A Comparative Study of Formalisms for Programming Language Definition.

A thesis presented in partial fulfilment
of the requirements for the degree of
Master of Science in
Computer Science at
Massey University.

Ian Joseph Thompson

January 1975

Abstract

This study looks at a number of methods for defining the full syntax and semantics of computer programming languages. The syntax, especially the nature of context-dependent conditions in it, is first examined, then some extensions of context-free grammars are compared to see to what extent they can encompass the full context-conditions of typical programming languages. It is found that several syntax extensions are inadequate in this regard, and that the ability to calculate complicated functions and conditions, and to eventually delete the values of such functions, is needed. This ability may be obtained either by allowing unrestricted rules and meta-variables in the phrase-structure, or by associating mathematical functions either with individual production rules or with the whole context-free structure, to transform it into an 'abstract syntax'.

Since the form of a definition of a programming language semantics depends critically on how one conceives "meaning", five main types of semantics are considered: these are called 'natural', 'propositional', 'functional', and 'structural' semantics, as well as a semantics based on string rewriting rules. The five types are compared for their success in defining the semantics of computing languages, of the example Algol-like language ALEX in particular. Among other conclusions, it is found that the semantics of structures and computations on structures is the only type sufficiently comprehensive, precise, and readable.

Acknowledgements

In the preparation of this thesis, I wish particularly to thank Lloyd Thomas and Professor Tate for many valuable discussions, suggestions, and criticisms, and also to thank Bob Doran for help in his obtaining important source materials.

Table of Contents

1 The problems of programming language definition

Defining a computing language is generally done in two stages:

1) syntax : defining as rigorously as possible the set of all possible programs of the language, together with their formal structures and substructures.
2) semantics : associating with each such program its meaning, so that the effects of executing the program with its data are as rigorously defined as possible.

Details of definitions of these two stages will be discussed in the following two chapters (although the exact demarcation between the stages has varied for different people; I shall discuss this further in section 2.2).

The aim is to be able to define a significantly large language, including both stages of definition, and to this end there are several criteria for comparing the different systems examined later.

1) Scope of the definition method

Is it applicable to all features of all programming languages, or are there some features that can be encompassed either not at all, only with great difficulty, or at the cost of breaking up a neat system?

2) Elegance

A general aim is for a definition as readable, concise, and 'transparent' as possible. A readable definition should be understandable even with only a short initiation into the details of the formalism ; it should not be written in a wholly foreign language. A definition should also transparently follow the language being defined; this means that small changes in the language should require only small changes in the formal definition. Concerning conciseness, one should

distinguish between the method and its application to specific languages: a very simple method will generally lead to a very complicated definition.

3) Rigour

Syntax definitions should define, ideally, all and only the programs in the language, and assign correct formal structures to valid programs. They should avoid overlapping, incompatible, ambiguous, and/or missing specifications. Similarly with semantics. Note, however, that it is occasionally desireable to leave certain parts of a standard definition either completely open, or to deliberately give only a partial definition of them. For example, the details of real arithmetic, beyond certain basic conditions, may be postponed beyond the standard definition; and in any case the effect of merging parallel operations should intentionally be left undetermined.

4) Formalisation

The formalists' ideal is that a definition should say everything that can be said about all programs in the language, and in such a manner that mechanical statements can be made about the program without either using human understanding at this point, or running it on a computer with specific data. Such statements, for example, could concern the mechanical design of implementations, or the mechanical proofs of correctness, equivalence, etc., of programs in the language.

Chapter 2 looks at the syntactic, and chapter 3 the semantic, components of definitions of programming languages, and in the appendices I have used those methods which are sufficiently powerful for the definition of "ALEX". ALEX is the name which henceforth I give to a certain subset of Algol 60; it does not include arrays, for-loops, conditional expressions, or designational expressions, but it does include mixed-mode arithmetic, procedures, functions, call-by-name and call-by-value parameters, "goto" and "if" statements, and the implicit declaration of labels.

2 Syntax

2.1 Introduction

Syntactic theory is concerned with the formal systems to be used for the grammars of the languages to be defined. A grammar is a set of rules prescribing which sequences of symbols over a given alphabet constitute programs in the language, and it should also define the structure of valid programs in such a way that efficient translation or interpretation is possible.

The first grammars to be used in the definition of full-size languages were context-free grammars, and, by now, their theory is becoming well understood (see, for example, the text Hopcroft and Ullman,1968), having been by far the most common topic for formal investigations. As well, they are increasingly being applied in the construction of compilers. Feldman and Gries(1968) give an extensive survey of such and similar applications.

However, these context-free formalisms are inadequate to describe the full syntax of computer languages, and most of chapter 2 will be concerned with the various methods that have been proposed to overcome such inadequacies, and to try to capture these context-sensitive conditions.

In the next section we will look at the details of these problems, and after that consider the progressively more complex and more powerful formalisms proposed as possible solutions.

2.2 Context-dependent conditions

Context dependencies arise in all practical programming: when it comes to checking the scopes of identifiers; checking that identifiers are not declared twice in the same block; checking that identifier attributes given by the declaration are consistent both among themselves and with those presupposed by the later uses of that identifier; the notion of type conversion in general; and the correspondence between formal and actual parameters.

In some more complex languages - PL/I especially - because of the great variety of contexts in which identifiers can be used, and because many identifiers need only be implicitly or contextually declared, many of the context conditions are idiosyncratic, and need individual treatment. For example, in PL/I, there are no clean and simple universal rules telling how 'implicit' and 'contextual' attributes are derived from all the particular contexts. These processes are difficult to describe by a context-free grammar: it will be seen later (section 2.4.2 and Appendix E) how Ledgard(1974) has to use special functions, and table attributes of nonterminal symbols, to specify contextual declarations.

Because context-free grammars are easy to mechanically convert into parsing algorithms, the use of BNF to describe the context-free requirements on syntax has had a profound effect in computer science. It is common for compiler writers and others to use a context-free grammar to determine the main phrase structures of the source program. But, as I have described above, there are many critical problems of syntax that transcend context-free specifications. Hence the context-free grammar used in the compiler must be a 'covering grammar': one which generates all valid source programs, and many more invalid ones besides. It is then easy to consign the specific context conditions to the ragbag of 'semantics', to be dealt with by hand-written procedures in later stages of the compiler: stages that handle symbol tables and their entries, type conversions, parameter matching, etc.

This arrangement may work well for the compiler writer, but when someone wants a rigorous specification of the (context-sensitive) syntax, he should not be required to plumb the barely-fathomable depths of the compiler writer's ingenuity just to find out, say, in PL/I, whether labels are allowed on "declare" statements, or the exact nature of conversions in structure assignments. As a matter of principle, I believe that 'semantics' should be recognised for what it is, as meaning and effects, and not confused with context-dependent conditions: these conditions are essentially syntactical, being determinable statically, independently of any program execution. They may still be represented and enforced by auxiliary procedures written separately, but formally they are part of the syntactic component. When someone wants to standardise the language, and to rigorously define the set of all legal programs independently of any compiler, he will see that the hand-written procedures that are in the later stages of a compiler effectively act as syntactic conditions as well: if the condition is not satisfied in a source program, then that procedure will flag an error. He will want to separate these conditions from other functions of the procedures (e.g. code generation) and include all the conditions in the syntactic component, along with the covering context-free grammar.

This leads to the aim of trying to synthesize both the generative grammar and the context-dependent conditions within one syntactic formalism. This formalism must of course be more powerful than context-free grammars. For example, the grammar of this very simple set of Algol programs

'begin real w ; w := w end' for all w in $(a,b,c)^*$

is not context-free : if a context-free language does contain the above set of programs, then it must contain other programs that are incorrect Algol because an identifier will have been generated in the assignment statement that is not in the declaration part. And this is only a simple example; when arrays with any number

of dimensions or procedures with any number of parameters are allowed, the practical problems of checking identifier consistencies are more general and more complex.

Note that only if the context dependencies have significant regularity will it be possible to extend the syntax in a uniform manner. For example, in Algol 68, where constraints concerning the declaration and use of identifiers are independent of other features of the language (such as modes), the identification of identifiers may be simply stated, and even included in the phrase structure (see below, section 2.5). In PL/I, on the other hand, many constraints are specific to particular modes and particular contexts of use, so that it is here better to define these constraints using individual conditions, written alongside the relevant context-free rules of a covering grammar.

