly to integral values, as for example, 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 reference-to-reference-to-integral. Executing the assignment:

```
B := A
```

sets the value of B to 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 changes the integral value to which A refers, the value that B refers to indirectly. To obtain the integral value to which B refers, the value of B must be "dereferenced" twice. 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 mode integral or boolean. The modes of the two values thus obtained must be identical. The operations + and * between integral values represent addition and multiplication respectively. Between boolean values they represent the logical "or" and "and" respectively. The operations = and ≠ 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. It must be possible, by dereferencing sufficiently, to obtain the same primitive mode from both sides.
2. If the mode of the identifier on the left side contains $n_r$ occurrences of "reference-to" and the mode of the value of the right side contains $n_\ell$ such occurrences, then the relation $n_\ell - 1 \leq n_r$

For example, given the declarations:

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

both the assignments:

| | |
|---|---|
| *A := 16* | $n_\ell = 1,\ n_r = 0$ |
| *C := D* | $n_\ell = 2,\ n_r = 3$ |

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

```
A := B
```

violates the first condition, and the assignments

| | |
|---|---|
| *C := 20* | $n_\ell = 2, \quad n_r = 0$ |
| *D := A* | $n_\ell = 3, \quad n_r = 1$ |

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 statement, consider the following program: (The line numbers are for reference only.)

```
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 the line 09 has been executed, two chains of references will have been set up. The state is shown schematically in Figure 2.1.

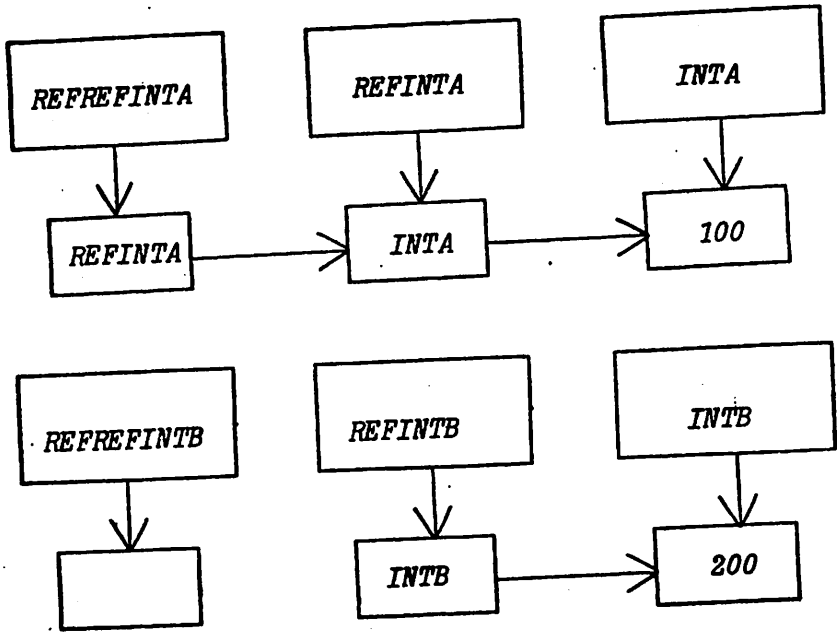


Figure 2.1

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 at this point is shown in Figure 2.2.

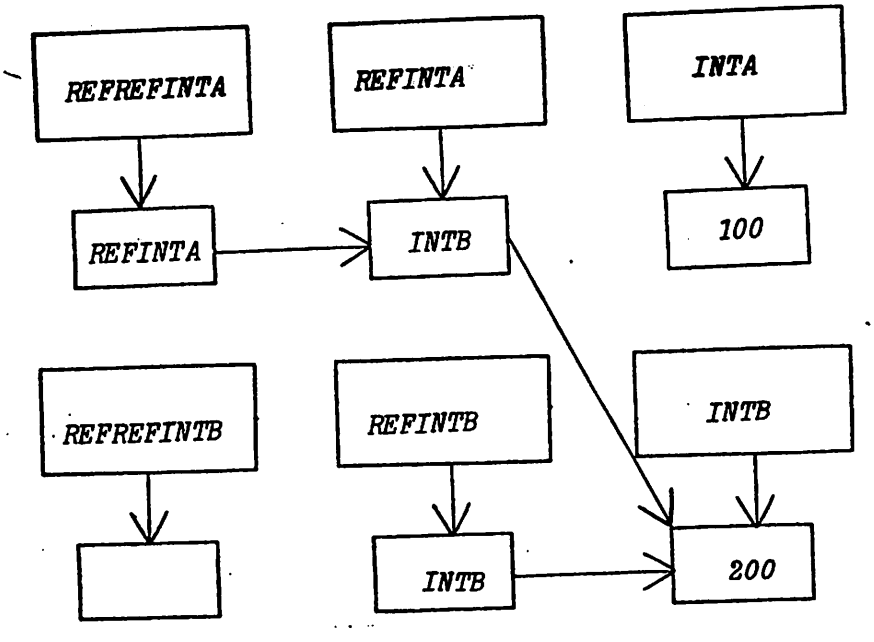


Figure 2.2

The assignment of line 11 makes no change in the value of *INTB* because of the effect of statement 10. The *input* statement of line 10 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 a *input* statement is executed.) The result is as depicted in Figure 2.3.

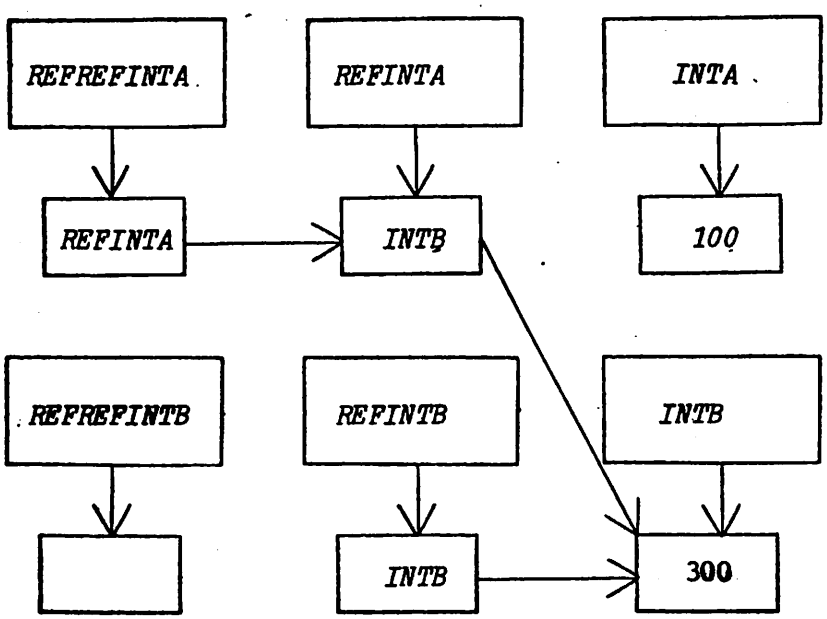


Figure 2.3

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 which 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 the length of the program. Any actual implementation will be bounded by machine constraints in these areas. Table 2.2 lists the points at which the implementor must supply information to complete the definition of the language.

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.

Table 2.2 Implementation defined features of ASPLE

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

# 3. W-GRAMMARS

The use of two-level grammars known as "W-grammars" as a definition technique was developed by van Wijngaarden and used for the description of Algol 68 [W2]. Cleveland 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].

## 3.1. The Notion of Meta-Productions

The restriction of BNF productions to the definition of the context-free parts of languages is due to the requirement that the number of productions be finite. W-grammars are able to define context-sensitive restrictions and semantics by using an infinite set of productions generated from a finite set of rules. These rules are divided into two groups, the meta-productions and the hyper-rules. The hyper-rules are prototypes for context-free productions, and the meta-productions describe how the context-free productions are to be derived.

Meta-productions are context-free productions. The non-terminals, called meta-notions, are written in upper-case letters, e.g. INTBOOL. The terminals, called proto-notions, are written in lower-case characters, e.g. letter. In this context, "lower-case characters", include all the symbols used for writing ASPLE programs. For example, ":=" and *int* are both proto-notions.

Consider the following meta-productions taken from the W-grammar of ASPLE in Table 3.1.

```
[MP01] ALPHA    :: a | b | ... | z | begin
                 | end | int | bool | ref | input
                 | input | output | true
                 | false | if | fi | then | else
                 | while | do | = | ≠ | + | *
                 | := | ⊥ | ⊮

[MP03] NOTION   :: ALPHA
                 | NOTION ALPHA

[MP07] INTBOOL  :: int
                 | bool

[MP08] MODE     :: INTBOOL
                 | ref MODE
```

## Table 3.1: Metaproductions for the W-grammar Definition of ASPLE

[MP01] ALPHA :: a | b | ... | z | *begin* | *end* | *int* | *bool* | *ref* | *input* | *input* | *output* | *true* | *false* | *if* | *fi* | *then* | *else* | *while* | *do* | = | ≠ | + | * | := | ± | ∅

[MP02] EMPTY ::

[MP03] NOTION :: ALPHA | NOTION ALPHA

[MP04] NOTETY :: NOTION | EMPTY

[MP05] TAG :: LETTER | TAG LETTER

[MP06] LETTER :: *A* | *B* | ... | *Z*

[MP07] INTBOOL :: *int* | *bool*

[MP08] MODE :: INTBOOL | *ref* MODE

[MP09] ONES :: one | ONES one

[MP10] NUMBER :: ONES | EMPTY

[MP11] RADIX :: one one one one one one one one one one

[MP12] BOOL :: *true* | *false*

[MP13] VALUE :: NUMBER | BOOL

[MP14] BOX :: VALUE | undefined | TAG

[MP15] LOC :: loc TAG has MODE refers BOX end

[MP16] LOCS :: LOC | LOCS LOC

[MP17] LOCSETY :: LOCS | EMPTY

[MP18] TABLE :: LOCS

[MP19] UNIT :: loop | assignment | conditional | transput

[MP20] SNAP :: memory LOCS FILE FILE

[MP21] SNAPS :: SNAP | SNAPS SNAP

[MP22] SNAPSETY :: SNAPS | EMPTY

[MP23] DATA :: EMPTY | space VALUE DATA

[MP24] FILE :: DATA end of file

[MP25] RELATE :: = | ≠

[MP26] OPER :: + | * | RELATE

[MP27] EXP :: left EXP OPER EXP right | VALUE | DEREFSETY TAG

[MP28] DEREFSETY :: EMPTY | deref DEREFSETY

[MP29] REFS :: *ref* | REFS *ref*

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

[MP31] STMTS :: STMT | STMTS STMT

[MP32] STMTSETY :: STMTS | EMPTY

[MP33] ALPHABET :: *ABCDEFGHIJKLMNOPQRSTUVWXYZ*

[MP34] MAXLEN :: . . .
[implementation defined measure of maximum program length $n_1$]

[MP35] MAXTABLE :: LOC LOC . . . LOC
[The number of occurrences of "LOC" is the implementation defined quantity $n_2$]

[MP36] MAXDIG :: ONES token ONES token . . . ONES token
[the maximum number of digits in the implementation defined quantity $n_3$]

[MP37] MAXLENGID :: LETTER LETTER . . . LETTER
[the number of occurrences of "LETTER" is the implementation defined quantity $n_4$]

[MP38] MAXINT :: one one . . . one
[the number of occurrences of "one" is the implementation defined quantity $n_5$]

[MP39] MAXFILELEN :: . . .
[implementation defined measure of maximum size of an output file $n_6$]

Using these meta-productions, we can generate:

[MP01] From the meta-notion ALPHA, any lower case symbol or blank [denoted by "ƀ"]

[MP03] From the meta-notion NOTION, any sequence of lower case symbols and blanks

[MP07] From the meta-notion INTBOOL, "*int*" and "*bool*"

[MP08] From the meta-notion MODE, infinitely many proto-notions consisting of a (possibly empty) sequence of the symbols "*ref*" followed by "*int*" or "*bool*".

A hyper-rule is a blueprint from which context-free productions can be generated. A context-free production is obtained from a hyper-rule by replacing each meta-notion by a proto-notion derived from the meta-productions. In the hyper-rule, all occurrences of the same meta-notion must be replaced by the same proto-notion. This is the uniform replacement rule.

For example, the hyper-rule,

```
[HR59]   NOTION sequence  ::=  NOTION,
                             |  NOTION sequence,
                                NOTION,
```

when used in conjunction with the meta-productions, [MP01] and [MP03], can generate an infinite set of productions. Two of these are:

```
value sequence   ::=  value,
                   |  value sequence, value,

abc sequence     ::=  abc,
                   |  abc sequence, abc,
```

since both value and abc can be derived from NOTION. However, the production

```
value sequence   ::=  abc,
                   |  value sequence, abc,
```

cannot be obtained since the uniform replacement rule would be violated. This simple substitution technique is used to generate the infinite set of context-free productions required for the specification of the context-sensitive requirements and semantics of a language.

Part of the philosophy of W-grammars is that the context-free productions, although generated mechanically, should be read almost like prose. To accomplish this, the non-terminals of the context-free productions generally define phrases, and the right sides of productions generally define sequences of other phrases separated by commas.

## 3.2. Overview of the W-grammar Definition of ASPLE

The meta-productions in Table 3.1 and the hyper-rules in Table 3.2 form a W-grammar that defines all aspects of the syntax and semantics of ASPLE. The starting hyper-rule, [HR01], shown here with line numbers added for reference, affords an overview of these three segments.