So now we look at various extensions of context-free grammars: extensions that try to encompass the context conditions in their formalisms.

2.3 Extended grammars

2.3.1 Context-sensitive grammars

Context-sensitive grammars restrict the applicability of context-free rules of the form A ::= w to specific surrounding contexts so that they have rules of the form rAs ::= rws (where A is a nonterminal, and w, r, and s are arbitary strings of terminals and nonterminals, provided w is not null).

However, straight context-sensitive formalisms are rarely used because they are too general for existing programming languages, which require only a number of specific types of context-dependent conditions. If they were used, definitions would be longwinded and untransparent ones: programming languages have context conditions between parts of a program that may be arbitarily far apart, such as between the declaration and all the uses of an identifier. Because the context-sensitive formalism only allows conditions from the immediately surrounding strings, incorporating the context dependencies of practical importance would involve, typically, having several 'marker' symbols that can travel to and fro along the partially expanded sentential form, carrying vital information, say, from declarations to applications later in the program. Defining all these operations for the marker symbols which do not directly illuminate the context dependencies or the semantic interpretation requires many times as many rules as there might be context-free rules in a covering grammar.

For example, to generate just the strings of the form $a^m b^n a^m b^n$, for any positive m and n, requires a set of rules such as

S ::= AXBAB
AXB ::= AABX / ABY
XA ::= UAA
BU ::= UB
AU ::= AX
BYA ::= BBAY
YA ::= AY

```
AYB ::= TABB / ABB
AT ::= TA
BAT ::= BYA
A ::= a
B ::= b
```

or, in the official "rAs ::= rws" format of above :

```
S ::= AXBAB        AXB ::= AXF        AXF ::= ABF
ABF ::= ABY        AXB ::= AAXG       AAXG ::= AABG
AABG ::= AABX      XA ::= UAA         BU ::= BE
BE ::= UE          UE ::= UB          AU ::= AX
BYA ::= BBYH       BBYH ::= BBAH      BBAH ::= BBAY
YA ::= YI          YI ::= AI          AI ::= AY
AYB ::= JYBB       JYBB ::= JABB      JABB ::= TABB
AYB ::= ABB        AT ::= AK          AK ::= TK
TK ::= TA          BAT ::= BAL        BAL ::= BYL
BYL ::= BYA        A ::= a            B ::= b.
```

The nonterminals X,Y,T, and U (and E through L) are only used as markers that carry information backwards and forwards along the sentential form to synchronise separate expansions. Having these extra symbols, and their associated rules, results in the context conditions of the original language being represented most obscurely, even for a language as simple as this example.

Further, context-sensitive grammars do not have nice features such as parsing algorithms which are computationally tolerable. And this is only complicated by the opacity of the derivations as given above, where many powerful extended rules are used to define a context-dependency whose algorithmic specification may be quite simple. It would be very much easier to write a small program to directly check for strings of the form $a^m b^n a^m b^n$ than it would be to use a program operating only from the context-sensitive rules listed above. Using the small program to directly check for the context-dependent conditions, which is the simpler, faster, and more readable alternative, defeats many of the purposes of using context-sensitive grammars.

2.3.2 Scattered-context grammars

The difficulties of the general context-sensitive formalism seem to stem from its only allowing contextual conditions on immediately surrounding strings. This limitation is avoided by scattered context grammars, introduced by Greibach and Hopcroft(1968), which allow rules of the form

$$(A_1,\ldots,A_n) ::= (w_1,\ldots,w_n)$$

which, when applied, means we have derivations like

$$x_1A_1x_2A_2\ldots x_nA_nx_{n+1} \Rightarrow x_1w_1x_2w_2\ldots x_nw_nx_{n+1}$$

for any intervening strings x_i. The basic formalism does not limit n, but Greibach and Hopcroft(1968) prove that increasing n above 2 does not increase the generative power of a scattered context grammar.

For an example of how this kind of grammar can express context conditions, consider the generation of the strings a^mb^n,a^mb^n by the set of rules

S ::= ABCAB
A ::= AA
B ::= BB
(A,C,A) ::= (a,C,a)
(B,C,B) ::= (b,C,b)
C ::= ,

I consider this set decidedly more elegant than that using the general context-sensitive formalism.

Thus, in principle at least, scattered context grammars can deal with ensuring that all used identifiers have been declared, but what about other sorts of context conditions, such as preventing the same identifier being declared twice in one block? The first problem is essentially one of generating disparate identifiers which are the same, the second of generating identifiers which are all different: a 'negative' rather than a 'positive' contextual condition.

Milgram and Rosenfeld(1971) discuss this problem, and argue that since the number of symbols to the left (or right) of a given symbol is potentially unbounded, it may be necessary to introduce explicit negative

contextual conditions into the formalism, since such conditions can no longer be imitated by sets of positive conditions. For example, in an ordinary context-sensitive grammar, a 'production' of the form

A ::= w if there is no C immediately to the left of A

is equivalent to the set of productions

XA ::= Xw for all X≠C in the vocabulary,

since once some X≠C is present immediately on A's left, C cannot be present. But in a scattered context grammar, in trying to represent the 'production'

A ::= w if there is no C anywhere to the left of A,

the number of different substrings that can occur just to the left of A is unbounded, and hence there is no general way of excluding C by any finite set of conditions. This suggests that scattered context grammars be generalised to allow negative context conditions directly: notationally, it could be represented by a line ($\bar{}$) above the symbol. The last 'production' above can then be written as

$(\bar{C},A) ::= (\bar{C},w)$.

Milgram and Rosenfeld(1971) prove that the resulting formalism is exactly as powerful as that of context-sensitive grammars, so now it is possible to define the programming-language-like sequence

$\underline{b}\ i_1,i_2,\ ..\ ,i_m\ ;\ v_1,v_2,\ ..\ ,v_n\ \underline{e}$

where each of the i's and v's is any identifier, but with the further context conditions that (a) all the i's are different (i.e. each identifier declared only once here) and (b) each of the v's is identical to one of the i's (i.e. only identifiers declared can be used).

S ::= $\underline{b}$ D ; U $\underline{e}$

D ::= I / I,D

U ::= V / V,U

($\underline{b}$, $\bar{t}$, I , $\bar{t}$, ;) ::= ($\underline{b}$,$\bar{t}$,t,$\bar{t}$,;) for all identifiers t

($\underline{b}$,t,;,V) ::= ($\underline{b}$,t,;,t) for all identifers t.

If there are a finite number of possible identifiers then there will be a correspondingly finite number of rules, but it is hardly a satisfactory situation when the size of the grammar increases in proportion to the number of identifiers possible in the language.

Alternatively, the formulae above could be regarded as rule schemata, which are 'meta-rules' for the generation of an indefinite number of other ordinary rules. This idea will be discussed more fully later, in section 2.5.2, with respect to the van Wijngaarden grammars, but even so, strictly speaking, the above example is not a valid scattered-context grammar. For only single symbols are allowed in the left-hand sequences of valid rules (see p9), and it is an essential part of the above grammar that, rather than checking for the symbol "t" alone, as in the rule

(b, t, ;, V) ::= (b, t, ;, t),

it checks for a large number of specific identifiers, e.g.

(b, a , ;, V) ::= (b, a , ;, a),

(b, ab, ;, V) ::= (b, ab, ;, ab), etc.

This means that it requires any sized string as a context condition, and this was not allowed in the original formulation.

However, even if scattered-context grammars are generalised to allow all of positive and negative context conditions, meta-rules and meta-variables, and having any sized strings in the context conditions, they are still not adequate to define full-scale programming languages. In particular they cannot handle identifiers having one of a number of attributes. For in a phrase-structure system such as this one, all relevant information must be contained in the current sentential form. Therefore information about variable attributes must be so included, but, because this information should not appear in the final program, it must be deleted at some stage by some rule. This is impossible, however, in even the extended scattered-context formalism, because it is based on context-sensitive rules rather than on unrestricted rules. Thus to handle identifier attributes, they must either be represented outside the phrase structure (cf. section 2.4), or unrestricted rewriting rules must be allowed (cf. section 2.5).

2.3.3 Programmed grammars

Another way of restricting the application of context-free rules is used in the programmed grammars of Rosenkrantz(1969). These have labelled productions, and allow successors of a given production to be chosen by specifying two sets of labels(production names). To try that production, the current sentential form is searched for an occurence of the non-terminal symbol on the left-hand side of the rule. If one is found (or the leftmost, if more than one), it is replaced by the expansion on the right-hand side of the production rule. Then, depending on whether or not the rule was used, one of the two sets of labels is taken, and an arbitary selection from it is made to give the next production to be tried.

An example, given by Rosenkrantz, is a programmed grammar which generates sentences of the form nha^n, where n is a nonnegative integer expressed as a binary number. A typical sentence is 101haaaaa - a sort of pseudo Hollerith field.

1.	S ::= 1SB	S(3)
2.	S ::= 0S	S(3)
3.	A ::= BB	S(3)F(4)
4.	B ::= A	S(4)F(1,2,5)
5.	S ::= h	S(6)
6.	A ::= a	S(6)F(STOP)

While these grammars define a proper extension of context-free grammars that is properly included within the context-sensitive family, they are not well suited to defining those context conditions found in programming languages. For example to define all and only the set of strings w;w (w being any identifier on the alphabet of 'a' and 'b'), which is a simplified version of the problem of matching the declaration with the use of an identifier, requires the productions

1.	S ::= A;B	S(2,3,6,7)
2.	A ::= a	S(4)
3.	A ::= b	S(5)
4.	B ::= a	S(STOP)
5.	B ::= b	S(STOP)
6.	A ::= aA	S(8)
7.	A ::= bA	S(9)
8.	B ::= aB	S(2,3,6,7)
9.	B ::= bB	S(2,3,6,7)

For a larger number, n say, of alphabet letters, we would require 4n+1 productions just to define w;w. Just as with the scattered context grammars of the previous section, this is clearly not a satisfactory arrangement. A definition would be much better if it were to use a single nonterminal (say 'alphanumeric-character') to generate the variety of any particular alphabet.

Generalising from this example, there is the result that a programmed grammar, with context-free core rules, cannot generate the set $w_1;w_2;\ldots;w_m$ where all the w_i are equal for this would need a number of rules varying with m, and m here has no fixed bound.

2.4 Grammars with associated context-dependent conditions

2.4.1 Property grammars (Stearns and Lewis,1969)

Instead of building the context conditions into the phrase structure rules, the aim of property grammars is to add extra conditions to each rule and require that the condition associated with any rule be satisfied before it can be applied. There are various ways of representing these associated conditions - see the following sections - but in property grammars these are conditions on the properties of identifiers at each place in the syntactic structure. Each nonterminal symbol has associated with it a table in which each identifier is given a certain numerical property. An example,given by Stearns and Lewis(1969), is a very simple block-structured language in which we have the choice of values for an identifier, 'id' say,

0 - not used
1 - this nonterminal produces id
2 - produces a declaration of id as a variable
3 - " " " " " " " label
4 - produces a use of id as a variable
5 - " " " " " " " label.

That is, at each node in the syntax tree - for each nonterminal - there is a table of values. Each identifier will index the table, and must each have one of the property values e.g. of the list above.

Conditions of productions are given as a collection of vectors of properties. If, for example, the rule is A ::= BCD, then we give it the vectors 0110 and 1300:

```
A ::= BCD
0     110
1     300
```

This means that when this rule is used in a derivation, each identifier must 'satisfy' one of the vectors. It may have property 0 in the table at A, 1 in the tables at B and C, and 0 at D; or else it will have 1 at A, 3 at B, and property 0 in the tables at both C and D.

Note that the grammar is both 'local' and 'static'. It is static in the sense that no order of derivation is required or implied, and it is local in the sense that the correctness depends only on there being a correct relation between each nonterminal and its immediate descendants.

To demonstrate their method, Stearns and Lewis apply it to a skeletal subset of Algol 60 - just enough to include block structure, and the declaration and use of identifiers as variables and labels (labels are declared by their being statement prefixes). They use the underlying grammar

P ::= B
B ::= begin D L end
D ::= dec ID, D / null
L ::= S ; L / null
S ::= B / use ID / lab ID / goto ID

where ID is any identifier
dec ID = declaration of ID as a variable
use ID = statement using ID as a variable
lab ID = statement labelled with the identifier ID
goto ID = statement causing transfer to ID.

Then, using the attributes 0 through 5 described earlier this section, Stearns and Lewis give the production-and-condition set

```
P ::= B        B ::= begin D L end        S ::= B
0      0       0       0   0 0   0       0      0
                0       0   2 0   0       4      4
                0       0   2 4   0       5      5
L ::= S;L       0       0   0 3   0
0      0 0      4       0   0 4   0
3      0 3      5       0   0 5   0       S ::= use ID
3      3 0                                0         0
3      3 5                                4         1
3      5 3      D ::= dec ID, D
4      0 4      0         0     0
4      4 0      2         0     2         S ::= lab ID
4      4 4      2         1     0         0         0
5      0 5                                3         1
5      5 0
5      5 5
                D ::= null
                0                         S ::= goto ID
                                          0         0
L ::= null                                5         1
0
```

This scheme is elegantly applicable to such a simple language as this. However, for others it is seriously restricting in that it requires a finite set of properties to be chosen beforehand. This is not always possible, even in a languge such as Algol 60. For instance, an array may have an arbitarily large number of dimensions, and a procedure or function can have any number of parameters (each of a certain type). Although some implementations use run-time checking of both subscripting and parameter matching, this is only useful when arrays or procedures are passed as procedure arguments without the type of the corresponding formal parameter being fully specified. In all other situations it is possible and usually desirable to check these conditions syntactically, and property grammars with finite sets of properties are unable to do this when any number of array dimensions or procedure parameters is allowed, for then there is no fixed bound to the number of properties possible for any identifier.

2.4.2 Grammars using 'inherited' and 'derived' attributes, Production Systems

Whereas property grammars have a value for each identifier at each node in the syntax tree, grammars in this system have attributes only for each node (each nonterminal symbol). But these attributes can be much more complex than just integer values. They could be functions, for example, or they could be whole tables, which can be used to model, as in property grammars, the symbol table of identifiers 'visible' to that particular place in the program. And the values in these tables and of these functions are not restricted to integers: any mathematical object is possible.

The method for constructing these attributes is also more complex than that of property grammars. Instead of having a predetermined table of alternatives, with each production rule, for 'A' say, there are now associated instructions to calculate the attributes of this A. Each attribute of A may be either composed from the attributes of the descendant nodes of A, or it may depend only on the attributes of the ancestors of A, i.e. on the context of A. These are called by Knuth(1968) 'synthesised' (or 'derived') and 'inherited' attributes respectively. So the initial nonterminal (e.g. 'program' at the root of the parse tree) can have no inherited attributes, and the derived attributes of terminals, constants, and identifiers etc., can only be constant.

In an actual parse tree, there may be considerable interaction between inherited and derived attributes while they are being calculated, but potentially circular definitions can be detected using an algorithm formulated in Knuth(1968).

Knuth(1971) contends that the definition of programming languages by means of inherited and derived attributes corresponds closely to the way people understand them, because the essential idea of such definitions is that each attribute is defined by local rules: rules and

conditions on each nonterminal that start only from the attributes in the immediate neighbourhood of that node.

For examples of his method Knuth(1971) defines Turingol, an Algol-like language to program Turing machines, and he gives a formal definition of the class of all reducible lambda-calculus expressions. Maurer(1972) also uses the method: to define a simple subset of Fortran II (Fortran without declarations, subroutines, arrays, common, or input-output, but with real and integer expressions and assignments, if statements, go to statements plain and computed, do statements, and general program organisation). Most of his definition concerns the operational semantics of the various statements. This will be considered in more detail later (section 3.4); what is of concern here is how he deals with the several context conditions present in his subset of Fortran. In fact, his problems are simple; since there is no problem of matching declarations to uses because variables are not declared in this subset. And labels cannot be confused with variables or integer constants because their possible contexts of use do not overlap. That only leaves the problems of checking that there are no duplicate label definitions, and that there are no transfers to undefined labels. To this purpose, one attribute of each program 'section' is the 'label-function', which is a set of ordered pairs (label,line-number), one for each of the labels used in that section. When another statement is added to the section by the production

section,x ::= section,z statement,v

there are the associated instructions

$$x^{\text{label-function}} = z^{\text{label-function}} \cup (v^{\text{label}}, v^{\text{line-number}})$$

and the condition

$$(v^{\text{label}} = \text{null OR } z^{\text{label-function}}(v^{\text{label}}) \text{ is undefined})$$

The first instruction is used to construct the composite label-function attribute as a derived attribute: derived from the label-function attribute of z and the

label and line-number attributes of the statement v. The condition checks that there are no duplicate labels by checking, if the added statement has a label, that that label does not already occur within the preceding section z. Together with a similarly worded condition that ensures that labels used in "go to" statements must have an entry in the label-function of the whole program, all the context-dependent conditions in this simple Fortran subset have been defined.