```
[HR01]   program ::= begin                                    01.
                     dcl train of TABLE1                      02.
                     TABLE1 restrictions,                     03.
                     TABLE1 STMTS stm train,                  04.
                     end,                                     05.
                     FILE1 stream,                            06.
                     FILE2 stream,                            07.
                     execute STMTS with                       08.
                        memory TABLE1 FILE1 end of file       09.
                        SNAPSETY                              10.
                        memory TABLE2 FILE3 FILE2             11.
```

Line 01 specifies that a program always starts with the symbol *begin*. Line 02 specifies a declare train. It contains the meta-notion $TABLE_1$ from which a proto-notion corresponding to the "symbol table" of the program can be derived. The subscript on $TABLE_1$ serves to distinguish this meta-notion from the meta-

## Table 3.2 Hyper-rules for the W-grammar Definition of ASPLE

[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 $TABLE_2$ $FILE_3$ $FILE_2$.

[HR02] dcl train of LOCS LOCSETY ::= MODE,
*ref* MODE definitions of LOCS,
;,
dcl train of LOCSETY,
| where LOCSETY is EMPTY,
MODE declarer,
*ref* MODE definitions of LOCS,
;.

[HR03] MODE definitions of loc TAG has MODE refers undefined end LOCSETY ::= TAG,
,,
MODE definitions of LOCSETY,
| where LOCSETY is EMPTY,
TAG,

[HR04] 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 $TAG_2$,

[HR05] 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,

[HR06] TABLE STMT STMTSETY stmts ::= TABLE STMT UNIT,
;,
TABLE STMTSETY stmts,
| where STMTSETY is EMPTY,
TABLE STMT UNIT,

[HR07] TABLE TAG becomes EXP val assignment ::= TABLE *ref* MODE TAG identifiers,
:=,
TABLE EXP MODE value,

[HR08] TABLE if EXP then $STMTS_1$ else $STMTS_2$ fi conditional ::= *if*,
TABLE EXP *bool* value,
*then*,
$TABLE\ STMTS_1$ stmts,
$TABLE\ STMTS_2$ elsend,

[HR09] TABLE STMTS elsend ::= *fi*,
where STMTS is EMPTY,
| *else*,
TABLE STMTS stmts,
*fi*,

[HR10] TABLE while EXP do STMTS end loop ::= *while*,
TABLE EXP *bool* value,
*do*,
TABLE STMTS stmts,
*end*,

## Table 3.2 Continued

```
[HR11]  TABLE EXP input transput  ::=  input,
                                       string TABLE EXP ref INTBOOL identifier,

[HR12]  TABLE EXP output transput  ::=  output,
                                        TABLE EXP INTBOOL value,

[HR13]  TABLE left EXP1 plus EXP2 right INTBOOL value  ::=  TABLE EXP1 INTBOOL value,
                                                            +,
                                                            TABLE EXP2 INTBOOL value,

[HR14]  TABLE EXP MODE value  ::=  TABLE EXP MODE factor,

[HR15]  TABLE left EXP1 times EXP2 right INTBOOL factor  ::=  TABLE EXP1 INTBOOL factor,
                                                              *,
                                                              TABLE EXP2 INTBOOL primary,

[HR16]  TABLE EXP MODE factor  ::=  TABLE EXP MODE primary,

[HR17]  TABLE EXP MODE primary  ::=  strong TABLE EXP MODE identifier,
                                   |  TABLE EXP MODE value pack,
                                   |  where MODE is INTBOOL,
                                      MODE EXP denotation,
                                   |  where MODE is bool,
                                      TABLE EXP compare pack,

[HR18]  TABLE left EXP1 RELATE EXP2 right compare  ::=  TABLE EXP1 int value,
                                                        RELATE,
                                                        TABLE EXP2 int value,

[HR19]  strong TABLE deref EXP MODE identifier  ::=  strong TABLE EXP ref MODE identifier,

[HR20]  strong TABLE TAG MODE identifier  ::=  TABLE MODE TAG identifier,

[HR21]  TABLE MODE TAG identifier  ::=  TAG,
                                        where TABLE contains loc TAG has MODE,
                                        where MAXLENGID contains TAG,

[HR22]  bool BOOL denotation  ::=  BOOL,

[HR23]  int NUMBER1 denotation  ::=  NUMBER1 token,
                                   |  int NUMBER2 denotation,
                                      NUMBER3 token,
                                      where NUMBER4 = NUMBER2 times RADIX,
                                      where NUMBER1 = NUMBER4 plus NUMBER3,
                                      where MAXDIG contains NUMBER1 denotation,

[HR24]  token  ::=  0,
[HR25]  one token  ::=  1,
[HR26]  one one token  ::=  2,
[HR27]  one one one token  ::=  3,
[HR28]  one one one one token  ::=  4,
[HR29]  one one one one one token  ::=  5,
[HR30]  one one one one one one token  ::=  6,
[HR31]  one one one one one one one token  ::=  7,
[HR32]  one one one one one one one one token  ::=  8,
[HR33]  one one one one one one one one one token  ::=  9,

[HR34]  space VALUE FILE stream  ::=  VALUE denotation,
                                      ⊥
                                      FILE stream,
                                   |  VALUE denotation,
                                      eof,
                                      where FILE is end of file,
```

## Table 3.2 Continued

[HR35] execute STMT STMTS with SNAPS$_1$ SNAP SNAPS$_2$ ::= execute STMT with SNAPS$_1$ SNAP,
execute STMTS with SNAP SNAPS$_2$.

[HR36] 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,

[HR37] 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$_2$.

[HR38] execute TAG becomes EXP val with SNAP$_1$ SNAP$_2$ ::= evaluate EXP from SNAP$_1$ 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$_2$.

[HR39] execute DEREFSETY TAG$_1$ input with SNAP$_1$ SNAP$_2$ ::= evaluate DEREFSETY TAG$_1$ from SNAP$_1$ giving TAG$_2$,
where SNAP$_1$ is
memory LOCSETY$_1$
loc TAG$_2$ has *ref* INTBOOL refers BOX$_1$ end
LOCSETY$_2$ space VALUE FILE$_1$ FILE$_2$,
where SNAP$_2$ is
memory LOCSETY$_1$
loc TAG$_2$ has *ref* INTBOOL refers VALUE and
LOCSETY$_2$ space VALUE FILE$_1$ FILE$_2$,
where VALUE matches INTBOOL,
where SNAP$_1$ is memory LOCS end of file FILE,
(*end of file error*) abnormal termination,

[HR40] where NUMBER matches INTBOOL ::= where INTBOOL is *int*,
| where INTBOOL is *bool*,
(*input error*) abnormal termination,

[HR41] where BOOL matches INTBOOL ::= where INTBOOL is *bool*,
| where INTBOOL is *int*,
(*input error*) abnormal termination,

[HR42] execute EXP output with SNAP$_1$ SNAP$_2$ ::= evaluate EXP from SNAP$_1$ giving VALUE,
where SNAP$_1$ is memory LOCS FILE$_1$ DATA end of file,
where SNAP$_2$ is memory LOCS FILE$_1$ DATA space VALUE end of FILE,
where MAXDATA contains DATA space VALUE end of file,
| evaluate EXP from SNAP$_1$ giving VALUE,
where SNAP$_1$ is memory LOCS FILE$_1$ DATA end of file,
where SNAP$_2$ is memory LOCS FILE$_1$ DATA space VALUE end of file,
where DATA space VALUE end of file contains MAXDATA,
(*output file overflow*) abnormal termination,

## Table 3.2 Continued

[HR43] execute empty with SNAP SNAP ::= true,

[HR44] evaluate left $EXP_1$ OPER $EXP_2$ right from SNAP giving $VALUE_1$ ::= evaluate $EXP_1$ from SNAP giving $VALUE_2$,
evaluate $EXP_2$ from SNAP giving $VALUE_3$,
where $VALUE_1$ equals $VALUE_2$ OPER $VALUE_3$,

[HR45] evaluate deref DEREFSETY TAG from SNAP giving $BOX_1$ ::= evaluate DEREFSETY BOX from SNAP giving $BOX_1$,
where SNAP contains loc TAG has MODE refers
$BOX_\_$ end,

[HR46] evaluate BOX from SNAP giving BOX ::= where BOX differs from undefined,
| where BOX is undefined,
*(uninitialized variable reference error)* abnormal termination.

[HR47] where $NUMBER_1$ equals $NUMBER_2$ + $NUMBER_3$ ::= where MAXINT contains $NUMBER_2$ $NUMBER_3$,
where $NUMBER_1$ is $NUMBER_2$ $NUMBER_3$,
| where $NUMBER_2$ $NUMBER_3$ one contains MAXINT,

[HR48] where $NUMBER_1$ equals $NUMBER_2$ * $NUMBER_3$ one ::= where MAXINT contains $NUMBER_1$,
where $NUMBER_1$ is $NUMBER_4$ $NUMBER_2$,
where $NUMBER_4$ equals $NUMBER_2$ * $NUMBER_3$,
| where $NUMBER_4$ $NUMBER_2$ one contains MAXINT,
where $NUMBER_1$ equals $NUMBER_2$ * $NUMBER_3$,

[HR49] where NUMBER equals NUMBER * one ::= true,

[HR50] where EMPTY equals NUMBER * EMPTY ::= true,

[HR51] where *true* equals $BOOL_1$ + $BOOL_2$ ::= where $BOOL_1$ is *true*,
| where $BOOL_2$ is *true*,

[HR52] where *false* equals *false* + *false* ::= true,

[HR53] where *false* equals $BOOL_1$ * $BOOL_2$ ::= where $BOOL_1$ is *false*,
| where $BOOL_2$ is *false*,

[HR54] where *true* equals *true* ^ *true* ::= true,

[HR55] where *true* equals NUMBER = NUMBER ::= true,

[HR56] where *false* equals $NUMBER_1$ = $NUMBER_2$ ::= where $NUMBER_1$ differs from NUMBER,

[HR57] where *false* equals NUMBER ≠ NUMBER ::= true,

[HR58] where *true* equals NUMBER ≠ NUMBER ::= where $NUMBER_1$ differs from NUMBER,

[HR59] NOTION sequence ::= NOTION,
| NOTION sequence,
NOTION,

[HR60] NOTION pack ::= (,
NOTION,
),

[HR61] true ::= EMPTY,

[HR62] where NOTETY is NOTETY ::= true,

[HR63] where $NOTETY_1$ NOTION $NOTETY_2$ contains NOTION ::= true,

[HR64] where NOTETY $ALPHA_1$ differs from $NOTETY_2$ $ALPHA_1$ ::= where $NOTETY_1$ differs from $NOTETY_2$ ,
| where $ALPHA_1$ precedes $ALPHA_2$ in ALPHABET,
| where $ALPHA_1$ precedes $ALPHA_2$ in ALPHABET,

[HR65] where $ALPHA_1$ precedes $ALPHA_2$ in $NOTETY_1$ $ALPHA_1$ $NOTETY_2$ $ALPHA_2$ $NOTETY_3$ ::= true,

notion $TABLE_2$ in line 11, since the uniform replacement rule applies only to non-terminals with identical subscripts. Other hyper-rules in Table 3.2 insure that the symbol table corresponds exactly to the declare train of the program. In addition to serving as a symbol table, $TABLE_1$ 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 $TABLE_1$ resulting from the declare train. This is done through [HR05], which checks that no identifier is declared more than once and that the number of declared identifiers is not more than the implementation-defined maximum.

Line 04 specifies a statement train and uses the symbol table $TABLE_1$ to check the context-sensitive requirements on statements. Line 04 also contains a meta-notion STMTS, which is replaced by an abstracted form of the statement train suitable for execution. It is this abstracted form of the program that will be used to specify the semantics of the program.

Lines 06 and 07 generate the input and output files. $FILE_1$ denotes the input file and $FILE_2$ denotes the output file obtained after execution of a program. Lines 08 through 11 specify the semantics of executing STMTS, starting with the initial memory state in $TABLE_1$ and the input file $FILE_1$. The initial state of the output file is empty and this is represented by end of file. SNAPSETY, which is generated by other hyper-rules, 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. By the uniform replacement rule, the proto-notion replacing $FILE_2$ must be the same as the one in line 07, which generates the final output file.

At many stages in the application of the productions, there are checks that certain proto-notions correspond according to the rules of ASPLE. For example in

line 02. $TABLE_1$ must be derivable from the declare train, and in line 08, the sequence of memory snapshots must follow from the abstracted program STMTS. These checks are accomplished by rules in the grammar that ensure that only the allowable combinations reduce to EMPTY, i.e. nothing. Thus, a legal program and its meaning is defined by a W-grammar as a program for which there exists a derivation tree whose terminals are the written form of an ASPLE program and an appropriate pair of input and output files, and where all auxiliary non-terminals reduce to the empty string.

## 3.3 The Symbol Table

The symbol table of a W-grammar is the major vehicle for the specification of the context-sensitive requirements and semantics of ASPLE. A symbol table is a proto-notion derived from the meta-notion TABLE. In this section, we will follow in detail the derivation of a valid declare train from a symbol table. This derivation is typical of the rest of the W-grammar.

```
[MP18] TABLE    :: LOCS

[MP16] LOCS     :: LOC
                 | LOCS LOC
```

These meta-productions define a TABLE as a non-empty sequence of proto-notions derived from LOC:

```
[MP15] LOC      :: loc TAG has MODE refers BOX end
```

Here, "loc " "has " "refers " and "end" serve as delimiters to simplify the reading and make the proto-notion unambiguous. The meta-notion TAG is defined as:

```
[MP05] TAG      :: LETTER
                 | TAG LETTER

[MP06] LETTER   :: A | B | ... | Z
```

Thus, TAG produces a proto-notion that represents an identifier. As given earlier, the meta-notion MODE generates proto-notions for the mode of an identifier. The meta-notion BOX, which holds the value of an identifier, is defined as:

```
[MP14] BOX      :: VALUE
                 | undefined
                 | TAG
```

In $TABLE_1$, BOX is initially replaced by "undefined", showing that the initial value of an identifier is undefined.

The proto-notion derived from TABLE that corresponds to the declare train

*int A;*

*bool AB;*

*ref int C*

is thus

loc *A* has *ref int* refers undefined end

loc *AB* has *ref bool* refers undefined end

loc *C* has *ref ref int* refers undefined end

Substituting this proto-notion in line 2 of [HR01], we obtain the proto-notion:

dcl train of loc *A* has *ref int* refers undefined end

loc *AB* has *ref bool* refers undefined end

loc *C* has *ref ref int* refers undefined end

The first part of the hyper-rule:

```
[HR02] dcl train of LOCS LOCSETY ::= MODE,
                                     ref MODE definitions of LOCS,
                                     i,
                                     dcl train of LOCSETY,
                                   | where LOCSETY is EMPTY,
                                     MODE declarer,
                                     ref MODE definitions of LOCS,
                                     i.
```

allows the generation of the production

```
dcl train of loc A has ref int refers undefined end
              loc AB has ref bool refers undefined end
              loc C has ref ref int refers undefined end
                    ::= int,
                        ref int definitions of loc A has ref int refers
                                                        undefined end,
                        ;,
                        dcl train of loc AB has ref bool refers undefined end
                                    loc C has ref ref int refers undefined
                                                                    end,
```

Using [HR03]

```
[HR03]  MODE definitions of loc TAG has MODE refers undefined end LOCSETY
                    ::= TAG,
                        ,,
                        MODE definitions of LOCSETY,
                      | where LOCSETY is EMPTY,
                        TAG,
```

we can derive the production

```
ref int definitions of loc A has ref int refers undefined end EMPTY
                    ::= where EMPTY is EMPTY,
                        A
```

From

```
[HR02]  where NOTETY is NOTETY  ::=  true,
```

we have the production

where EMPTY is EMPTY ::= true

Using these productions we are left with " *int* A';" from [HR02] and [HR03] plus proto-notions for the remainder of the declare train. A similar technique will generate the complete declare train from the symbol table.

## 3.4 The Internal Representation of the Statement Train

To specify the semantics of ASPLE, the W-grammar also uses an internal representation of the statement train of the source program. This internal representation is a proto-notion that can be derived from the meta-notion STMTS. For example, the proto-notion:

*C* becomes *A* val,

*A* input,

deref deref *C* output,

represents the statement train of the program:

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

The correspondence between this proto-notion 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 in Section 3.3.

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