The 'Production Systems' of Ledgard(1974) have many similarities to the schemes of Knuth and Maurer, but Ledgard has used the idea of complex attributes much more extensively, especially for defining the context conditions. He uses a great variety of conditions, operations, and functions to compute the required attributes. As well as using well-known functions, in Production Systems one can define, recursively if necessary, new functions of any nature that would help the definition as a whole.

As an example of a largerscale definition, Ledgard (1974) defines the complete syntax of a significant part of PL/I. This subset - which he calls PL.1 - includes block structure, explicit, implicit, and contextual declarations, checking compatibilities of declarations with uses, and use of arithmetic, string, pointer and structured variables and their possible conversions. In Appendix E, I have used his method to define the subset of Algol 60 which I have called ALEX in chapter 1. In this definition there are special functions for computing symbol-tables from declarations, and for computing types of expressions. Neither of these functions is very simple, but because they are separate from, rather than integral with, the phrase structure rules, one can directly use more powerful mathematics e.g. set theory. It is particularly useful, as in this definition, to compose and use functions mathematically, completely avoiding string-handling automata, e.g. Turing machines, whose calculation of even simple functions tends to be extremely laborious.

2.4.3 Conditions defined by functions on the whole context-free syntactic structure

The previous methods have associated the various context conditions with particular production rules so that they can be checked when those rules are applied; it is also possible to define the conditions by functions on the whole program. To do it this way, all the program text is first parsed to give its 'concrete syntax', this is then transformed by a 'translate' function to give the 'abstract syntax'. The translation function, operating on the whole concrete-syntax structure, is able to check and change as much of that structure as is desired: it can check any necessary context conditions, and usually it will rearrange the syntax to remove 'syntactic sugar' such as extra keywords, semicolons, etc., and to produce a structure most suitable for later use eg. for interpretation or code generation.

There are many such practical advantages in having two stages of syntax in this way: whereas context-free grammars assume that the set of objects whose syntax is being defined is a set of strings, and, therefore, sensitive to the textual order in which the components appear in a string, compilers and interpreters are concerned not with strings written down but with structures represented in some more abstract fashion. Components of structures in a computer are identified not by a linear textual ordering but by pointers (selectors) that associate another object with each of the components of a structure. 'Abstract syntax' is the formal representation of these general structures, and the Vienna Definition Language (VDL) is a language especially designed to describe these structures and operations on them (see section 3.6.2.1 and Lucas(1968) or Wegner(1972). Also the next section).

Another advantage is that arbitary functions and conditions may be defined on any part of the concrete syntax to verify context conditions of any complexity. Often a programming language will allow contextual

declarations, or a large variety of default attributes and defaulting constructions, and later sections of the definitions, eg. an interpreter section, should not have to work out these details each time statements are executed. For, properly viewed, these are syntactic details, being determinable statically, and independently of any particular execution of the program.

2.4.3.1 The Vienna Definition method

As mentioned earlier, the Vienna Definition Language (VDL) was designed to handle abstract syntaxes structured with 'objects' and 'selectors'. More detail of this language is first given, closely following the presentation of Wegner(1972), then in section 2.4.3.1.2 a way of going from the program text to the abstract syntax, checking context conditions in the process, is examined.

2.4.3.1.1 Objects and selectors and operations in VDL

There are two classes of data objects in VDL:

1) Elementary objects, atomic objects which have no components but have names which may be relevant during interpretation.
2) Composite objects, which may be built up from elementary objects by 'construction operators'. Composite objects have components that may be selected by unique selectors.The components may be either elementary objects or composite objects.

Figure 1 is a representation of a composite object whose three components a,+,b may be selected by the selectors s_1,s_2 and s_3 respectively. The association of a selector s with an object ob will be represented by the notation <s:ob> and will be referred to as a 'selector-object pair'. The construction of the composite object of Figure 1 from its three components may be accomplished by the application of the 'construction operator' u_0 to the three selector-object pairs $\langle s_1:a\rangle,\langle s_2:+\rangle,\langle s_3:b\rangle$:

$u_0(\langle s_1:a\rangle,\langle s_2:+\rangle,\langle s_3:b\rangle)$.

The construction operator u_0 takes as its arguments a variable number of selector-object pairs of the form $\langle s_i:t_i\rangle$, where s_i is a unique selector and t_i is an elementary object or a tree-structured composite object. The general form of u_0 is illustrated in Figure 2.

Selectors may be used as 'operators' that, when applied to a structured object, select one of its components. For example, if 't' is the composite object of Figure 1, then $s_1(t)$ selects the elementary object 'a'. When the object to which the selector s is applied has no edge labeled "s" emanating from its root vertex, then s(t) is defined as the null object 'null'.

In order to be able to update and manipulate data structures, the construction operator u_0 is now generalised to allow assignment of values to components of data structures and allocation of new components in an existing data structure. The assignment operator, denoted by u, may defined as follows:

$u(t;\langle x:t'\rangle)$ is: assign the value t' to the x-component of t if t has a x-component; add a new x-component t' to t if t does not have a x-component; delete the x-component of t if t'=null. This is also extended to allow many selector-object pairs $\langle x:t'\rangle$.

The 'u' and 'u_0' operators are central to the Vienna Definition Language. In terms of these data-structure operations it is possible to build translation or interpretation routines as required. In section 3.6.2.1 this composition of elementary operations into conditional expressions and subroutines will be described; it is in this way that the 'translate' routine of Appendix D and the interpret-program instructions of Appendix I are constructed.

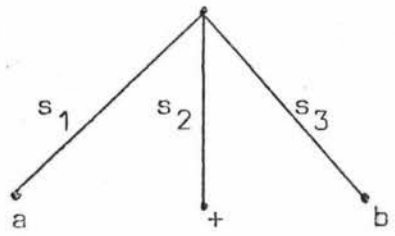

Figure 1.

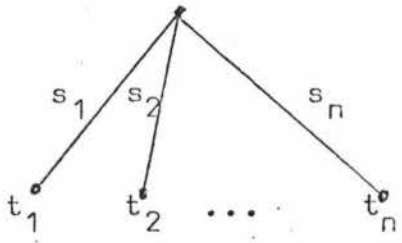

Figure 2. The composite object
$u_0(\langle s_1:t_1\rangle, \langle s_2:t_2\rangle, \ldots, \langle s_n:t_n\rangle)$.

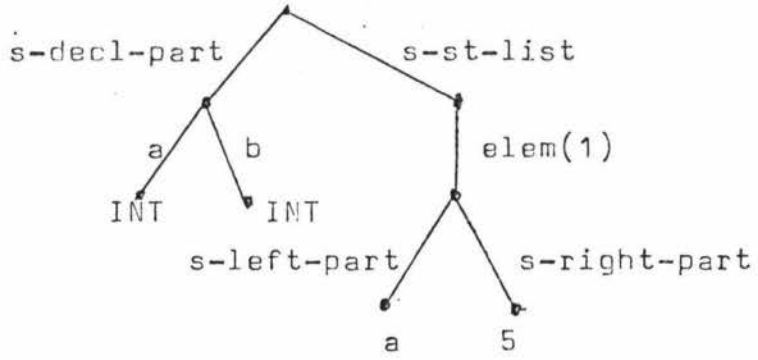

Figure 3. VDL abstract structure.

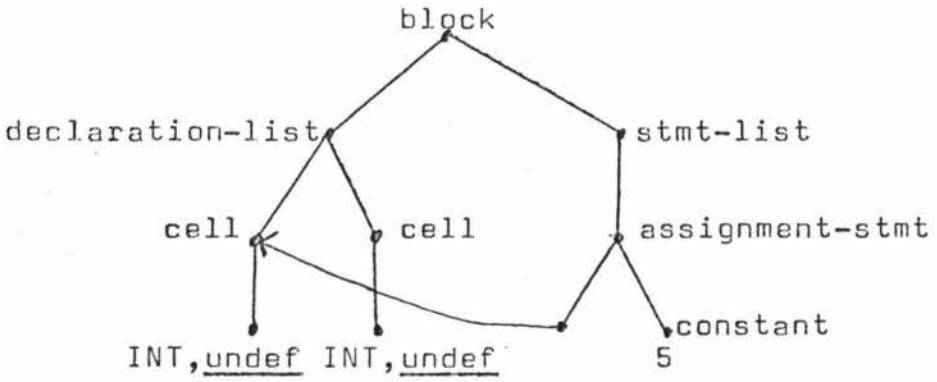

Figure 4. LFMS abstract structure

2.4.3.1.2 Constructing the abstract syntax in VDL

The translation from concrete programs to abstract programs is performed in two steps by the two functions 'parse' and 'translate': if 'txt' is a concrete program, the corresponding abstract program is defined as

translate(parse(txt)).

The link between the two steps, namely the result of parse and the argument of translate, is a structured object t which is called the abstract representation of txt and may be thought of as the parsing tree of txt, according to the concrete syntax of the language.

For the example language ALEX, in Appendix A is given the concrete syntax in the usual Backus Normal Form (BNF). This notation, however, is not the most suitable for a language which uses data structures with selectors and unordered components. Instead, the VDL definition uses the 'abstract representation of the concrete syntax' given in Appendix B. This, together with a function 'generate', constitutes a formal definition of an algorithm for generating all concrete programs of the programming language. Hence they are equivalent to the production rules of the concrete syntax (Appendix A) together with the instructions for their use.

The function 'generate', mapping the object t, satisfying the predicate is-c-progr of Appendix B, into a set of character value lists, is now defined[1] as follows:

generate(t) =
  is-null(t) ::= null
  slength(t)=0 ::= t
  T ::= generate(s_1(t)) + $\overset{\text{slength(t)}}{\underset{i=2}{\text{CONC}}}$ (generate(s-del(t)) +generate(s_i(t)))

(it has been assumed that a special selector s-del has been used to select list delimiters (cf. e.g., AR3, AR6 and AR12 etc. of Appendix B))

The function 'parse', which is the inverse of the function 'generate', is defined, following Lucas(1968), as follows:

1 see Appendix D for the definition of "slength".

parse(txt) =
(ι t) (txt = generate(t) & is-c-progr(t))[1]

Assuming that the concrete syntax is unambiguous, the meaning of this definition is that the function 'parse' transforms a program character-text into its parsing tree 't', provided the list is a syntactically correct program. This is in preparation for the 'translate' function, and so far depends only on the covering context-free grammar, which includes no context conditions.

'Translate' is a function on the concrete syntax to transform it into an abstract-syntax structure with the same meaning, and at the same time to check all the context-dependencies that can be statically verified in a reasonable time. For ALEX, this requires the 19 functions defined in Appendix D, and produces a structure satisfying the predicate 'is-progr' of the abstract syntax, defined in Appendix C.

A total of 11 context conditions are checked during the operation of the translation function of Appendix D, these are:

in T1 valid concrete syntax.

T6 no duplicate declaration of the same identifier at the same block level.

T8 the return-expression of a function must be of the required type.

T9 no duplicate identifiers in a formal-parameter list.

T12 gotos must refer to declared labels.

T13 expression in an if-clause must be of type 'logical'.

T14 the expression in an assignment stmt. must be convertible to the type of the variable.

T15 a procedure call must reference a procedure id.

T16 the lengths of formal- and actual-parameter lists must be the equal.

T17 an actual parameter must be convertible to the type of the formal parameter.

T18 variables in expressions must have been declared as such.

The conjunction of all these conditions is not explicitly defined as a function that can be evaluated.

1 see Appendix D for the definition of ι, the iota quantifier.

Rather, in the course of converting the syntax from concrete to abstract, if any one of the conditions is not satisfied, then the corresponding translation function will produce error, and so further translation in this defining translator ceases. So, instead of having conditions on the concrete syntax explicitly stated, they are now implicit in the operation of the translation functions.

2.4.3.2 Linked-Forest Manipulation Systems

A slightly different, but related, form for the abstract syntax, and another way of constructing it, has been desribed by Čulík(1973). There are three features in his Systems which have not been considered yet in this survey: 1) to allow arbitary links around the tree structures, 2) to have Markov-like rewriting rules to manipulate these linked-forest structures, and 3) to construct the abstract syntax in parallel with the concrete text by having small linked-forest manipulation systems at strategic places to check context conditions.

Allowing arbitary links points out some shortcomings of the Vienna Definition method that concern some particular context-dependent relationships: for example, to relate a goto-stmt. to the corresponding label, the label to the corresponding program point, procedure calls to procedure declarations, etc. I digress to describe the problem, and the solution attempted in VDL, in more detail.

The Vienna Definition method handles such relations by allowing selectors to be elementary objects. The result of selecting such an elementary object would be a further selector which may be applied to some other tree node which is already known - such as a directory node. For example, the small ALEX program

begin integer a,b; a:=5 end

would be represented by the abstract syntax tree of Figure 3.

Here, the left part of the tree representing 'a:=5' is the name 'a', and this name is used as a selector on the 'decl-part' node to obtain the declaration of this identifier 'a'.

This scheme works well for situations where, from some known node, only one selection is needed to find the required destination. With labels, the situation is not nearly so elegant. In ALEX I have followed the idea used in the PL/I definition (Alber,1969) in constructing "declarations" for each label that occurs in the given block. This declaration gives for the label a list of selectors which when successively applied to the statement-list of that block, will locate the statement originally indicated by that label. Speaking formally with respect to the ALEX definition, a label attribute is the integer-list

$$(\langle elem(1):i_1\rangle, \langle elem(2):i_2\rangle, \ \ldots\ , \langle elem(n):i_n\rangle).$$

Then, if b is the block in which the label is declared,

$$s_{i_n}.s_{i_{n-1}}. \ \ldots\ .s_{i_1}(\text{s-st-list(b)})$$

is the labelled statement.

Having to store a vector of selectors in this manner is a cumbersome construction, and results from not having pointers that can directly link disparate parts of the syntax tree, as, for example, is possible in Čulík's(1973) 'linked-forest manipulation systems'.

In addition to the tree structures of VDL, there can now be additional directed edges of a different type(called 'pointers' or 'links' or 'designators') that more directly reflect the context-sensitive aspects of the program. For example, it is possible to represent the short program of above by the structure of Figure 4 (satisfying Appendix M).

Now one of the great advantages in having only pure tree structures is that subtree replacement and construction is easily done, and guaranteed to produce a valid result. With the extra links, in the extended formalism, more care must be taken. Čulík(1973) formally defines a class of 'linked-forest manipulation systems' (LFMS):

these are more complicated than subtree replacement mechanisms because now more general types of subgraphs than trees are being considered, and also because of the linking. However, he manages to define the manipulation systems in such a way that coalescing an unchanged part of a graph with a newly created subgraph causes no problems. The formalism is more properly described in section 3.6.2.4: where the similarities to Markov systems are considered in more detail.

As in the Vienna method, a concrete syntax is defined using a context-free grammar. But instead of converting this to an abstract syntax once the whole program has been parsed, there is now with each context-free rule a production for the parallel construction of the abstract-syntax tree. The ALEX definition of Appendix G requires 29 such pairs of productions.

This by itself would only be sufficient for producing the abstract syntax if there were no context-sensitive conditions: it is just the formalism for syntax-directed translations of CF languages. To check these conditions and translate properly to the abstract syntax, with links where appropriate, Čulík(1973) associates with several pairs of productions a small linked-forest manipulation system. These LFMSs operate on all the abstract syntax derived from the node with which each is associated, and the structure suitably prepared, e.g. for interpretation, and in which context conditions have been verified, is returned.

For example, after all the context-free syntax for a block or procedure has been chosen and constructed, there is a LFMS to check that no identifier has been declared twice (Appendix G, rule 2.1), to put in links (designators) from the uses to the declaration of identifiers (rules 2.2 to 5), to check that parameter lists agree in number and types (rules 2.7 to 10). Several other changes are made so that the resulting abstract syntax, with links, is most useful for the semantic part of the definition (eg. rule 2.11, which converts declarations into data cells for execution). The semantic part will be described in section 3.6.2.4 (see also Appendices M and N).