```
[HR07]  TABLE TAG becomes EXP val assignment  ::=  TABLE ref MODE TAG identifiers,
                                                   :=,
                                                   TABLE EXP MODE value.
```

is based on the written form of the statement

identifier := value

The left-side part "TAG becomes EXP val" is the internal representation of the statement. On the right side, the MODE of the identifier and the MODE of the expression value must be compatible, that is, their primitive modes must be the same and the mode of the value must contain one less *ref* than the declared mode of the identifier in the TABLE. This agreement is enforced by a production generated from:

[HR21] .TABLE MODE TAG identifier ::= TAG,
where TABLE contains loc TAG has MODE,
where MAXLENGID contains TAG;

## 3.5 The 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 proto-notion derived from SNAPSETY. This is a sequence of proto-notions derived from SNAP (meaning "snapshot") which is of the form "memory LOCS FILE FILE". As we have already seen, LOCS generates a proto-notion that records the values of the variables and was initially set up as part of $TABLE_1$. The two proto-notions derived from FILE represent the input and output files. Line 8 of [HR01] provides 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 $TABLE_2$ $FILE_1$ end of file" where $TABLE_1$ is the symbol table where 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 proto-notion substituted into line 07 of [HR01]. Line 08 of [HR01] will reduce to EMPTY only if this sequence of snapshots corresponds exactly to the execution of the proto-notion derived from STMTS.

The starting and final snapshots corresponding to the execution of the ASPLE program given in the previous section are:

memory *A* loc *A* has *ref bool* refers undefined end
loc *C* has *ref ref bool* refers undefined end
space *true* space *true* space *true* end of file
end of file

and

memory loc *A* has *ref bool* refers *true* end
loc *C* has *ref ref bool* refers end
space *true* space *true* end of file
space *true* end of file

respectively.

The execution semantics of the assignment is described by the hyper-rule:

[HR38] execute TAG becomes EXP val with $SNAP_1$ $SNAP_2$ ::= evaluate EXP from $SNAP_1$ 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
$LOCSETY_2$ $FILE_1$ $FILE_2$.

This hyper-rule 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 of expression evaluation is performed with numbers expressed in an internal form consisting of strings of the digit one. The meta-notion 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.

## 4. PRODUCTION SYSTEMS AND HOARE'S AXIOMATIC APPROACH

We next 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 of Post [P1], and later developed by Smullyan [S3], Donovan and Ledgard [D1], and again by Ledgard, which after several iterations resulted in [L3].

Production Systems are a generative grammar somewhat like BNF. The additional power of Production Systems over BNF 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 enumberable set, including the set of syntactically legal programs in a language and the translation of programs into a target language.

In addition to the use of a theoretically well-based formal system, the recent development of the Production Systems notation has been guided by principles believed important to a clear and concise notation. These principles include:

(a) The keeping of the basic notation to a minimum

(b) The introduction of abbreviations to the notation only when a substantial gain is manifest

(c) The need to isolate the context-free from the context-sensitive requirements on syntax

(d) The need to appeal to context-sensitive requirements on syntax in the definition of translation

(e) The recognition 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 will be discussed in Section 4.2.

## 4.1. Syntax

A definition of the complete ASPLE syntax, including context-sensitive requirements, is given in Table 4.1. To understand this definition, the concept of a syntactic "environment" must first be clarified. An environment is a correspondence between identifiers and modes. The environment for a program is computed by applying the function "DECLARED ENV" [PS25] 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

$\rho_1$ = A → REF INTEGER,
B → REF REF INTEGER,
C → REF REF REF INTEGER

To specify the context-sensitive requirements on ASPLE, several other functions are defined. The "DOMAIN" [PS47] of an environment $\rho$ is the list of identifiers occurring in $\rho$. For example, using $\rho_1$ above,

DOMAIN($\rho_1$) = A,B,C

The function "DECLARED MODE" [PS27] 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 declared in $\rho$. Using $\rho_1$ above,

DECLARED MODE(B:$\rho_1$) ≡ **REF INTEGER**

DECLARED MODE(A+B:$\rho_1$) ≡ **INTEGER**

The declared mode of "A+X" is $\rho_1$ is undefined (in the sense of being not derivable, since X has not been declared). A function "DECLARED PRIM MODE" [PS37] is also defined, which given an expression and an environment, yields the primitive mode obtained by dereferencing the declared mode to obtain one of the primtive modes, **INTEGER or BOOLEAN. For example:**

# Table 4.1 Production System Specifying the Complete ASPLE Syntax

*[Main Productions]*

[PS01] PROGRAM<*begin* dt ; st *end*>
← ρ ≡ DECLARED ENV(dt) &
DIFF IDLIST<DOMAIN(ρ)> .&
[*All declared identifiers must be different*]
LEGAL<st:ρ> &
[*The statement train must be legal in* ρ]
$\eta_1 \geq$ LENGTH(*).
[$\eta_1$ *is the maximum program length*]

[PS02] dt DECLARE TRAIN<$d_1$; ... ;$d_n$>.
$\eta_2 \geq$ NUM DECLARED IDS(*)
[$\eta_2$ *is the maximum number of declared ids*]

[PS03] d DECLARATION<m $i_1$, ... $i_n$>,
← dm ≡ DERIVED MODE(m) &

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

[PS05] st STM TRAIN<$s_1$; ...;$s_n$> & LEGAL<*:ρ>
← LEGAL<$s_1$:ρ> i.e. & ... & LEGAL<*:ρ>
[*A statement train is legal in* ρ *only if all statement are legal in* ρ]

[PS06] s STATEMENT<s>
← ( ASGT STM<s>
| COND STM<s>
| LOOP STM<s>
| IO STM<s> ).

[PS07] s ASGT STM<i':'=e> & LEGAL<*:ρ>
← $dm_i$ ≡ DECLARED MODE(i:ρ) &
$dm_e$ ≡ DECLARED MODE(e:ρ) &
PRIM MODE($dm_i$) = PRIM MODE($dm_e$).
[*The primitive modes of i and e in* ρ *must be identical*]
$n_i$ ≡ NUM REFS($dm_i$) &
$n_e$ ≡ NUM REFS($dm_e$) &
$n_i \leq n_e+1$.
[*The mode of i must be obtainable from the mode of e by dereferencing e*]

[PS08] s COND STM<*if* e *then* st *fi*> & LEGAL<*:ρ>
← DECLARED PRIM MODE(e:ρ) = BOOLEAN &
[*The mode of e in* ρ *must be boolean*]
LEGAL<st:ρ>.
[*st must also be legal in* ρ]

[PS09] s COND STM<*if* e *then* $st_1$ *else* $st_2$ *fi*> & LEGAL<*:ρ>
← DECLARED PRIM MODE(e:ρ) = BOOLEAN, &
LEGAL<$st_1$:ρ> & LEGAL<$st_2$:ρ>.

[PS10] s LOOP STM<*while* e *do* st *end*> & LEGAL<*:ρ>
← DECLARED PRIM MODE(e:ρ) = BOOLEAN &
LEGAL<st:ρ>.

[PS11] s IO STM<*input* i> & LEGAL<*:ρ>
← i ∈ DOMAIN(ρ).
[*i must be declared in* ρ].

[PS12] s IO STM<*output* e> & LEGAL<*:ρ>
← LEGAL<e:ρ>;

[PS13] e EXPRESSION<f> & LEGAL<*:ρ>
← LEGAL<f:ρ>.

[PS14] e EXPRESSION<e+f> & LEGAL<*:ρ>
← DECLARED PRIM MODE(e:ρ)
= DECLARED PRIM MODE(f:ρ).
[*The primitive modes of e and f in* ρ *must be identical*]

[PS15] f FACTOR<p> & LEGAL<*:ρ>
← LEGAL<p:ρ>.

[PS16] f FACTOR<p'*'f> & LEGAL<*:ρ>
← DECLARED PRIM MODE(p:ρ)
= DECLARED PRIM MODE(f:ρ).

[PS17] p PRIMARY<($e_1$=$e_2$) | ($e_1 \neq e_2$)> & LEGAL<*:ρ>
← DECLARED PRIM MODE($e_1$:ρ) = INTEGER &
DECLARED PRIM MODE($e_2$:ρ) = INTEGER.

[PS18] p PRIMARY<(e)> & LEGAL<*:ρ>
← LEGAL<e:ρ>.

[PS19] p PRIMARY<i> & LEGAL<*:ρ>
← i ∈ DOMAIN(ρ).
[*i must be declared in* ρ]

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

[PS21] int INTEGER<$d_1$... $d_n$>
← $\eta_3 \geq n$.
[$\eta_3$ *is the max length of integers*]

## Table 4.1 Continued

[PS22] i IDENTIFIER<$i_1$ .. $i_n$>
. ← $n_4 \geq n$.
[$n_4$ *is the max length of identifiers*]

[PS23] d DIGIT< *0* | *1* | ... | *9* >.
l LETTER<*A* | *B* | ... | *Z* >.

[PS24] dm DERIVED ASPLE MODE <INTEGER | BOOLEAN | REF dm>.

[*Auxiliary Functions*]

[PS25] DECLARED ENV($d_1$; ...; $d_n$)
≡ DECLARED ENV($d_1$) ','
... ','
DECLARED ENV($d_n$).

[PS26] DECLARED ENV(m $i_1$, ... ,$i_n$)
≡ $i_1$→dm, ..., $i_n$→dm
← dm ≡ DERIVED MODE(m).

[PS27] DECLARED MODE(e+f:ρ)
≡ INTEGER
← DECLARED PRIM MODE(e:ρ) = INTEGER &
DECLARED PRIM MODE(f:ρ) = INTEGER.

[PS28] DECLARED MODE(e+f:ρ)
≡ BOOLEAN
← DECLARED PRIM MODE(e:ρ) = BOOLEAN &
DECLARED PRIM MODE(f:ρ) = BOOLEAN.

[PS29] DECLARED MODE(f'*'p:ρ)
≡ DECLARED MODE(f→p:ρ).

[PS30] DECLARED MODE('('$e_1$=$e_2$')' :ρ)
≡ BOOLEAN
← DECLARED PRIM MODE($e_1$:ρ) = INTEGER &
DECLARED PRIM MODE($e_2$:ρ) = INTEGER.

[PS31] DECLARED MODE('('$e_1$≠$e_2$')':ρ)
≡ DECLARED MODE('('$e_1$=$e_2$')' :ρ).

[PS32] DECLARED MODE('('e')':ρ)
≡ DECLARED PRIM MODE(e:ρ).

[PS33] DECLARED MODE(i:ρ)
≡ dm
← i→dm ∈ ρ.
[*i→dm must occur in* ρ]

[PS34] DECLARED MODE(*true*:ρ) ≡ BOOLEAN.
[PS35] DECLARED MODE(*false*:ρ) ≡ BOOLEAN.
[PS36] DECLARED MODE(n:ρ) ≡ INTEGER.

[PS37] DECLARED PRIM MODE(e:ρ) ≡ dm'
← dm ≡ DECLARED MODE(e:ρ) &
dm' ≡ PRIM MODE(dm).

[PS38] PRIM MODE(INTEGER) ≡ INTEGER.
[PS39] PRIM MODE(BOOLEAN) ≡ BOOLEAN.
[PS40] PRIM MODE(REF dm) ≡ PRIM MODE(dm).

[PS41] DERIVED MODE(*int*) ≡ REF INTEGER.
[PS42] DERIVED MODE(*bool*) ≡ REF BOOLEAN.
[PS43] DERIVED MODE(*ref* m) ≡ REF dm
← dm ≡ DERIVED MODE(m).

[PS44] NUM REFS(INTEGER) ≡ 0.
[PS45] NUM REFS(BOOLEAN) ≡ 0.
[PS46] NUM REFS(REF dm) ≡ 1 + NUM REFS(dm).

[PS47] DOMAIN($i_1$ → $dm_1$, ... $i_n$ → $dm_n$)
≡ $i_1$, ..., $i_n$.

[*Functions for Implementation Dependent Requirements*]

[PS48] NUM DECLARED IDS($d_1$; ...; $d_n$)
≡ NUM DECLARED IDS($d_1$) +
... +
NUM DECLARED IDS($d_n$).

[PS49] NUM DECLARED IDS(m $i_1$, ..., $i_n$)
≡ n.

[PS50] LENGTH [*implementation defined function to compute the length of a program* $n_1$]

DECLARED PRIM MODE(B:$\rho_1$) = INTEGER
DECLARED PRIM MODE(A+B:$\rho_1$) = INTEGER

Similarly, the functions "PRIM MODE"[PS38] and "NUM REFS" [PS44] when applied to a mode yield the corresponding primitive mode and the number of references.

In particular, consider the production for assignment statements

```
[PS07]  ASGT STM<i':'=e>   &   LEGAL<*:ρ>
            ← dm_i ≡ DECLARED MODE(i:ρ)  &
              dm_e ≡ DECLARED MODE(e:ρ)  &
              PRIM MODE(dm_i) = PRIM MODE(dm_e)  &
              n_i  ≡ NUM REFS(dm_i)  &
              n_e  ≡ NUM REFS(dm_e)  &
              n_i  ≤ n_e + 1.
```

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

i := e

is an assignment statement, and the pair

<i':'=e : ρ>

is a member of the set LEGAL, if

i is an identifier, and
e is an expression, and
ρ is an environment, and
$dm_i$ and $dm_e$ are modes derived from ASPLE declarations, and
$n_i$ and $n_e$ are integers,

and if

$dm_i$ is obtained by applying the function "DECLARED MODE" to (i:ρ), and
$dm_e$ is obtained by applying the function "DECLARED MODE" to (e:ρ), and
the function "PRIM MODE" maps $dm_i$ and $dm_e$ into identical primitive modes, and
$n_i$ is obtained by applying the function "NUM REFS" to $dm_i$, and
$n_e$ is obtained by applying the function "NUM REFS" to $dm_e$, and
$n_i$ is less than or equal to $n_e$+1.

Here the symbol "*" in the conclusion for LEGAL is used in place of the string

i := e

being defined, and the production system variables i, e, ρ, dm, and n (possibly with subscripts) are defined in subsequent productions. The quotes on the symbol ":" are 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 4.1.

[PS01] PROGRAM<*begin* dt ; st *end*>
← ρ ≡ DECLARED ENV(dt) &
DIFF IDLIST<DOMAIN(ρ)> &
LEGAL<st:ρ> &
$\eta_1$ > LENGTH(*).

A string of the form

*begin* dt ; st *end*

is a valid program if

dt is a declare train, and
st is a statement train, and
ρ is the declared environment obtained from dt, and
the domain of ρ is a list of different identifiers, and
st is legal in ρ, and
$\eta_1$ is greater than the implementation defined length of the program.

[PS05] STM TRAIN<$s_1$; ... ;$s_n$> & LEGAL<*:ρ>
← LEGAL<$s_1$:ρ> &... & LEGAL<$s_n$:ρ>.

A sequence of statements of the form

$s_1$; ...; $s_n$

is a statement train and the statement train is legal in ρ , if

$s_1$ through $s_n$ are statements, and
$s_1$ through $s_n$ are legal in ρ.

[PS14] EXPRESSION<e+f> & LEGAL<*:ρ>
← DECLARED PRIM MODE(e:ρ) = DECLARED PRIM MODE(f:ρ).

A string of the form

e+f

is an expression and the expression is legal in $\rho$, if

e is an expression, and
f is a factor, and
the declared primitive mode of e in $\rho$ is identical to the declared primitive mode of f in $\rho$.

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

Program 1

```
begin
    int A;
    ref int B;
    ref ref int C;
        A := 100;
        B := A;
        C := B;
        input C;
        output A
end
```

Program 2

```
begin
    int A;
    int B;
    ref ref int C;
        A := 100;
        B := A;
        C := B;
        input C;
        output A
end
```

Using the productions for "DECLARED ENV" [PS25], the environments for the two programs are:

| $\rho_1$ | $\rho_2$ |
|---|---|
| A → REF INTEGER, | A → REF INTEGER, |
| B → REF REF INTEGER, | B → REF INTEGER, |
| C → REF REF REF INTEGER | C → REF REF REF INTEGER |

From the premise "LEGAL<st:$\rho$>" in the production for "PROGRAM" [PS01], the statement trains are legal only if the statement trains are legal using $\rho_1$ and $\rho_2$ respectively. Using the productions for "STM TRAIN" [PS05], each statement in a statement train is legal only if each individual statement is legal using $\rho_1$ and $\rho_2$ respectively.

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

i := e

(a) $dm_i$ is the declared mode of i in $\rho$, and
(b) $dm_e$ is the declared mode of e in $\rho$, and
(c) the primitive modes obtained from $dm_i$ and $dm_e$ are identical, and
(d) the number of references in $dm_i$ is less than or equal to one plus the number of references in $dm_e$.

For the two programs given earlier, the statement "A := 100" is legal, since both $\rho = \rho_1$ and $\rho = \rho_2$.

$$dm_i \equiv \underline{\text{DECLARED MODE}}(A:\rho) = \text{REF INTEGER}$$
$$dm_e \equiv \underline{\text{DECLARED MODE}}(100:\rho) = \text{INTEGER}$$
$$\underline{\text{PRIM MODE}}(dm_i) = \text{INTEGER} = \underline{\text{PRIM MODE}}(dm_e)$$
$$n_i \equiv \underline{\text{NUM REFS}}(dm_i) = 1$$
$$n_e \equiv \underline{\text{NUM REFS}}(dm_e) = 0$$
$$n_i \leq n_e + 1$$

On the other hand, the assignment "C := B" is legal in $\rho_1$, but not in $\rho_2$. since

| for $\rho_1$ | for $\rho_2$ |
|---|---|
| $n_i = 3$ | $n_i = 3$ |
| $n_e = 2$ | $n_e = 1$ |
| $n_i \leq n_e + 1$ | $n_i > n_e + 1$ |

The productions of Table 4.1 should now be clear to the reader. For more detail, the reader is referred to [L3].

## 4.2 Semantics

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 of legal programs into a target language that expresses the meaning of a program.* In this section, we shall use the axiomatic

*Note: Production systems could be used directly to define semantics by specifying a set of triples <program : input file : output file> for each legal program. This approach has not been tried.

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 4.2.

The axiomatic approach differs significantly from the other semantic approaches in this paper in that the approach 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 which permits assertions about the membership of objects in sets and the results of applying operations to objects, e.g. 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], no attempt is made to separate these two parts. We believe this separation to be important, for it shows the user when one can proceed automatically and when one must make "mental leaps" in the attempt to prove a program correct.

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

(1) SEM is the name of a production system function mapping legal ASPLE programs and statements into assertions that can be generated automatically.

(2) PROVABLE is a production system predicate naming a set of ordered pairs $\langle a_1 : a_2 \rangle$, where $a_1$ and $a_2$ are assertions. This predicate is true only if $a_2$ can be derived from $a_1$ by the user. The rules for deriving $a_2$ from $a_1$ are those of the predicate calculus, and are not given in this paper.

(3) a, $a_1$, $a_2$, 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 readily generated by inspection of Table 4.2.

The first production of Table 4.2 specifies the overall assertions for

Table 4.2 Production System Mapping Legal ASPLE Programs into Proof Verification Rules

*[Assertions for Programs]*

[PT01] SEM(*begin* dt ; st *end*)
≡ true {*} a"
← a ≡ DECLARED ASSERTIONS(dt) &
a' ≡ a ∧ $a_{exp}$ ∧ $a_{file}$ ∧ $f_{in}$ = β ∧ $f_{out}$ = [ ] &
ρ ≡ DECLARED ENV(dt) &
SEM STM(st:ρ) ≡ a' {st} a"
[$a_{exp}$ *are the assertions for expressions*]
[$a_{file}$ *are the assertions for the input and output files*]
[β *is the user supplied input file*]

*[Assertions for Declarations]*

[PT02] DECLARED ASSERTIONS($d_1$; ... ;$d_n$)
≡ $a_1$ ∧ ... ∧ $a_n$
← DECLARED ASSERTIONS($d_1$) ≡ $a_1$ &
... &
DECLARED ASSERTIONS($d_n$) ≡ $a_n$.

[PT03] DECLARED ASSERTIONS(m $i_1$, ... ,$i_n$)
≡ $i_1$ε dm ∧ ... ∧ $i_n$ε dm
← dm ≡ DERIVED MODE(m).

*[Assertions for Statements]*

[PT04] SEM STM($s_1$:$s_2$; ... ;$s_n$ : ρ) = $a_1$ {*} $a_{n+1}$
← PROVABLE<$a_1$:$a'_1$> & SEM STM($s_1$:ρ) ≡ $a'_1$ {$s_1$} $a_2$ &
PROVABLE<$a_2$:$a'_2$> & SEM STM($s_2$:ρ) ≡ $a'_2$ {$s_2$} $a_3$ &
... &
PROVABLE<$a_n$:$a'_n$> & SEM STM($s_n$:ρ) ≡ $a'_n$ {$s_n$} $a_{n+1}$
[*Before statements, a new assertion* $a'_i$ *may need to be created and derived from* $a_i$]

[PT05] SEM STM(i ':'=e : ρ)
≡ $a^{i}_{\dagger deref(e,n)}$ {*} a
← $dm_i$ ≡ DECLARED MODE(i:ρ) &
$dm_e$ ≡ DECLARED MODE(e:ρ) &
$n_i$ ≡ NUM REFS($dm_i$) &
$n_e$ ≡ NUM REFS($dm_e$) &
n ≡ ($n_e$ - $n_i$) + 1.
[*If the number* $n_i$ *of refs to i is 1, then* $n=n_e$, *i.e. the value of e is dereferenced once to obtain a primitive value*]

[PT06] SEM STM(*if* e *then* st *fi* : ρ)
≡ a {*} a'
← SEM STM(st:ρ) ≡ a ∧ primval(e) {st} a' &
PROVABLE<a ∧ ¬primval(e) : a'>.

[PT07] SEM STM(*if* e *then* $st_1$ *else* $st_2$ *fi* : ρ)
≡ a {*} a'
← SEM STM($st_1$) ≡ a ∧ primval(e) {$st_1$} a' &
SEM STM($st_2$) ≡ a ∧ ¬primval(e) {$st_2$} a'.

[PT08] SEM STM(*while* e *do* st *end* : ρ)
≡ a {*} $a_{inv}$ ∧ primval(e)
← PROVABLE<a : $a_{inv}$> &
SEM STM(st:ρ) ≡ $a_{inv}$ ∧ primval(e) {st} $a_{inv}$
[$a_{inv}$ *is the invariant for the loop*]

[PT09] SEM STM(*input* i : ρ)
≡ $a^{f_{in}}_{rest(f_{in}) \; \dagger first(f_{in})}$ $^{\dagger deref(i,n)}$
∧ first($f_{in}$) ε $dm_i$ {*} a.
← $dm_i$ ≡ DECLARED PRIM MODE(i:ρ) &
n ≡ NUM REFS(dm).
[*Dereferencing i by n refs must yield an identifier*]
[*The first value in* $f_{in}$ *must be contained in the set of values denoted by* $dm_i$]

[PT10] SEM STM(*output* e : ρ)
≡ $a^{f_{out}}_{cat(f_{out},\ primval(e))}$ {*} a.

[*Assertions* $a_{exp}$ *for Expressions*]

[*Assertions for primval*]

[PT11] primval(e) ε NUMBER
⊃ primval(e+f) = sum(primval(e), primval(f)).

[PT12] primval(p) ε NUMBER
⊃ primval(p*f) = product(primval(p), primval(f)).

[PT13] primval(e) ε BOOLEAN
⊃ primval(e+f) = or (primval(e), primval (f)).

[PT14] primval(p) ε BOOLEAN
⊃ primval(p*f) = and (primval(p), primval(f)).

[PT15] primval($e_1$) = primval($e_2$)
⊃ primval ('('$e_1$='$e_2$')') = true.

[PT16] primval($e_1$) ≠ primval($e_2$)
⊃ primval ('('$e_1$='$e_2$')') = false.

Table 4.2 Continued

[PT17] primval ('('$e_1$≠$e_2$')') = primval ('('$e_1$=$e_2$')').

[PT18] i ≠ tv ⊃ primval(i) = primval(v)

[PT19] primval(int) = int.

[PT20] primval(*true*) = true.

[PT21] primval(*false*) = false.

[*Assertions for dereferencing*]

[PT22] deref(v,0) = v ← (IDENTIFIER<v> | INTEGER<v> | BOOLEAN<v>).

[PT23] i = tv. ⊃ deref(i,n) = deref(v, n-1) ← (IDENTIFIER<v> | INTEGER<v> | BOOLEAN<v>).

[*Assertions for Integers*]

[PT24] 0, IMAX ∈ INTEGER.

[*IMAX is the implementation defined quantity* $\eta_5$]

[PT25] int ≠ IMAX ⊃ succ(int) ∈ INTEGER.

[PT26] int = IMAX ⊃ succ(int) = ...

[*Implementation defined value upon arithmetic overflow*]

[PT27] int ≠ 0 ⊃ pred(int) ∈ INTEGER.

[PT28] sum(int,0) = int.

[PT29] $int_1$ ≠ 0 ∧ $int_1$ ≠ IMAX ⊃ sum($int_1$, $int_2$) = sum(succ($int_1$), pred($int_2$))

...

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

[PT30] and(true, true) = true

[PT31] and(true, false) = false

...

[*The conventional axioms for Booleans*]

[*Assertions* $a_{file}$ *Input and Output Files*]

[ib ∈ INTEGER BOOLEAN]

[f ∈ FILE]

[] ∈ FILE.
cat(ib,f) ∈ FILE. ← $\eta_6$ ≥ FILELENGTH(f)

[*FILELENGTH is the implementation defined quantity for computing file lengths* $\eta_6$]

first(cat(ib,f)) = ib.
rest(cat(ib,f)) = f.
eof([]) = true.
eof(cat(ib,f) = false.

[PT01] SEM(*begin* dt ; st *end*)

≡ true {*} a"

← a ≡ DECLARED ASSERTIONS(dt) &

a' ≡ a ∧ $a_{exp}$ ∧ $a_{file}$ ∧ $f_{in}$ = β ∧ $f_{out}$ = [ ] &

ρ ≡ DECLARED ENV(dt) &

SEM STM(st:ρ) ≡ a' {st} a".

This production may be read: if

a is the assertion generated from the declare train dt, and

$a_{exp}$ is the assertion for expressions (to be discussed later), and

$a_{file}$ is the assertion for input and output files (to be discussed later), and

β is the user supplied input file, and

$f_{in}$, $f_{out}$ are files, where $f_{in}$ = β and $f_{out}$ is empty, and

a' is the assertion
$a \wedge a_{exp} \wedge a_{file} \wedge f_{in} = \beta \wedge f_{out} = [\ ]$ and

a" is the assertion obtained from the statement train, given that a' is true before the statement train.

then

$a^{\pi}$ is the assertion upon termination of the program.

The assertions generated from ASPLE declare trains [PT02] are simply the assertions of set membership for each declared identifier. For example, the declare train

*ref int* *A*;
*ref bool* *B*

yields the assertions

*A* ∈ REF REF INTEGER ∧
*B* ∈ REF REF BOOLEAN

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

(a) a proof that the assertion $a'_i$ before each statement is provable from the previous assertion $a_i$.

(b) the generation of the assertion $a_{i+1}$ upon termination of each statement

For example, the semantics of a statement train with two statements is specified by the following production obtained from [PT04];

$$\underline{\text{SEM STM}}(s_1;s_2 : \rho) = a_1 \{*\} a_3$$
$$\leftarrow \text{PROVABLE}\langle a_1 : a'_1\rangle \ \&\ \underline{\text{SEM STM}}(s_1:\rho) \equiv a'_1 \{s_1\} a_2 \ \&$$
$$\text{PROVABLE}\langle a_2 : a'_2\rangle \ \&\ \underline{\text{SEM STM}}(s_2:\rho) \equiv a'_2 \{s_2\} a_3.$$

This creation of new assertions $a'_1$ and $a'_2$ that are provable from $a_1$ and $a_2$ respectively reflects the "mental leaps" required by the user for proofs about subsequent statements.

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

$$\underline{\text{SEM STM}}(s) \equiv a_1 \{*\} a_2$$
$$\leftarrow p_1, p_2, \ldots, p_n.$$

Here $a_1$ is any assertion that is true before execution of the statement, $a_2$ is the assertion derived from $a_1$ after execution of the statement, and $p_1$ through $p_n$ are the predicates that must be true in order to generate $a_2$ from $a_1$.

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

[PT05]
$$\underline{\text{SEM STM}}(i\ ':'=\ e : \rho)$$
$$\equiv a^{i}_{\uparrow \underline{\text{deref}}(e,n)} \{*\} a$$
$$\leftarrow dm_i \equiv \underline{\text{DECLARED MODE}}(i:\rho) \ \&$$
$$dm_e \equiv \underline{\text{DECLARED MODE}}(e:\rho) \ \&$$
$$n_i \equiv \underline{\text{NUM REFS}}(dm_i) \ \&$$
$$n_e \equiv \underline{\text{NUM REFS}}(dm_e) \ \&$$
$$n \equiv (n_e - n_i) + 1.$$

This production may be read

the assertion a may be derived from the assertion $a^{i}_{\uparrow\underline{deref}(e,n)}$

if

$dm_i$ and $dm_e$ are the derived modes of i and e in ρ, and

$n_i$ and $n_e$ are the number of refs in i and e, and

n equals $(n_e - n_i) + 1$.

The up-arrow "↑" denotes a "pointer" to a value. In general, the notation $a^{i}_{v}$ denotes the assertion obtained from a by replacing all free occurrences of i by v. In the above production, v is ↑deref(e,n), i.e. a pointer to the dereferenced values of e. This value must be well-defined (i.e. not undefined), and the assertion before the assignment must be identical to $a^{i}_{\uparrow\underline{deref}(e,n)}$. In a sense the proof rule for assignment appears the wrong way around, for the assertion replacing i by a value appears before the statement. This initially counter-intuitive definition reflects two facts:

(a) the value of e must be obtained before the statement is executed.

(b) any assertions involving i before execution of the statement must be true when i is replaced by a pointer to the primitive value of e.

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

[PT08] SEM STM(*while* e *do* st *end* : ρ)
≡ a {*} $a_{inv}$ ∧ primval(e)
← PROVABLE<a : $a_{inv}$> &
SEM STM(st:ρ) ≡ $a_{inv}$ ∧ primval(e) {st} $a_{inv}$.

Here the predicate PROVABLE must be used to derive the loop invariant $a_{inv}$ from any assertion a that is true before the loop, and SEM STM(st:ρ) must be shown to not alter the truth of $a_{inv}$ when the primitive value of e is true. The

invariant $a_{inv}$ of this production is a free variable, and thus 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.

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

[PT11] primval(e) ∈ NUMBER
⊃ primval(e+f) = sum(primval(e), primval(f)).

[PT12] primval(p) ∈ NUMBER
⊃ primval(p*f) = product(primval(p), primval(f)).

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

[PT24] 0, IMAX ∈ INTEGER.

[PT25] int ≠ IMAX ⊃ succ(int) ∈ INTEGER.

[PT27] int ≠ 0 ⊃ pred(int) ∈ INTEGER.

and so forth. The number IMAX is the implementation defined maximum integer $\eta_5$.

For dereferencing identifiers we have:

[PT22] deref(v,0) = v
← (IDENTIFIER<v> | INTEGER<v> | BOOLEAN<v>).

i.e. dereferencing a value by zero refs yields the value itself, and

[PT23] i = ↑v
⊃ deref(i,n) = deref (v, n-1)
← (IDENTIFIER<v> | INTEGER<v> | BOOLEAN<v>).

i.e. dereferencing a value by n refs results in removal of n refs.

Finally, one important point. In the production system of Table 4.2, no explicit mention is made of cases where syntactically legal programs result in semantic errors. Like BNF and Production Systems in the specification of syntax, semantic errors can be deduced only by the impossibility of deriving a valid result. For example, in the semantic definition of assingment 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 derefenced value is not defined.

# 5. THE VIENNA DEFINITION LANGUAGE

One of the earliest proposals for the rigorous definition of a programming language was the use of an actual implementation [G1]. Two major objections to this technique are the inevitable encroachment of the host hardware and the restricted availability of the definition. To escape these objections, the IBM Vienna Laboratories developed the notion 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, used originally for a formal definition of PL/I [L6].

## 5.1 An Overview of VDL

A formal definition in VDL is founded on the notion of an "abstract machine" (see Figure 5.1). 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 with the senamics 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 are checked. 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.

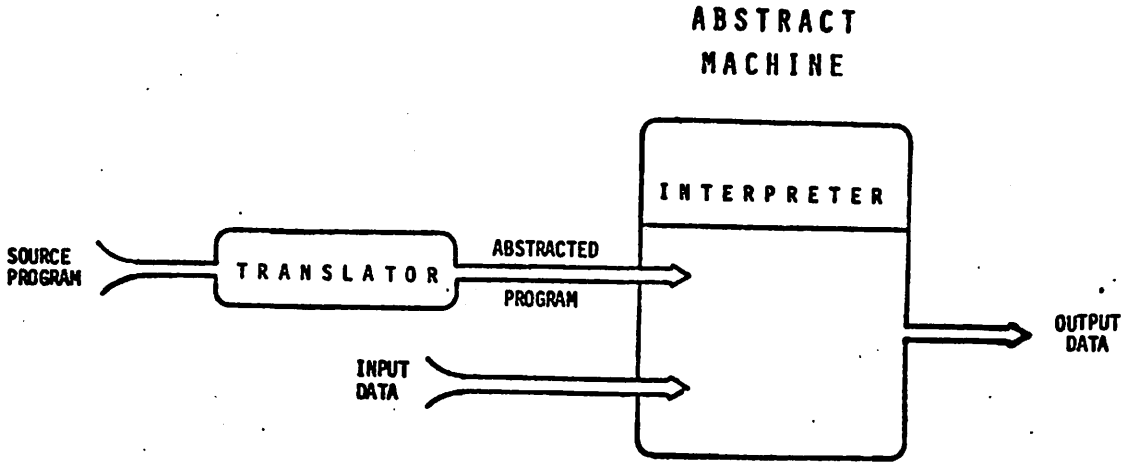


Figure 5.1 Schematic of a Programming Language Definition in VDL

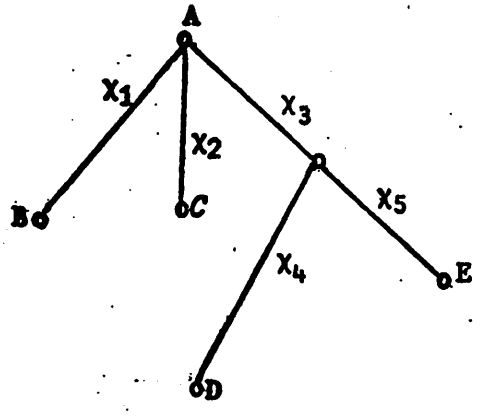


Figure 5.2 A Composite VDL Object

In VDL, both the abstract machine and the program are objects that can be represented as trees. There are two classes of objects: elementary objects, with no components, and composite objects, with a finite number of objects as immediate components.

Figure 5.2 shows a representation of a composite object named A. This object has three immediate components, each uniquely named by its selector, $\chi_1$, $\chi_2$, and $\chi_3$. We denote the immediate component $\chi_1$ of A by $\chi_1(A)$. This is the elementary object B. Similarly, we denote by $\chi_4 \circ \chi_3(A)$ the component $\chi_4$ of $\chi_3(A)$. The selector $\chi_4 \circ \chi_3$ is a composite selector. The application of a selector to an object with no selector of that name yields the null object, denoted by $\Omega$. For example, $\chi_7(A) = \Omega$ and $\chi_3 \circ \chi_2(A) = \Omega$.

The composite object $\chi_3(A)$ has two components named $\chi_4$ and $\chi_5$. The composites are the elementary objects named D and E, respectively. We define A and $\chi_3(A)$ by a list of selector-object pairs, i.e.

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

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

To specify subclasses of the class of objects, predicates are defined that are true only for members of the subclass; all such predicates have the prefix "is-". For example, is-$\Omega$(Z) will be true if and only if Z is a null object.

Objects can be modified by using the $\mu$ operator. The result of $\mu(A:\langle \chi_1 : F \rangle)$ is an object constructed from A by deleting the $\chi_1$ component, if it exists, and adding a component $\langle \chi_1 : F \rangle$. Applying $\mu(A:\langle \chi_1 : F \rangle)$ to the object A of Figure 5.2 replaces the elementary object B by the elementary object F.

A special case of this operation is the $\mu_o$ operator which constructs a new object from a set of selector-elementary-object pairs.

Objects that represent lists are often used in VDL. If L is an object that represents a list of n objects, none of them null, then the elements of L are named by the selectors, elem(i), $1 \leq i \geq n$. VDL also makes use of the elementary functions, length(L), head(L), tail(L), and the concatenation of two lists, $L_1 \cap L_2$, all with the usual meanings. By convention, objects satisfying the predicate "is-x-list" denote lists whose components satisfy the predicate "is-x".

## 5.2 The Abstract Machine

The abstract machine used to define ASPLE, the "ASPLE Machine", is specified by its machine state $\xi$, an object satisfying the predicate is-state. This predicate is defined in Table 5.1. Rule [M01] specifies that $\xi$ has five components, the abstracted program to be interpreted, (described in Section 5.3), a machine-control part, (described in Section 5.5), a storage part, and two files.

The storage part of the ASPLE Machine:

[M02] is-storage = ((<id: is-value> || is-identifier(id))

is a set of pairs of the form <id: is-value>, where the selector id satisfies the predicate is-identifier, and the value part represents an object that can be obtained by applying an identifier selector to the storage component of the Machine. From [M03]

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

an ASPLE value is either a constant or an identifier.

TABLE 5.1 DEFINITION OF THE ASPLE MACHINE STATE

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

[M02] is-storage = (([<id: is-value> || is-identifier(id)])
[*each value in storage is selected by its identifier*]

[M03] is-value = is-const ∨ is-identifier

[M04] is-const = is-boolean ∨ is-integer

[INITIAL STATE OF THE ASPLE MACHINE]

$\xi = \mu_0$(<s-program: translate(Γ)>, [*initialized by performing translate function on concrete program*]

<s-control: interpret-program>,
<s-store: Ω>,
<s-input: [*input file for program, obtained from a source outside this definition*]
<s-output: is-<>>) [*output file is initially empty*]

TABLE 5.2 DEFINITION OF PREDICATE IS-C-PROGRAM

[C01] is-c-program = (<$s_1$: is-begin>, <$s_2$: is-c-dcl-train>, <$s_3$: is-i>, <$s_4$: is-c-stm-train>, <$s_5$: is end>)
[C02] is-c-dcl-train = (<s-del: is-;>, <$s_1$: is-c-declaration>,..., <$s_{n_1}$: is-c-declaration>)
[C03] is-c-stm-train = (<s-del: is-;>, <$s_1$: is-c-statement ,..., <$s_{n_2}$: is-c-statement>)
[C04] is-c-declarations = (<$s_1$: is-c-mode>, <$s_2$: is-c-idlist>)
[C05] is-c-statement = is-c-asgt-stm ∨ is-c-cond-stm ∨ is-c-loop-stm ∨ is-c-input-stm ∨ is-c-output-stm
[C06] is-c-mode = (<$s_1$: is-Ω ∨ (<$s_1$: is-ref>,..., s<$s_{n_3}$: is-ref>)>, <$s_2$: is-bool ∨ is-int>)
[C07] is-c-idlist = (<s-del: is-,>, <$s_1$: is-c-id>,..., <$s_{n_4}$: is-c-id>)
[C08] is-c-asgt-stm = (<$s_1$: is-c-id>, <$s_2$: is-:=>, <$s_3$: is-c-exp>)
[C09] is-c-cond-stm = (<$s_1$: is-if>, <$s_2$: is-c-exp>, <$s_3$: is-then>, <$s_4$: is-c-stm-train>,
<$s_5$: is-Ω ∨ is-c-else-part>, <$s_6$: is-fi>)
[C10] is-c-loop-stm = (<$s_1$: is-while>, <$s_2$: is-c-exp>, <$s_3$: is-do>, <$s_4$: is-c-stm-train>, <$s_5$: is-end>)
[C11] is-c-input-stm = (<$s_1$: is-input>, <$s_2$: is-c-id>)
[C12] is-c-output-stm = (<$s_1$: is-output>, <$s_2$: is-c-exp>)
[C13] is-c-else-part = (<$s_1$: is-else>, <$s_2$: is-c-stm-train>)
[C14] is-c-exp = is-c-factor ∨ (<$s_1$: is-c-exp>, <$s_2$: is-+>, <$s_3$: is-c-factor>)
[C15] is-c-factor = is-c-primary ∨ (<$s_1$: is-c-factor>, <$s_2$: is-*>, <$s_3$: is-c-primary>)
[C16] is-c-primary = is-c-id ∨ is-c-bool-const ∨ is-c-int-const ∨
(<$s_1$: is-(>, <$s_2$: is-c-exp ∨ is-c-compare>, <$s_3$: is-)>)
[C17] is-c-compare = (<$s_1$: is-c-exp>, <$s_2$: is-= ∨ is-≠>, <$s_3$: is-c-exp>)
[C18] is-c-bool-const = is-true ∨ is-false
[C19] is-c-int-const = (<$s_1$: is-c-digit>,..., <$s_{n_5}$: is-c-digit>)
[C20] is-c-id = (<$s_1$: is-c-letter>,..., <$s_{n_6}$: is-c-letter>)
[C21] is-c-digit = is-0 ∨ is-1 ∨...∨ is-9
[C22] is-c-letter = is-A ∨ is-B ∨...∨ is-Z

The input and output files are lists of objects satisfying the predicate is-const and are, therefore, lists of boolean and integer values.

The program part of the ASPLE Machine is an abstracted ASPLE program, described in Section 5.3. The program part is initialized by performing the function "translate" on an original source program; the Translator is described in Section 5.4. The control part of the ASPLE Machine is initialized to the machine operation "interpret-program", which is defined in Section 5.5. The ASPLE Machine's store is initially empty, reflecting the ASPLE rule that the value of all variables in initially empty, reflecting the ASPLE rule that the value of all variables is initially undefined. The input file is initialized to the input data for the program and the output file, to an empty list.

## 5.3 The 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 5.2. This definition is derived directly from the context-free syntax of ASPLE shown in Table 2.1. 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, $s_1$, $s_2$, ..., assumed to be mutually distinguishable. They are used to construct objects whose structure is similar to VDL lists, except that some of the components may be null.

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 2.1:

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

with definition [C01] of Table 5.2:

[C01] is-c-program = (<$s_1$: is-*begin*>, <$s_2$: is-c-dcl-train>, <$s_3$: is-;>, <$s_4$: is-c-stm-train>, <$s_5$: is-*end*>)

C1 specifies that an object satisfying is-c-program has five immediate components, $s_1$, ... ,$s_5$. The first component is an elementary object satisfying the predicate s-*begin* and corresponds to the *begin* symbol in the character-string representation. The second component is an object satisfying is-c-dcl-train, defined in production [C02]:

A declare train consists of a sequence of declarations separated by semicolons. This production shows the VDL convention for representing a sequence of items separated by a delimiter. The selector s-del selects an object representing the delimiter, and the names $s_1$, $s_2$ ... select the successive items of the sequence.

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-program" defined in Table 5.3. The definition of the elementary objects has been left somewhat informal, indicated by the use of italic type.

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 required for execution.

TABLE 5.3 DEFINITION OF THE PREDICATE IS-PROGRAM

| | | |
|---|---|---|
| [A01] | is-program | = is-statement-list |
| [A02] | is-statement | = is-assignment ∨ is-conditional ∨ is-loop ∨ is-input ∨ is-output |
| [A03] | is-assignment | = (<target: is- identifier>, <source: is-expr>) |
| [A04] | is-conditional | = (<condition: is-expr>, <true-part: is-statement-list>, <false-part: is-statement-list>) |
| [A05] | is-loop | = (<condition: is-expr>, <body: is-statement-list>) |
| [A06] | is-input | = (<target: is-loc>, <mode: is-mode>) |
| [A07] | is-output | = (<source: is-expr>) |
| [A08] | is-loc | = (<name: is-identifier>, <deref: is-integer>) |
| [A09] | is-expr | = is-value ∨ is-operation |
| [A10] | is-value | = is-boolean ∨ is-integer ∨ is-loc |
| [A11] | is-operation | = (<op1: is-expr>, <op2: is-expr>, <operator: is-operator>) |
| [A12] | is-mode | = *a set of two elementary objects represented by* {*int*, *bool*} |
| [A13] | is-identifier | = *an infinite set of distinguishable elementary objects* |
| [A14] | is-integer | = *an infinite set of elementary objects denoting the integer values. The subset that denote the integer values less than 10 are represented by* {*0*, *1*, ..., *9*} |
| [A15] | is-boolean | = *a set of two elementary objects denoting truth values and represented by* {*true*, *false*} |
| [A16] | is-operator | = *a set of elementary objects represented by* {*plus*, *mult*, *or*, *and*, *equal*, *notequal*} |

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

Abstracted ASPLE programs are simpler than the corresponding concrete programs. There are no declarations, and the only explicit type information is contained in the representation of the input statement where it is needed to check that the type of the input value matches the type of the target value. There is, however, some implicit type information contained in the representation of operators, for example, the "+" of the original program has been translated into *plus* or *or* according to the operand type. This is very similar to the situation in complied machine code where there is implicit type information contained in the operation codes.

## 5.4 The 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 as a set of functions, many of them recursive, shown in Table 5.4.

Generally, the functions consist of conditional expressions of the form

$$p_1 \rightarrow e_1, p_2 \rightarrow e_2, \ldots, p_n \rightarrow e_n$$

where $p_i$ is a predicate expression and $e_i$ is an expression defining the action to be taken. The value of this conditional is the value of the first evaluated expression $e_i$ where $p_i$ 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) ≤ $n_1$ → $\mu(\xi_0.<$-program: trans-program(t)>)
true → error

TABLE 5.4 THE ASPLE TRANSLATOR