2.5 van Wijngaarden grammars

2.5.1 Grammars with unrestricted rewriting rules (type 0)[1]

Type 0 grammars are those with rules of the form v ::= w , where v & w are any sentential forms (v having at least one nonterminal symbol). Traditionally in computer science these grammars have been little used. They are the most general phrase-generation grammars- being able to produce all recursively enumerable sets, and to simulate Turing machines - but being so general they have little built-in structure that can be systematically exploited. A parser, for example, cannot represent the derivation structure of a program by a simple tree.

However, by imposing some discipline on the use of these 'unrestricted' rules, one can benefit from the power of type 0 systems while having an intelligible grammar too. One can define a wide variety of predicates to delimit various sets of strings - such as the sets A,B such that A=B, A≠B, A is contained in B, or A begins with B, etc. - that cannot be defined by context-free grammars, and use these set predicates to judiciously enforce context-sensitive conditions through the grammar. It would no longer be necessary to have conditions written apart from the production rules, e.g. as in Production Systems (see section 2.4.2, also Appendix E). Instead of using set-theoretic notions to define, for example, symbol-table membership, it can now be done syntactically.

[1] Type 0 in the Chomsky hierarchy.

2.5.2 Grammars with metasyntactic variables

To properly define a set predicate of the section above, 'A=B' for example, it would be best if it were possible to define it only once, with A and B string variables, and then apply it to as many particular strings as required. Because A and B stand for syntactic variables (i.e. nonterminals), they will be called "metavariables", and productions which show which strings of ordinary variables they can produce are called "metaproductions". An example, from Appendix F, is the predicate definition for A=B :

where NOTETY is NOTETY : true.

with the metaproductions for NOTETY to generate any character string (possibly empty):

NOTETY :: NOTION ; EMPTY .
NOTION :: CHAR ; NOTION CHAR.
CHAR :: a ; b ; c ... etc.

The predicate definition is therefore a finite abbreviation for the infinite number of ordinary productions

where is : true.
where a is a : true.
where b is b : true.
where ab is ab : true. etc.

There are no productions for "where a is b", so "true" can never be produced from it.

2.5.2.1 An historical digression

Metavariables were introduced in Algol W (Bauer et al. 1968) to abbreviate the writing of rules for real, double precision real, integer etc., but the first large scale use was in the 1968 definition of Algol 68 (van Wijngaarden et al., 1968). Here, again , they were used not to describe context-dependencies, but to allow the definition by a finite syntax of all of an infinite number of modes; of the rules for their valid declaration and uses in expressions etc. Thus, in a finite number of rules and metarules, it was possible to define the matching of actual and formal parameters to a procedure or operator, even though the number of parameters has no fixed bound.

Baker(1972) gives a general introduction to van Wijngaarden grammars, which use both unrestricted rewriting rules and metavariables; and one of his examples is how parameter matching can be defined by them. (See, for example, the rules for "procedure call" and "function call" in Appendix F).

These formalisms were originally designed to abbreviate to manageable proportions the description of all the context-free rules necessary for all the modes of procedures, structures, etc., that a programmer might want to use in his program, but it was soon realised that the formal system was much more powerful than the BNF notation used in the Algol 60 report (Naur et. al.,1963) : whereas in Chomsky's classification BNF is of type 2, these two-level grammars are of type 0 (Sintzoff(1967), see also Baker(1972)).

At first the context conditions of Algol 68 to do with avoiding multiple declarations, matching use with some preceding declaration, etc., were defined by special "context conditions" in English which were quite separate from the phrase-structure formalism. These were avoided in subsequent versions of the Algol 68 report, resulting in van Wijngaarden et al.(1974), by using the meta-syntactic features to define various predicates as mentioned above, and described in more detail below. Appendix F gives a definition of the complete syntax of ALEX using this method.

2.5.2.2 Context-dependencies defined by van Wijngaarden grammars

The entire process of matching indentifiers with their declarations can now be described in the syntax, using a metavariable (e.g. "NEST") to stand for the 'symbol table': it has metaproductions which are capable of describing, and of passing on to the descendant constructs, all the declared information which is available at any particular place in the program.

In the 1974 definition of Algol 68 - which I largely follow in the ALEX definition in Appendix F - "NEST"

generates a sequence of "LAYER"s, one for each block; each LAYER being a sequence of properties(PROPS) of identifiers. The program

```
begin real x,y;              new x has real y has real
      ..
   begin integer x;          new x has real y has real new x has
         ..                                              integer
   end
end
```

program NEST

requires, then, for each block the "NEST"s indicated.

In the generation of a valid "program", "NEST" is first expanded to give all the variables required, together with their desired attributes. Now one of the features of van Wijngaarden grammars is that strings generated may be easily copied. In the present situation, one copy of the "NEST" is used to construct all the declarations, and other copies are passed down the derivation tree wherever an identifier will be generated. So we have "NEST statement", "NEST assignment", then, for example, "real NEST expression", "real NEST unary" etc., finally down to "real NEST identifier". Productions for "TYPE NEST identifier" must then generate some identifier "ID" such that "ID" has a valid "TYPE" in the "NEST" of this place in the derivation tree. To do this, the ALEX definition includes the scheme

TYPE NEST identifier: ID token,
where ID has TYPE from NEST.

The "ID token" generates the identifier representation, and the "where" clause is a predicate. The "where .. has .. from .." is defined in Appendix F in the section "identifiers".

Extensive use is made of such predicates. These are strings of variables (and metavariables) which are deliberately made to yield blind alleys when certain conditions are not met, and yield empty terminal productions otherwise. Only if they produce "true", which produces "EMPTY" : the disappearance of the

predicate, will a valid program be produced. There are no other terminal productions of a predicate apart from the empty string: only blind alleys containing some non-terminals.

All this may seem excessively complicated, but it has been successfully used for a large scale language such as Algol 68. In that case, the disciplined use of type 0 productions (i.e. ones yielding EMPTY irreducibly) has given a concise formal definition of the contextual conditions, even though the language has been influenced to some extent by this method of definition.

One feature that aids our following the definition is the English-like names of the predicates. Once a person knows, for example, that the predicate "differs from" has been formally defined, its further details are not so important for him. For the compiler-writer too: he could directly define the predicate in terms of, say, machine primitives. Koster(1969,1971a,b) shows how in general to construct procedures from rules with metavariables. But for 'obvious' predicates it would be more efficient to ignore the details of the formal definition to instead interpret them directly. Of course, some predicates are not so obvious: the Algol 68 definition makes extensive use of these in defining more complex conditions such as the equivalence or non-equivalence of modes, and operator identification, etc. (see section 7 of van Wijngaarden et al.,1974). But these are still given names meaningful in English.

2.6 Conclusion

On the basis of the arguments that have been given, I conclude that the first four methods surveyed - contex-sensitive, scattered-context, programmed, and property grammars - are either too weak for, or unable to express in a reasonable number of rules, the context conditions of a medium-sized language such as Algol or ALEX.

The remaining formalisms can be divided into two broad groups - those which use ordinary mathematics (e.g. including set theory), and those which use unrestricted rewriting rules with metavariables, to describe the necessary context conditions. In the former, there are grammars using inherited and derived attributes, Production Systems, and the Vienna Definition method. In the later group there are the van Wijngaarden grammars, while the Linked-forest Manipulation Systems can be looked upon as a cross between the two groups.

The groups have their characteristic advantages and disadvantages. The van Wijngaarden grammars have the advantage of staying close to the basic notion of a syntactic grammar for phrase-structure languages and still including the required contextual conditions. However, being a definition only of the syntax, there is no easy way of formally attaching the semantics of the language being specified. In the Algol 68 report (van Wijngaarden et al.,1968) this is done by associating English 'semantics' sections after each section of syntax rules - following here the Algol 60 report (Naur et al.,1963). This may be quite readable, but, compared with what has been attained in some other methods, it should be possible to make this association more formal.