```
[T01] translate(t)=
        program-length(t) ≤ n1  →  μ(t*,< -program:  trans-program(t)>)
        true                    →  error [program too long]

[T02] trans-program(t)=
        number-of-identifiers(s2(t))<n2  →  trans-stm-train(s4(t))
        true                             →  error [too many variables declared]

        [where is-c-dcl-train(s2(t)) and is-c-stm-train(s4(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            →  μ0(<elem(i): trans-stmt(s1(t))> || 1 ≤ i ≤ slength(t)))

        [where is-c-statement(s1(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, otherwise the righthand side is
                                 a reference and the appropriate amount of de-referencing must be
                                 calculated]
        ref-chain-length(s1(t)) = 1  →  μ0(<target: make-id(s1(t))>, <source: trans-expr(s3(t))>)
        true                         →  μ0(<target: make-id(s1(t))>, <source: trans-ref(s3(t),
                                          ref-chain-length(s1(t))-1)>)

      [where: is-c-id(s1(t)), and is-c-exp(s3(t))]

[T07]  trans-cond-stm(t)=
         primitive-mode(s2(t)) = bool  →
                  μ0(<condition: trans-expr(s2(t))>, <true-part: trans-stmt-train(s4(t))>,
                                                     <false-part: trans-stmt-train(s2.s5(t))>)
         true                          →  error  [mode of conditional expression not boolean]

       [where: is-c-exp(s2(t)), is-c-stm-train(s4(t)), and is-c-stm-train(s2.s5(t))]

[T08]  trans-loop-stm(t)=

         primitive-mode(s2(t)) = bool  →  μ0(<condition: trans-expr(s2(t))>, <body: trans-stm-train(s4(t))>)
         true                          →  error  [mode of conditional expression not boolean]

       [where: is-c-exp(s2(t)), and is-c-stm-train(s4(t))]

[T09]  trans-input-stm(t)=
                μ0(<target: trans-ref(s2(t),1)>, <mode: primitive-mode(s2(t))>)

       [where: is-c-id(s2(t))]

[T10]  trans-output-stm(t)=
         valid-expr(t)  →  μ0(<source: trans-expr(s2(t))>)
         true           →  error  [invalid expression]

       [where: is-c-expr(s2(t))]

[T11]  trans-expr(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-expr     →  trans-expr(s2(t))   [t is a parenthesized expression]
         true                        →  μ0(<op1: trans-expr(s1(t))>, <op2: trans-expr(s3(t))>,
                                                                     <operator: make-operator(t)>)
                                          [if it is not a constant, identifier, or parenthesized
                                           expression then it 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  →  [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
                             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) ≤ $n_2$ → value-of-int-const(t)
true → error [*integer constant too big for implementation*]

[T16] value-of-int-constant(t)=
is-$\underline{0}$(t) → $\underline{0}$
is $\underline{1}$(t) → $\underline{1}$
⋮
is-$\underline{9}$(t) → $\underline{9}$

slength(t) < $n_3$ → $\sum_{i=1}^{\text{slength}(t)}$ value-of-int-const($s_i$(t)) · 10 ↑ (slength(t)-i)
true → [*too many digits in integer constant*]

[*where:* is-c-digit($s_i$(t)), 1 ≤ i ≤ slength(t)]

[T17] make-operator(t)=

primitive-mode($s_1$(t)) = $\underline{bool}$ & is-$\underline{+}$($s_2$(t)) → $\underline{or}$
primitive-mode($s_1$(t)) = $\underline{bool}$ & is-$\underline{*}$($s_2$(t)) → $\underline{and}$
primitive-mode($s_1$(t)) = $\underline{int}$ & is-$\underline{+}$($s_2$(t)) → $\underline{plus}$
primitive-mode($s_1$(t)) = $\underline{int}$ & is-$\underline{*}$($s_2$(t)) → $\underline{mult}$
primitive-mode($s_1$(t)) = $\underline{int}$ & is-$\underline{=}$($s_2$(t)) → $\underline{equal}$
primitive-mode($s_1$(t)) = $\underline{int}$ & is-$\underline{\neq}$($s_2$(t)) → $\underline{notequal}$

[*where:* is-c-exp($s_1$(t)),]

[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 → $\underline{bool}$
is-c-int-const → $\underline{int}$
is-c-parenthesized-expression(t) → primitive-mode($s_2$(t))
valid-compare(t) → $\underline{bool}$
valid-expr(t) → primitive-mode($s_1$(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($s_1$•mode-of-id(t))+1 [*this is an elementary object satisfying is-integer*]
true → $\underline{1}$

[*where:* $s_1$•mode-of-id(t) *is the list of ref's in the declaration of the identifier* t]

[T20] primitive-mode-of-id(t)=
is-$\underline{bool}$($s_2$(mode-of-id(t))) → $\underline{bool}$
is-$\underline{int}$($s_2$(mode-of-id(t))) → $\underline{int}$

[T21] mode-of-id(t)= [*find declaration that contains identifier equal to t and select mode part of declaration*]
(∃X)(X•$s_2$(PROG)=t) → $s_1$•((^X)(is-c-declaration(X(PROG))&(∃1)($s_1$•$s_2$•X(PROG)=t)))(PROG)
true → error [*identifier was not declared*]

[*where:* is-c-dcl-train($s_2$(PROG))]

[T22] program-length(t)=
¬is-slist(t) → $\underline{1}$

slength(t)
true → $\sum_{i=1}$ program-length($s_i$(t))

[T23] number-of-identifiers(t)=

slength(t)
valid-declare-train(t) → $\sum_{i=1}$ slength($s_2$•$s_1$(t))
true → error[*duplicate declarations in declare train*]

[*where:* is-c-idlist($s_2$•$s_1$(t)), 1 ≤ i ≤ slength(t)]

[T24] valid-declare-train(t)=
¬(∃$X_1$,$X_2$)($X_1$≠$X_2$ & is-c-id($X_1$(t)) & is-c-id($X_2$(t)) & $X_1$(t) = $X_2$(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-modes-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))]

[T26] valid-compare(t)=
is-c-compare(t) & primitive-mode($s_1$(t)) = $\underline{int}$ & primitive-mode($s_3$(t)) = $\underline{int}$

[*where:* is-c-exp($s_1$(t)) & is-c-exp($s_3$(t))]

[T27] valid-expr(t)=
¬is-=($s_2$(t)) & ¬is-≠($s_2$(t)) & (primitive mode($s_1$(t)) = primitive-mode($s_3$(t)))

[*where:* is-c-expr($s_1$(t)) & is-c-expr($s_2$(t))]

This rule has a single parameter, t, corresponding to an object satisfying "is-c-program."

The function translate checks that the length of the program, calculated by the operation program-length, is less than the implementation defined limit, $\eta_1$. If this condition is satisfied, the machine-state $\xi$ is constructed and the result of evaluating trans-program(t) is attached to $\xi$ by the selector s-program. 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.

A validity check often makes use of a predicate, for example, the predicate "valid-modes-for-assignment" defined in [T25]:

[T25] valid-modes-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)))

This predicate defines the rule needed for compatible modes in assignment, i.e., the primitve 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 T18 and T19, respectively.

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

## 5.5 The VDL Interpreter

In the previous sections we saw the construction of the abstracted program and its attachment as part of the initial machine state $\xi_0$ of the abstract machine. The control part of $\xi_0$ contains the operation interpret-program. Execution of this operation begins the interpretation of the abstracted program, which continues until the control part becomes empty or until an error is detected.

Informally, the control part of $\xi$ can be visualized as a stack of machine operations, some with arguments. The operation on the top of the stack is the next instruction to be executed and when execution is complete, it is removed from the stack. The execution of an operation causes one of the following:

a. The addition of new operations to the instruction stack.

b. 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 $\xi$.

The machine operations of the ASPLE Machine are defined in Table 5.5. The "interpret-program" operation is defined in I01.

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

Its effect is to cause the operation "interpret-statement-list" to be put on the operation stack with the abstracted program from $\xi$ as argument. The abstracted program, defined in Table 5.3, consists of a statement list.

The operation "interpret-statement" is defined in I02:

```
[I02] interpret-statement-list(t) =
          is-<>                    → Ω
          true                     → interpret-statement-list(tail(t));
                                       interpret-statement(head(t))
```

TABLE 5.5 DEFINITION OF THE ASPLE INTERPRETER

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

[I02]  interpret-statement-list(t) =   [defines interpretation sequence of statements in program]
           is-<>                        → Ω          [if t is empty list, do nothing]
           true                         → interpret-statement-list(tail(t));
                                              interpret-statement(head(t))

[I03]  interpret-statement(t)=
           is-assignment(t)             → interpret-assignment(t)
           is-conditional(t)            → interpret-conditional(t)
           is-loop(t)                   → interpret-loop(t)
           is-input(t)                  → interpret-input(t)
           is-output(t)                 → interpret-output(t)

[I04]  interpret-assignment(t)=
           assign(target(t), val):      [evaluate the right side then pass value to assign operation]
               val:  eval-expr(source(t))

       [where: is-identifier(target(t)) and is-expr(source(t))]

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

       [where: is-statement-list(true-part(t)), is-statement-list(false-part(t)), and is expression(cond(t))]

[I06]  interpret-loop(t)=
           eval-expr(cond(t)) = true    → interpret-loop(t);
                                              interpret-statement-list(body(t));
           true                         → Ω

       [where: is-expression(cond(t)), and is-statement-list(body(t))]

[I07]  interpret-input(t)=
           assign(destination, val);
              destination: eval-ref(name-target(t), derefotarget(t));
              val: read(mode(t))

       [where: is-loc(target(t)), is mode-(mode(t))]

[I08]  interpret-output(t)=
                  write(value);
                      value: eval-expr(source(t))

[I09]  eval-expr(t)=
           is-loc(t)                    → eval-ref(name(t), deref(t))
           is-value(t)                  → PASS: t
           is-infix-op(t)               → operate(val1, val2, operator(t));
                                                 val1:  eval-expr(op1(t));
                                                 val2:  eval-expr(op2(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 5.5 Continued

[I11 ] add(a,b)=
a + b < [*implementation-defined maximum*]$n_5$ → PASS: a + b [*is-integer(a+b)*]
true → PASS: implementation defined result

*for:* is-integer(a) *and* is-integer(b)

[I12 ] multiply(a,b)=
a . b < [*implementation defined maximum*]$n_5$ → PASS: a . b [*is-integer(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: *true*
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(s-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•s-store(ξ)) → PASS: id•s-store(ξ)[*obtain value of variable id from store*]
true → error [*reference to value that has not been set*]

[I20 ] write(v)=
length(s-output(ξ)) < [*an implementation defined maximum*]$n_6$ → μ(ξ,<s-output: s-output(ξ)^v>)
[*concatenate value v on end of output file*]
→ error
true [*number of items on output file greater then implementation defined maximum*]

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

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

and uses the same type of conditional expression as used in the translator. If the statement list t is empty, the instruction being executed is replaced in the operation stack by nothing, Ω. It is in this 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))
```

is put on the operation stack, with the "interpret-statement" operation on the top for execution next. The operation's argument is the first statement from the statement-list t, making this statement the next ASPLE statement to be interpreted. When this is completed, the operation "interpret-statement-list" will become the top operation in the stack and 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 the "interpret-assignment" operation defined in I04.

[I04]
```
interpret-assignment(t)=
    assign(target(t), val);
        val:  eval-expr(source(t))
```

As before, the "interpret-assignment" operation is replaced in the stack by:

```
assign(target(t), val);
    val:  eval-expr(source(t))
```

The "val:" part denotes that the execution of eval-expr will return a value, to be known locally in "interpret-assignment" operations as "val," and that this value will be substituted into the argument list of an as yet unexecuted instruction. The value replaces the argument denoted by "val" in the assign operation. In this way, the value computed by the eval-expr operation is passed to the assign operation for assignment to storage.

The definition of "eval-expr":

```
[I09]  eval-expr(t)=
           is-loc(t)          → eval-ref(name(t), deref(t))
           is-value(t)        → PASS: t
           is-infix-op(t)     → operate(val1, val2, operator(t));
                                    val1:  eval-expr(op1(t));
                                    val2:  eval-expr(op2(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 is a reference then a single new operator is put on the stack; if it is an operator with two operands then three new operations are added to the stack. However, if it is a value, i.e., it corresponds to a constant in the original program, the actual value is returned. This is signified by the word "PASS:" followed by the value to be returned.

Value returning operations can also make changes to other parts of $\xi$ through the use of the $\mu$ operator. For example, the "read" operation defined in [I21]:

```
[I21]  read(t)=
           is-<>(s-input(ξ))                    → error
           mode-of-const(head(s-input(ξ))=t     → μ(ξ; s-input: tail(s-input(ξ))>)
                                                  PASS:  head(s-input(ξ))
           true                                 → error
```

first checks that the end of file has not been reached. If it has, this is an error and interpretation stops at that 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 first value in the input file is deleted by the $\mu$ operator and this value is returned with the PASS: mechanism.

The definition of the interpreter instructions specifies the meaning of every legal construct of ASPLE. By making a clear separation between the translator and interpreter, it is possible to show the difference between the static and dynamic aspects of the language.

# 6. ATTRIBUTE GRAMMARS

We next discuss the definition technique of attribute grammars originially 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].

## 6.1 Overview

The context-free grammar of a language defines a derivation tree for each syntactically correct program of the language. The attribute grammar is based on this context-free grammar and associates attributes with the nodes of the derivation tree. The attribute grammar also specifies rules for deriving values for these attributes. Each attribute may take values from a set of possible values.

These are two kinds of attribute: inherited attributes whose values are obtained from the immediate parent node in the derivation tree and synthesized attributes whose values are obtained from the immediate descendants in the tree. In a production of the grammar, the inherited attributes of the left side and the synthesized attributes of the right side represent values furnished by the surrounding nodes in the derivation tree. The other attributes, i.e. the inherited attributes of the right side and the synthesized attributes of the left, must be computed according to the evaluation rules of the given production. These attributes represent values that are passed to the surrounding nodes.

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

Attribute evaluation rules other than simple value transfers are written using action symbols. Whereas the input symbols in the production rules of the grammar determine the form of the syntactically correct written ASPLE programs, the action

symbols represent the translation of a program as a sequence of actions. This corresponds to the practice of implementing programs in two phases: compilation followed by execution. Therefore, the meaning of a source program is specified in terms of the sequence of action symbols and values obtained in the translation of the source text of a program.

A full definition of a language using attribute grammars must be supplemented with a definition of the action symbols. Thus the attribute grammar approach is not a complete method for full formal definitions. Rather than choose a rigidly defined set of actions (for example, a particular machine language) we have, as is customary with attribute grammars, left the meaning of the action symbols informally defined.

## 6.2 The Attribute Grammar for ASPLE

The attribute grammar for ASPLE is shown in Table 6.1

The production for <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 < $n_2$
condition: prog-length < $n_1$

Here the written terminals of the language are shown in italic characters. In [AG01] these are: *begin*, ; and *end*. Associated with the syntactic category <program> is the attribute ↑memory, whose value represents the state of the program variables, each of which are initialized to an undefined value. The upward arrow indicates that it is a synthesized attribute. There are three inherited and three synthesized attributes associated with <dcl train>. The attributes associated with any category are always written in the same order. The three inherited attributes are prefixed by a downward arrow.