Those schemes, for example the Vienna Definition method or the Linked-forest Manipulation Systems, which define transformations from 'concrete' to 'abstract' syntaxes have advantages that are different than those of the van Wijngaarden grammars. They are more satisfactory when formal definitions of semantics or algorithmic analyses are to be included in the overall definition, as will be properly seen in the next chapter. The 'abstract

syntax', in which context conditions have been confirmed, is a useful intermediate structure (see section 2.4.3). However, for someone who only wants a specification of syntax, finding the answer to even a simple question (e.g. whether, in PL/I, labels are allowed on "declare" statements) can involve prolonged fathoming of the translation functions. For they are mathematically stated in such a way that one needs to be fully acquainted with the structures being analysed and reconstructed at each stage in order to be able to follow the translation of the sort of program in which one is interested. For someone not so initiated, the Vienna Definition Language is practically opaque, and definitions using it (e.g. Appendices D & I, or the PL/I definition of Walk et al.,1969) are unusable in many cases. I consider that a scheme should be of the sort where, if someone knows only a part of the method and structures involved, one can understand at least a proportional part of a definition using them. For this, it helps if English names with 'obvious' meanings are used, rather than large amounts of mathematical notation which may be hard to read.

3 Semantics

3.1 Introduction

Semantics is concerned with meaning, and when one studies the semantics of programming languages, one is concerned with the meaning of programs either as their 'effect' or as their 'value': there are different ways of representing meaning.

Semantic theory is a somewhat less developed subject compared with the theory of syntax, and as yet there are no direct applications of semantic theory which have the importance, say, of syntactic theory in compiler construction. However, semantic theory is not being neglected, and there are several important long-range goals:

1) to help in the investigation of the properties of specific programs, and in constructing proofs of correctness of given programs,
2) to help in the design and evaluation of languages,
3) to lead to methods for the complete formal definition of programming languages, to be used as a reference by the compiler writer and the programmer, and for standardisation purposes,
4) in combination with the formal studies of machine operations, to help in the construction of compilers.

The form of a semantic definition depends very much on how one conceives "meaning" : various alternatives are possible, and one can distinguish five main types of meaning, as well as a scheme modelled on syntactic systems:

1) natural semantics
2) compiler semantics
3) structural semantics
4) functional semantics
5) propositional semantics
6) semantics based on string rewriting rules,

in order of increasingly formal nature. However, the most formal definitions tend to be the least readable and useful, so the types of semantics will not be discussed in the order above.

3.2 Natural semantics

Natural meanings are those we use every day to think with, but unfortunately the sciences have not given us a good theoretical hold on them. Even so, they are still relevant for programming language definition. For if we are to come to understand any definition, it must be remembered that our very understanding presupposes these 'natural meanings'. Everyone who uses a programming language must have an informal desription of this language in his mind; otherwise he could not use the language. There is no question that these informal descriptions are probably different for different people, and that they will differ from any formal description. But in the end, any language definition fails or prevails according to how well it conveys our intuitive ideas about the language. Thus even in formal semantics one should always keep an eye on how people intuitively understand languages. Their informal understanding will of course differ greatly, as Naur(1964) remarks, from the formal description of any compiler, so that it is not very convenient to refer the programmer to the listing of the compiler and tell him to answer his own questions. A more easily understood definition could be based, for example, on the hierarchical specification of properties and actions of program structures : this is the basis of 'structured programming'. Later it will be seen how different definition methods compare in this respect.

3.2.1 Natural language semantics

Part way in rigour between a person's intuitive understanding and a formal definition, there are the natural languages: English in this case. Using natural languages has been the most common way of describing programming languages - familiar to everyone who has experience with programming manuals etc. These may range from an informal introduction to trying to specify the language as completely as possible (for an example, using PL/I, see IBM(1970)). With care one can obtain reasonable precision using English as a metalanguage especially if used in conjunction with some more formal scheme for syntax.

Note that this conjunction of English and formal syntax can be craftily arranged to greatly help the readability. So the Algol 60 report can have 'the symbols used for distinguishing the metalinguistic variables have been chosen to be words describing approxiamately the nature of the corresponding variable. Where words which have appeared in this manner are used elsewhere in the text they will refer to the corresponding syntactic definition.' Duncan(1964) contends that it is this convention which has given the Backus notation its value: 'by means of it, a word can be used as a metaidentifier and be manipulated formally while at the same time it can be used in the natural language text and even undergo inflexion. ... For example, you can have the word "label" in the syntax, and you can say "labels", "to label", "labeling", "labeled", "unlabeled", etc.' in the textual definition of the semantics. Such is the power of natural language that can be employed.

Returning to natural languages in general: they are easy for us to understand because they are informal, and much of what they say is tacit, not explicit. Schwartz (1970) points out that what one finds in a programming language manual is almost never a systematic formal definition of the meaning of programs. Instead, one finds a definition having a strongly expository flavour, in which certain salient features may be set forth with relative care, but in which much is suggested rather than stated and left for the reader, guided by certain principles of naturalness and minimum surprise, to supply by deduction. By their nature, definitions using natural languages are extremely difficult to process mechanically, even hypothetically. For a natural language itself incorporates a huge and not-formally-analysed body of tools, and we are still so far from being able to treat them formally, especially in regard to their semantic properties, that we have no way of processing such definitions, even in principle. That is, even if given what purports to be a complete natural language definition of a programming language, there is not yet any programmable

way either of verifying its completeness, or of mechanically transforming it into a complier or an interpreter for the language, or of mechanically determining whether any given compiler does in fact realise the object defined originally.

A definitional scheme ideally should aid us in rising from simplicity to complexity, and therefore a meta-language should be simpler than the language that it is being used to define. Although natural languages seem simple for us, they are very complex for computers, and so quite unsuitable in a mechanical world.

3.2.2 Semantics by compilers

It is possible to regard defining a programming language by a compiler as giving its meaning: by translating programs into another language whose semantics is known - usually a machine language for an actual or a hypothetical machine. For examples, see Feldman(1966), Garwick(1964), Nivat and Nolin(1964), or Wirth and Weber (1966),(also section 3.6.3).

Using compiler semantics assumes that the execution-time semantics of the target language is given as a primitive in terms of which source languages may be defined. This allows many of the more complicated aspects of the language to be treated only once, at compile-time, by algorithms which are guaranteed to terminate (the actual methods of treatment will be discussed in section 3.5.1).

However, especially from the theoretical viewpoint, definitions of languages by their compilers cannot be regarded as satisfactory for several reasons. One is that it only postpones the difficult problem of defining the basic interpretation of some computing language. And, as Wegner(1972) notes, a compiler by itself only gives the relation between the source language and the target language: it rarely gives anyone insight into the essential nature of the former. Because, too, compilers are usually heavily involved with language-irrelevant machine details, the best scheme would probably be a compiler-independent definition, and then one or more compilers can be constructed which satisfy this definition of the language.

3.3 Propositional semantics

Propositional meanings are those represented by well-formed formulae in some system of logical calculus. Each formula represents some statement, whose truth or falsity is explicitly asserted. In such assertions, all context-dependent meanings have been removed : each formula represents a statement standing alone with fixed (propositional) meaning. The theories of formal logic such as the propositiional or predicate calculi have been developed to great depths by logicians (see Mendelson, 1964, for example).

When it comes to defining programming languages by propositional meanings, though, the difficulties are much more formidable. Largely because of their size, as yet no-one has defined a whole language in this way: these meanings have more been used, to date at least, in looking at the meanings of particular programs. For example, many investigators have looked into the problems of proving that a particular program correctly calculates some given function.

The first significant work in this field is that of Floyd(1967). His method, applied to simple flowchart programs, consists of attaching predicates, using the values of the program's variables, to the edges of the flowchart. These predicates are therefore attached between every two successive statements, and at the start and end of the program, and are designed to specify the set of all states that may occur when control passes along that edge of the flowchart. The predicate with the input defines the domain of the program considered as a function, and the predicate associated with the output edge defines the set of all output states that may occur when the program terminates. The assertions represented by these predicates are then proved to have the property that whenever the control reaches any particular point in the program, the assertion attached to that point will be true. Clearly if the assertions have this property then whenever the program stops, the last assertion will be true. Since this last assertion is some statement about the expected results of running the program, it will have been proved that the results do indeed satisfy this statement for allowed inputs.