Attribute evaluation rules that are simple value transfers are specified by the use of identical names. Thus the value of memory is obtained from the descendants of <dcl train> and passed to <program>. The value of the attribute env

TABLE 6.1 ATTRIBUTE GRAMMAR OF ASPLE

[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* $n_2$]
condition: prog-length < $n_1$
[*prog-length is an implementation defined attribute whose evaluation rules must be added to the grammar*]

[AG02] <dcl train> ↓$env_1$ ↓$num\text{-}ids_1$ ↓$memory_1$ ↓$env_2$ ↓$num\text{-}ids_2$ ↓$memory_2$
::= <declaration> ↓$env_1$ ↓$num\text{-}ids_1$ ↓$memory_1$ ↓$env_2$ ↓$num\text{-}ids_2$ ↓$memory_2$
| <declaration> ↓$env_1$ ↓$num\text{-}ids_1$ ↓$memory_1$ ↓$env_3$ ↓$num\text{-}ids_3$ ↓$memory_3$
;
<dcl train> ↓$env_3$ ↓$num\text{-}ids_3$ ↓$memory_3$ ↓$env_2$ ↓$num\text{-}ids_2$ ↓$memory_2$

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

[AG04] <declaration> ↓$env_1$ ↓$num\text{-}ids_1$ ↓$memory_1$ ↓$env_2$ ↓$num\text{-}ids_2$ ↓$memory_2$
::= <mode> ↓prim-mode ↓refs
<id-list> ↓$env_1$ ↓$num\text{-}ids_1$ ↓prim-mode ↓refs ↓memory ↓$env_2$ ↓$num\text{-}ids_2$ ↓$memory_2$

[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 ↓$refs_2$ ↓$refs_1$

[AG06] <id-list> ↓$env_1$ ↓$num\text{-}ids_1$ ↓prim-mode ↓refs ↓$memory_1$ ↓$env_2$ ↓$num\text{-}ids_2$ ↓$memory_2$
::= <declared id> ↓$env_1$ ↓prim-mode ↓refs ↓$memory_1$ ↓$env_2$ ↓$memory_2$
add one id ↓$num\text{-}ids_1$ ↓$num\text{-}ids_2$
| <declared id> ↓$env_1$ ↓prim-mode ↓refs ↓$memory_1$ ↓$env_3$ ↓$memory_3$
add one id ↓$num\text{-}ids_1$ ↓$num\text{-}ids_3$
<id-list> ↓$env_3$ ↓$num\text{-}ids_3$ ↓prim-mode ↓refs ↓$memory_3$ ↓$env_2$ ↓$num\text{-}ids_2$ ↓$memory_2$

[AG07] <declared id> ↓$env_1$ ↓$prim\text{-}mode_1$ ↓$refs_1$ ↓$memory_1$ ↓$env_2$ ↓$memory_2$
::= <id> ↓$name_1$
insert declaration ↓$env_1$ ↓$name_1$ ↓$prim\text{-}mode_1$ ↓$refs_1$ ↓$env_2$
include variable ↓$memory_1$ ↓$name_1$ ↓$memory_2$
[*the name is added to memory and its value initialized to undefined*]
condition: ∄ (t)(t=($name_2$, $prim\text{-}mode_2$, $refs_2$) & t < $env_1$ & $name_2$=$name_1$)
[*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

[AG12]  <input stm> ↓env                ::= input
                                            <used id> ↓env ↑prim-mode ↑refs ↑name1
                                            <dereference> ↓refs ↓name ↓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-mode$_1$ ↑VALUE$_1$

::= <factor> ↓env ↓refs ↑prim-mode$_1$ ↑VALUE$_1$

| <exp> ↓env ↓*zero-refs* ↑prim-mode$_1$ ↑VALUE$_2$
+
<factor> ↓env ↓*zero-refs* ↑prim-mode$_2$ ↑VALUE$_3$
<+ action> ↑prim-mode$_1$ ↑VALUE$_2$ ↑VALUE$_3$ ↑VALUE$_1$
condition: prim-mode$_1$ = prim-mode$_2$
[*primitive modes must correspond*]
condition: refs = *zero-refs*
[*the mode of the expression is without any references*]

[AG17] <+ action> ↓prim-mode ↑VALUE$_1$ ↓VALUE$_2$ ↓VALUE$_3$

::= ADD ↓VALUE$_1$ ↓VALUE$_2$ ↑VALUE$_3$
condition: prim-mode = *int*

| OR ↓VALUE$_1$ ↓VALUE$_2$ ↑VALUE$_3$
condition: prim-mode = *bool*

[AG18] <factor> ↓env ↓refs ↑prim-mode$_1$ ↑VALUE$_1$

::= <primary> ↓env ↓refs ↑prim-mode$_1$ ↑VALUE$_1$

| <factor> ↓env ↓*zero-refs* ↑prim-mode$_1$ ↑VALUE$_2$
*
<primary> ↓env ↓*zero-refs* ↑prim-mode$_2$ ↑VALUE$_3$
<* action> ↑prim-mode$_1$ ↑VALUE$_2$ ↑VALUE$_3$ ↑VALUE$_1$
condition: prim-mode$_1$ = prim-mode$_2$
[*primitive modes must correspond*]
condition: refs = *zero-refs*
[*the mode of the factor is without any references*]

[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) ∈ 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*]
subtract one ref ↓refs$_1$ ↑refs$_3$
<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$_2$*]

TABLE 6.1 Continued

[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*

[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> + *zero-refs* +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$_1$ +num-digits$_2$ +value$_1$
::= <digit> +value$_1$
add one digit +num-digits$_1$ +num-digits$_2$

| <number> +num-digits$_1$ +num-digits$_3$ +value$_2$
<digit> +value$_3$
multiply +value$_2$ +*10* +value$_4$
add +value$_4$ +value$_3$ +value$_1$
add one digit +num-digits$_3$ +num-digits$_2$

[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> +*zero-letters* +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> +nb-letters$_1$ +num-letters$_2$ +name$_1$
::= <letter> +name$_1$
add one letter +num-letters$_1$ +num-letters$_2$

| <identifier> +num-letters$_1$ +num-letters$_3$ +name$_2$
<letter> +name$_3$
concatenate +name$_2$ +name$_3$ +name$_1$
add one letter +num-letters$_3$ +num-letters$_2$

[AG31] <letter> +name ::= *A*
give value to attribute + A +name

| *B*
give value to attribute + B +name
. . .

| *Z*
give value to attribute + Z +name

represents the environment of the program, a set of triples associating identifiers, primitive modes and lengths of the reference chain. The value of env is obtained from <dcl train> and is transferred to <stm train> where it has become an inherited attribute.

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 quantity $n_2$. The attribute prog-length, which serves to represent the length of the program, is not associated with any category since its computation is left for implementation definition.

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

[AG02] <dcl train> $\uparrow env_1 \uparrow num\text{-}ids_1 \uparrow memory_1 \uparrow env_2 \uparrow num\text{-}ids_2 \uparrow memory_2$

::= <declaration> $\uparrow env_1 \uparrow num\text{-}ids_1 \uparrow memory_1 \uparrow env_2 \uparrow num\text{-}ids_2 \uparrow memory_2$

| <declaration> $\uparrow env_1 \uparrow num\text{-}ids_1 \uparrow memory_1 \uparrow env_3 \uparrow num\text{-}ids_3 \uparrow memory_3$

;

<dcl train> $\uparrow env_3 \uparrow num\text{-}ids_3 \uparrow memory_3 \uparrow env_2 \uparrow num\text{-}ids_2 \uparrow memory_2$

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

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 of $env_1$ is inherited by <dcl train> on the left side of the production.
2. This value is passed down to <declaration> and the attribute value of $env_3$ is obtained.
3. The value of $env_3$ is then passed down to <dcl train> on the right side of the production to obtain the value of $env_2$.
4. The value of $env_2$ is then passed us as a synthesized attribute of <dcl train> on the left side of the production.

## 6.3 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 action and the attribute values are defined informally in Table 6.2.

A simple example of the use of action symbols is shown in [AG05]:

```
[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
```

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 give value to attribute, which denotes a function that takes a value and returns an attribute with that initial value. The attribute ref represents the length of a variable's reference chain. The action symbol

give value to attribute ↓*one-ref* ↑$refs_1$

defines the value of the attribute $ref_1$ to be a reference chain of length 1. In the third alternative, the action symbol

add one ref ↓$refs_2$ ↑$refs_1$

defines the length of the reference chain represented by $refs_1$ to be one greater than that represented by $refs_2$.

An example of the use of action symbols and the value passing mechanism is shown in Figure 6.1. This diagram depicts the sequence of evaluations for the env attribute in the derivation tree for the ASPLE program

```
begin
    int A, B;
    . . .
end
```

TABLE 6.2 DEFINITION OF ATTRIBUTES AND ACTION SYMBOLS

[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:*

*add one ref* $+refs_1$ $+refs_2$

*implies* $refs_2 = refs_1 + 1$

*subtract one ref* $+refs_1$ $+refs_2$

*implies* $refs_2 = refs_1 - 1$

[AT03] name [*name of variable*]

*values:* *arbitrary length character strings*

*constants:* A, B, ... ,Z

*action symbols:*

*concatenate* $+name_1$ $+name_2$ $+name_3$

*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* = $\emptyset$

*action symbol:*

*insert declaration* $+env_1$ +name +prim-mode +refs $+env_2$

*implies* $env_2 = env_1 \cup \{(name, prim\text{-}mode, refs)\}$

[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:*

*or* $boolean_1$ $boolean_2$ $boolean_3$

*implies* $boolean_3 = boolean_1 \vee boolean_2$

*and* $boolean_1$ $boolean_2$ $boolean_3$

*implies* $boolean_3 = boolean_1 \wedge boolean_2$

TABLE 6.2 Continued

[AT08] integral

*values:* *positive integers*

*constants:* *0, 1, ... ,9,10*

$n_5$ = *implementation defined maximum value for integral values*

*action symbols:*

*add* $\uparrow integral_1$ $\uparrow integral_2$ $\uparrow integral_3$

*implies if* $integral_1 + integral_2 < n_5$

*then* $integral_3 = integral_1 + integral_2$

*otherwise implementation defined result*

*multiply* $\uparrow integral_1$ $\uparrow integral_2$ $\uparrow integral_3$

*implies if* $integral_1 \times integral_2 < n_5$

*then* $integral_3 = integral_1 \times integral_2$

*otherwise implementation defined result*

*compare equal* $\uparrow integral_1$ $\uparrow integral_2$ $\uparrow$boolean

*implies if* $integral_1 = integral_2$

*then* boolean = *true*

*otherwise* boolean = *false*

*compare not equal* $\uparrow integral_1$ $\uparrow integral_2$ $\uparrow$boolean

*implies if* $integral_1 = integral_2$

*then* boolean = *false*

*otherwise* boolean = *true*

[AT09] memory [*memory state*]

*values:* *sets of pairs of the form* (name, stored-value)

*constant:* *empty-memory* = $\emptyset$

*action symbol:*

*include variable* $\uparrow memory_1$ $\uparrow$name $\uparrow memory_2$

*implies* $memory_1 = memory_2 \cup \{(name, undefined)\}$

[AT10] prog-length [*program length (implementation defined)*]

*constant:* $n_1$ = *implementation defined maximum*

[AT11] num-ids [*number of identifiers declared*]

*values:* *positive integers*

*constants:* *zero-ids* = *0*

$n_2$ = *implementation defined maximum*

*action symbol:*

*add one id* $\uparrow num\text{-}ids_1$ $\uparrow num\text{-}ids_2$

*implies* $num\text{-}ids_2 = num\text{-}ids_1 + 1$

[AT12] num-digits [*number of digits in a number*]

*values:* *positive integers*

*constants:* *zero-ids* = *0*

$n_3$ = *implementation defined maximum*

*action symbol:*

*add one digit* $\uparrow num\text{-}digits_1$ $\uparrow num\text{-}digits_2$

*implies* $num\text{-}digits_2 = num\text{-}digits_1 + 1$

[AT13] num-letters [*number of letters in an identifier*]

*values:* *positive integers*

*constants:* *zero-letters* = *0*

$n_4$ = *implementation defined maximum*

*action symbol:*

*add one letter* $\uparrow num\text{-}letters_1$ $\uparrow num\text{-}letters_2$

*implies* $num\text{-}letters_2 = num\text{-}letters_1 + 1$

Additional Data Type for the Execution Phase

*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 6.2 Continued

Additional Actions for the Execution Phase

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

*Execution of load* ↓name ↑stored-value$_1$ :

*if* ∃(t) (t=(name, stored-value$_2$) & t ∈ *global* memory & stored-value ≠ *undefined*)

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

*otherwise execution error* [*undefined variable reference*]

*execution of* store name stored-value$_1$:

(name, stored-value$_2$) ∈ *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*

*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* integer

*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$_{before}$ = $(v_1, \ldots v_n)$ & $n < n_6$

*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$_{before}$ = $(v_1, \ldots, v_n)$ & $n < n_6$

*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* [*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*]

A more complicated set of action symbols occurs in the definition of the loop statement in [AG11].

```
[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
```

The category <exp> has two inherited attributes, env and refs. In this case refs is set to *zero-refs* to show that an actual primitive value is required. The synthesized attribute prim-mode of <exp> gives the primitive mode of a value and the condition specified in the production stipulates that this primitive mode must be boolean. The second synthesized attribute of <exp> represents the actual expression value. This attribute is written in upper case to show that it can only be evaluated during the execution of the program. The action symbols written in upper case can be regarded as the terminals of the translation. The left-to-right order of the italic terminals in the derivation tree specifies the written form of the source program and the written sequence of action symbols corresponding to 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.

The production [AG22]:

```
[AG22]  <deref action>  ↓name  ↓refs1  ↓refs2  ↑VALUE1
                                  ::=  give value to attribute  ↓name  ↑VALUE1
                                       condition:  refs1 = refs2

                                     |  LOAD  ↓name  ↑VALUE2

                                       subtract one ref  ↓refs1  ↑refs3
                                      <deref action>  ↑VALUE2  ↓refs3  ↓refs2  ↑VALUE1
                                       condition:  refs1 > refs2
```

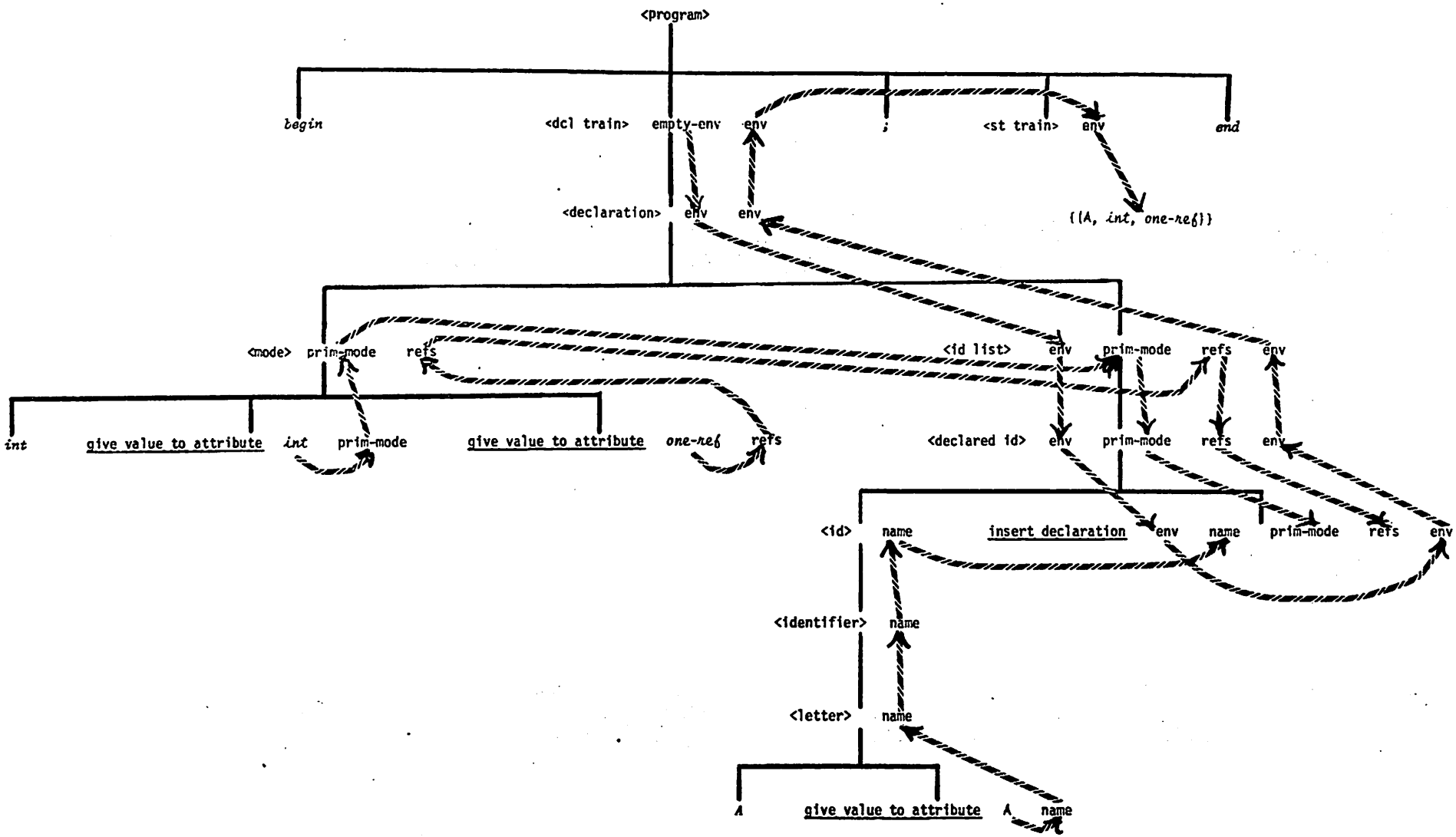


Figure 6.1: Derivation Tree for Attribute Grammars

presents a rather special case in that it requires additions to be made to the derivation tree during the evaluation of the attributes. This is done to generate the correct number of LOAD actions needed to perform the dereferencing. These actions cannot be constructed until the attribute refs has been evaluated.

For example, consider the program

```
begin
   int A;
   A = 1;
   output A
end
```

Using the productions of Table 6.1, it will be seen that after all possible attributes have been evaluated, the only actions that remain are LOAD and the WRITE INTEGRAL. This latter action is another example of something that must be added to the derivation tree during attribute evaluation when the value of prim-mode is known.

As shown above, the attribute grammar approach relies on the existence of some other target language for specifying semantics. For ASPLE, we have used the mechanism of action symbols. These action symbols are informally described in Table 6.2. 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. The final meaning of a program is taken as the final value of the output file state.

# 7. DISCUSSION

The example definitions illustrate a variety of formal models upon which we now base a more general discussion of formal definitions.

## 7.1 The Individual Definitions

A formal definition of a programming language should supply information to a variety of users. For example, serious programmers need to resolve detailed questions about facets of the language often not described in the language manuals. Language designers need to review their work and assess the full impact of their design decisions. Language implementors need a precise formulation of a language as a part of their job description. Writers of textbooks and reference manuals need information at all levels, from the general to the particular.

For 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 required information. As an illustration of typical questions that might be posed about a language, Table 7.1 shows six questions that cover a range of questions about ASPLE. To compare the four definition techniques in their ability to answer questions, we will take 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.

### 7.1a W-grammars

Since the question involves the assignment statement, we first look for a hyper-rule for assignments. While not immediately obvious, the relevant rule is hyper-rule [HR07]:

```
[HR07]  TABLE TAG becomes EXP val assignment  ::=  TABLE ref MODE TAG identifiers,
                                                   :=,
                                                   TABLE EXP MODE value,
```

Table 7.1 Sample Questions on ASPLE

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 dependant 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
```

which shows that the right-hand side of the assignment statement must be derivable from

TABLE EXP MODE value,

Following the form through several hyper-rules, we get to [HR17]:

```
[HR17]  TABLE EXP MODE primary  ::=  strong TABLE EXP MODE identifier,
                                   |  TABLE EXP MODE value pack,
                                   |  where MODE is INTBOOL,
                                      MODE EXP denotation,
                                   |  where MODE is bool,
                                      TABLE EXP compare pack,
```

Since the right-hand side of the assignment statement is a constant, a "denotation" in the W-grammar, we therefore choose the third alternative. From this we find that MODE, from which the declared mode of the identifier on the left side of the assignment can be derived, must be INTBOOL, and thus *ref int* is not permitted. Therefore, the assignment statement is illegal in the environment of the given program.

7.1b Production Systems

Here, we go directly to the production that deals with assignment statements, [PS07]:

```
[PS07]  :  ASGT STM<i':='e>  &  LEGAL<i:ρ>
        ⇒  dm_i = DECLARED MODE(i:ρ)  &
           dm_e = DECLARED MODE(e:ρ)  &
           PRIM MODE(dm_i) = PRIM MODE(dm_e).

           n_i = NUM REFS(dm_i)  &
           n_e = NUM REFS(dm_e)  &
           n_i ≤ n_e+1.
```

From this we see that $n_i$, the value of NUM REFS of the declared mode of the identifier, must be less than or equal to $n_e + 1$, the value of NUM REFS of the declared mode of the expression plus one. The value of NUM REFS for identifiers is derived from

[PS41]. through [PS46]:

[PS41] DERIVED MODE(*int*) ≡ REF INTEGER.
[PS43] DERIVED MODE(*ref* m) ≡ REF dm
• dm ≡ DERIVED MODE(m).

[PS44] NUM REFS(INTEGER) ≡ 0.
[PS46] NUM REFS(REF dm) ≡ 1 + NUM REFS(dm).

Here we see that the number of occurrences of REF for an identifier is one more than that given in the declaration for the identifier. For X, the number of references is 2. The value of NUM REFS for an integer constant is obtained from [PS36] and [PS44]:

[PS36] DECLARED MODE(n:ρ) ≡ INTEGER.
[PS44] NUM REFS(INTEGER) ≡ 0.

and thus is 0. Applying these values to the relation in [PS07],

$$n_i = 2 \qquad n_e = 0 \qquad n_i > n_e + 1$$

the assignment statement is shown to be illegal in the given context.

7.1 c The 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

where we see that the operation valid-mode-for-assignment is used to check the statement before translation. In [T25]:

[T25] valid-modes-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)))

we see that the value of ref-chain-length for the identifier minus 1 must be less than or equal to the value of ref-chain-length for the expression. In [T19]:

[T19] ref-chain-length(t)= is-c-id(t) → slength($s_1$·mode-of-id(t))+1
true → 0

the value of ref-chain-length for the identifier is one more than the number of occurrences of *ref* in the declaration and is, therefore, 2 whereas the value of ref-chain-length for any other type of expression is 0. Thus the relationship in valid-mode-for-assignment does not hold and the statement is rejected as being illegal in the given context.

## 7.1d Attribute Grammars

The production for the assignment statement is [AG09]:

```
[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
```

and the syntactic category <used id> is specified in [AG21]:

```
[AG21]  <used id> ↓env ↓prim-mode ↓refs ↓name
                                 ::= <id> ↓name
                                     condition: (name, prim-mode, refs) ∈ env
```

which shows that the number of references associated with the identifier is to be obtained from the environment. These were evaluated in [AG05]:

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

This shows that the value of the attribute refs is one greater than the number of occurrences of *ref* in the declaration. Thus the value of refs associated with <used id> in [AG09] is 2. This is reduced to 1 by the action symbol subtract one ref to give 1 as the value of the inherited attribute $refs_2$. Following the specification of <exp> takes us to [AG20]:

Since the right-hand side of the assignment is a constant, the second alternative applies and the condition stipulates that $refs_1$ = *zero-refs* Since the value of $refs_1$ is 1, the assignment statement is illegal in the given context.

**The reader is urged to use the definitions to obtain answers to the other questions in Table 7.1, and then draw conclusions on the relative clarity of the individual definitions.**

## 7.2. Formal Definitions in General

At present, the overwhelming proportion of formal definitions are exclusively for human consumption. The direct machine use of formal definitions is limited, and mainly used for automatic construction of a recognizer from a context-free grammar. Even beyond their limited machine use, it is our contention that all formal definitions must be well-suited for human consumption.

**While it may seem trite to remark on the paramount importance of the 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 helpful to clarity, is a dangerous mistress. She is the same siren that lures programmers onto the shoals of octal coding and the APL one-liner.

The use of the different definitions to answer the sample questions illustrates **that clarity depends both on the formal model and the notation of the definition method. However, even after the particular formalism and notation are chosen, there is still room to exercise the care and talent of the writer. The choice of mnemonic names, the use of comments and type faces, and the general organization play a vital part in the application of the formal mechanism.**

In the preparation of the example definitions in this paper, we have taken pains with myriad "details" to promote clarity, even to its extent of choosing a table layout that has almost doubled their conventional space requirements. We strongly feel that such an investment is small compared with the improvement in clarity and consequent usability of the definitions.

While there can be little argument about the need for clarity in formal definitions, there are many topics where the debate continues.

7.2a Should the Definition Model Be Based on the Notion of an Underlying Machine?

If the model is based on an underlying machine, there may be considerable extraneous detail in the definition that tends to obscure the meaning. For example, the mechanism for invoking operations in the VDL or Attribute Grammar definition has nothing directly to do with the ASPLE semantics. On the other hand, since the concept of an implementation is familiar to users, the use of a machine or machine-like instructions, albeit abstract, provides a readily grasped and precise metaphor that does not require the user to learn a new model of a more abstract nature.

7.2b 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" program be defined. There are several possible conventions, for example, a valid program is one with:

a. No context-free syntax errors.

b. No context-free or context-sensitive syntax errors.

c. No syntax errors and for which execution terminates for a particular set of input data.

d. No syntax errors and for which execution terminates for all possible sets of input data.

e. No syntax errors and for which execution terminates for all possible sets of input data and produces the "correct" answer.

The point of notable debate is between levels (c) and (d). In our example definitions, W-grammars and Attribute Grammars make no real distinction among the levels, Production Systems draws a clear line between levels (b-c), and VDL draws a clear line between levels (a-b) and (b-c).

### 7.2c How Should a Formal Definition Show Errors?

There are basically two different ways that formal definitions separate the legal programs from the illegal ones. A definition may be "analytic" and reject illegal programs explicitly, or the definition may be "generative" and make it impossible to generate an illegal program. From the user's point of view, the generative method leaves the question of whether a program is really illegal, 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-dependent or semantic errors explicitly but uses a generative approach that prevents the construction of a program with a context-free syntax error.

### 7.2 d Should a Definition Attempt to Indicate the Places Where an Implementation May Introduce Restrictions? 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. 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. The question whether it is possible to foresee all such restrictions is still open, although there is some evidence to suggest that it can in fact be done even for a very large language like PL/I.

## 7.3 The Importance of Formal Definitions

Because BNF is clear and is easy to learn and use, most definitions of programming languages include a BNF description of the context-free syntax. However, this is generally as far as the formal content of the definitions go, and as a result there is a tendency to believe that this is all that is required for a formal definition. There is an analagous confusion in many textbooks on compilers where the subject matter is limited to the theory of parsing. In formal definitions, the more difficult parts, the context-sensitive requirements and the semantics, are of much greater importance to the user.

Our example definitions indicate that the technology for full definitions is available, but there is still much work remaining before any notation achieves the level of general acceptance of BNF. This work must overcome considerable user resistance. This will only be done by great attention to the human engineering so that the general user feels that the definition is understandable by other than formal definition specialists.

Computer science has already made considerable progress without having a generally accepted 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, this progress has not been without severe consequences. For example:

a. There is still confusion over the difference between syntax and semantics.

b. Standardization efforts have been impeded by a lack of a well-accepted formal notation.

c. Despite the fact that there exists standards for programming languages, it is still chancy to move a program from one implementation to another, even on the same hardware.

d. 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.

e. Without a formal definition, it is difficult to write reference manuals and tutorial texts.

f. **Frequently the answers to detailed questions about a programming language 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. As an added benefit, there would be a tendency to improve the design of languages by bringing their complexities out into the open. It is easy to say that "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.

Despite the pressing need for further work on the development of formal definitions, they 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 meta-language 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.

## REFERENCES

[B1] G.V. Bochman
Semantics Evaluated from Left to Right
Technical Report 135, Department d'Informatique, Universite de Montreal, 1972.

[B2] R. Bosch, D. Grune, L. Meertens
ALEPH, A Language Encouraging Program Hierarchy
Proceedings of the International Computing Symposium, North Holland, 1973.

[C1] J. Cleaveland and R. Uzgalis
What Every Programmer Should Know About Grammar
Computer Science Department, University of California, Los Angeles, 1973.

[C2] K. Culik II
"A Model for the Formal Definition of Programming Languages"
International Journal of Computer Math, Volume 3, 1973.

[D1] J. Donovan and H.F. Ledgard
"A Formal System for the Specification of the Syntax and Translation of Computer Languages"
AFIPS, Proceedings of the Fall Joint Computer Conference, Volume 31, 1967.

[E1] C.C. Elgot and A. Robinson
"Random Access Stored-Program Machines. An Approach to Programming Languages"
Journal of ACM 11 (1964) No. 4, pp365-399

[F1] I. Fang
FOLDS, A Declarative Formal Language Definition System
Report STAN-CS-72-329, Computer Science Department, Stanford University, 1972.

[G1] J.V. Garwick
"The Definition of Programming Languages"
In: T.B. Steel Jr., Ed. Formal Language Description Languages, North-Holland Publ. Co. Amsterdam 1963, pp139-147.

[H1] C.A.R. Hoare and N. Wirth
"An Axiomatic Definition of the Programming Languages PASCAL"
Acta Informatica, pp. 335-355, Springer-Verlag, 1973.

[H2] C.A.R. Hoare
"Consistent and Complementary Formal Theories of the Semantics of Programming Languages"
Acta Informatica, Volume 3, 1974.

[K1] D.E. Knuth
"Examples of Formal Semantics"
Lecture Notes in Mathematics, Number 188, Springer-Verlag, New York, 1971.

[K2] C.H.A. Koster
"Affix Grammars"
Algol 68 Implementation, North-Holland Publishing Company, Amsterdam, 1971.
See also: D. Crowe, Generating parsers for affix grammars, Communications ACM 15, 728, 1972.

[L1] P.J. Landin
"Correspondence between Algol 60 and Church's Lambda-Notation"
Part 1: Communications of ACM 8 (1965) No. 2, pp89-101. Part 2: Communications of ACM 8 (1965) No. 3, pp158-165.

[L2] H.F. Ledgard
"Production Systems: Or Can We Do Better than BNF?"
Communications of the ACM, February 1974

[L3] H.F. Ledgard
"Production Systems: A Method for Defining the Syntax and Translation of Programming Languages"
Technical Report, University of Massachusetts, Amherst, 1975.

[L4] J.A.N. Lee
Computer Semantics
Van Nostrand, New York, 1972.

[L5] P.M. Lewis, D.J. Rosenkrantz, F.E. Stearns
"Attributed Translations"
ACM Symposium on Theory of Computing, Austin, April 1973.

[L6] P. Lucas, P. Lauer, and H. Stigleitner
Method and Notation for the Formal Definition of Programming Languages
IBM Technical Report 25.087, IBM Laboratory, Vienna, 1968.

[L7] P. Lucas and K. Walk
"On the Formal Description of PL/I"
Annual Review of Automatic Programming, Volume 6, Number 3, Pergamon Press, New York, 1969.

[M1] J. McCarthy
"Towards a Mathematical Science of Computation"
In: C.M. Popplewell, Ed. Information Processing 1962, North-Holland Publ. Co., Amsterdam 1963, pp21-28.

[M2] J. McCarthy
"A Formal Description of a Subset of ALGOL"
In: T.B. Steel Jr., Ed. Formal Language Description Languages, North-Holland Publ. Co. Amsterdam 1963, pp1-12.

[P1] E.L. Post
"Formal Reductions of the General Combinatorial Decision Problem"
American Journal of Mathematics, Volume 65, pp. 197-215, 1943.

[S1] P.F. Schuler
"Weakly Context-Sensitive Languages as a Model for Programming Languages"
Acta Informatica, Springer-Verlag, New York, 1974.

[S2] D. Scott and C. Strachey
"Toward a Mathematical Semantics for Computer Languages"
PRG-6, Programming Research Group, Oxford, 1971.

[S3] R.M. Smullyan
"Theory of Formal Systems"
Annals of Mathematical Studies, Number 47, Princeton University Press, Princeton, N.J., 1961.

[S4] T.B. Steel Jr.,
"Standards for Computers and Information Processing"
In: F.L. Alt and M. Rubinoff, Eds., Advances in Computers, Vol 8, Academic Press, New York 1967, pp

[T1] Robert Tennant
"The Denotational Semantics of Programming Languages"
To appear in Communications of the ACM, December, 1975(?)

[W1] P. Wegner
"The Vienna Definition Language"
Computing Surveys, March 1972.

[W2] A. van Wijngaarden, B.J. Mailloux, J.E. Peck, C.H.A. Koster
Report on the Algorithmic Language Algol 68
MR 101, Mathematisch Centrum, Amsterdam, 1969.