Floyd(1967) showed that the proof that the assertions attached do have the required property could be done on the basis of local tests. In fact, it is sufficient to show that the assertions attached before and after each statement satisfy some condition related to the semantics of that statement. Ashcroft(1972) calls this condition the "verification condition" for that particular statement. If this verification condition is consistent and complete then it contains a great deal of information about the statement, and Ashcroft(1972) wonders whether it could itself be considered as a definition of the semantics of the statement: "If there is a rule for constructing consistent and complete verification conditions, for all possible statements of a given type then this rule can be considered to be a semantic definition of this type of statement. In the simple flowchart case the execution of any program consists only of the successive execution of statements, and the rules for each type of statement suffice as a semantic definition of the language." He says, however, that this idea of defining a language via verification conditions has not caught on.

As discussed below, there are doubts about the adequacy of this approach for computer science, but it deserves to be developed logically. Most work on the computer science side has been in proofs by 'inductive assertions', as Floyd's method has been named. See, for example, Hoare(1971b) and London(1972) for actual proofs, and London(1971) for a comparison of various such applications.

More efficient proofs result if more general relations are formalised concerning larger-scale steps in the program, as it is a common failing of proof techniques in predicate calculus to get lost in a maze of partial derivations: to be continually rediscovering and misapplying trivial lemmas. Floyd's original method has been developed by Manna(1969,1971), Manna and Waldinger(1971), and by Hoare(1969,1971a): these investigators consider as more important correctness conditions for a whole program, or significant section of a program. That is, they look for input-output specifications for that program,

working from the concept of 'meaning' as a correctness criterion which constitutes a necessary and sufficient condition for the program to realise a given function. By relying less on the picture of assertions attached to edges of flowcharts, it has been found possible to treat languages which at first did not seem amenable to Floyd's method, such as languages involving recursion (Manna and Pnueli, 1970). As well as simple flowchart languages, and simple recursive languages, the method has been applied to non-deterministic programs (Manna 1970) and parallel programs (Ashcroft and Manna 1971). Techniques have been investigated for synthesizing programs from their input-output specification: Manna and Waldinger (1971).

There are reasons, however, why this form of semantic definition - by input-output specifications - is not altogether suitable as a universal normal form for specifying the semantics of programming languages. The proofs must first assume that the program terminates : otherwise it is useless to talk of any input - output relations. Programming languages in general have an undecidable halting problem, so while there are proofs for particular programs that are known to terminate , there can be no general definition of the meanings of all possible programs in some programming language . But one can talk of the meaning of a program even if it does not terminate, because, for example, it may print out intermediate results in a perfectly well-defined manner.

Secondly, input-output semantics regard all programs which realise the same function as equivalent. This is very good for the logician, but the computer scientist is more often interested in the problems of representation of programs in different programming languages, or in the differences of implementation in various systems. These problems are not part of the logical specification of the programs, but they are not totally irrelevant in semantics, and they are especially not irrelevant during the application of semantic theory.

Input-output definitions using propositional semantics are non-constructive definitions : rather than explicitly giving sequences of operations, they only state the essential constraints that must be satisfied by all allowable sequences. In the representation and implementation of programs, on the other hand, the aim is to specify each operation as definitely as possible.

There are many researchers, hoever, who are mechanically minded, and who see propositional meanings as the only type ultimately satisfactory, so the standing of propositional meanings should be expected to improve eventually.

3.4 Functional semantics

Defining a program by functional semantics consists of specifying what partial function the program denotes by defining this function in terms of the functions denoted by the subparts of the program. Compared with the computational method (to be discussed in more detail in section 3.6, but in essence consisting of specifying how the individual computations of the program are carried out), the functional method is much more acceptable mathematically : it gives a true mathematical model of the programming language. This is largely because the theory of functional application is a branch of mathematics which is reasonably well understood, and it is far harder to demonstrate that two machines with different command sequences produce the same result than it is to prove that two functional expressions represent the same value.

The general aim in using functional meanings to define programs is therefore to regard the whole program as an expression - in its most general sense as a function of its constituent variables - whose value is to be found. Because the simplest sort of computer is at least a sequence of commands, the composition of functions needs to be considered, and this composition gives an expression which is the (functional) meaning of the whole program. To define, then, a simple programming language, the rules for constructing the function corresponding to each command could be given. This is done, for example, by Maurer(1972), who regards these functions as a further product of the BNF grammar which defines the syntax of the program. (As seen in section 2.4.2, this extra machinery can be used to check various context-dependent conditions too).Maurer (1972) gives a description of parts of Fortran II with meanings by functions. His subset includes assignments, arithmetic expressions, IF, GO TO, and DO statements, and statement sequences. However, he does not consider subroutines or parameters: these require an extended formalism to represent in functional semantics, and that found most useful has been the lambda calculus.

3.4.1 The lambda calculus

The lambda calculus has been extensively investigated in mathematics (see, e.g. Church,1941) before it was introduced for use in defining programming semantics, first in the LISP language (McCarthy et al., 1962). Wegner(1968, sections 3.5 - 3.10) gives a general introduction to the lambda calculus for computer science.

As a start towards the theoretical semantics of programming languages using the lambda calculus, Landin(1964a) used it for modelling just the arithmetic expressions that occur in programs. That is, he modeled the three ways of constructing expressions: 1) by forming lambda-expressions (e.g. in the case where the expression contains formal parameters, or where an auxiliary function is being invoked) ; 2) by forming operator-operand combinations for infix arithmetic operators ; and 3) by forming lists of expressions for parameter lists. He used the term "applicative expression" to describe the general nature of these constructions, and described an abstract machine (the "SECD") for evaluating the applicative expressions in any particular environment.

Since languages are more than just expressions - containing assignments, gotos and loops too - it is more difficult to describe a full language using functional meanings represented by lambda calculus expressions. Landin(1964b,1965) has to extend the calculus to take imperative features of languages into account. Jumps are now taken care of by treating them as procedure calls - except, however, that this will not work with an 'unnatural' jump into a compund statement where a 'natural' exit is expected. Consider, for example :

```
if b then begin ...; l:s ; ... end else goto l  ;
```

This sort of situation is a problem, too, with other methods of defining Algol-like languages (see also section 3.6.3.1). It is a case of the tree structure of the program being seriously disturbed by by gotos - so Landin(1964b) argues that 'there are some

grounds for blaming Algol 60 rather than our model'.

To deal with assignment, both the notion of applicative expression and the structure of the evaluating machine are extended. This is really admitting that functional meanings are inadequate to describe the full range of assignments and control statements in a modern programming language: because one has to build them into a machine, operating imperatively rather than purely functionally, instead of treating them in a unified manner in the lambda calculus of functions.

But the hybrid system is still meaningful, and is useful for a definition of Algol 60 semantics - essentially by exhibiting how to model constructs of Algol 60 by means of extended applicative expressions. A formal description of the correspondence is given in Landin(1964b,1965), with an 'abstract Algol' as an intermediate step. The formal system is used as a basis for a discussion of alternatives to various Algol 60 concepts e.g. the different parameter mechanisms of call-by-reference as opposed to call-by-name & -value, and some variations on the 'own' concept.

Strachey's(1964) work on semantics by functions has been developed in parallel with the design of the programming language CPL (e.g. see Barron et el.,1963). Unlike Landin, he does not propose extending the lambda-calculus formalism, and he does not need to postulate as an essential part of the basic semantic description of the language any special 'evaluating mechanism' or notional computer. These are avoided by first introducing two basic concepts of the "left-hand value" and the "right-hand value" of an expression. These correspond to "address" and "value" of an assignment operation in a conventional computer, but they are more general, being useable in other situations such as the specification of procedure parameters. With these two concepts, an "abstract store" is defined, which is a function from left-hand values to right-hand values. In the mathematical model of a program, s, the abstract store,

is passed as an extra parameter to every operation. This obviates the need for a separate abstract machine to maintain the state of variables undergoing assignments: by passing the state functions explicitly at each step, all the overwriting or updating is concentrated into a single quantity, and, at least in principle, it is never altered - a wholly new abstract store is formed by each updating operation, without losing the old one. For example, Strachey(1964) defines an update operator U for each left-hand value a, and uses

$$s' = (U(a))(b,s)$$

to construct a new abstract store s' in which a is now mapped onto the new right-hand value b. He then describes a method for associating (compositions of) functions with (sequences of) program commands. In the general case, when a loop structure is present in the sequence of commands, the association will lead to a set of mutually-recursive functions. As yet this method of composition is only informally described, but is moderately comprehensive, being able to handle jumps, block declarations, different modes of calling parameters, etc. Furthermore, it leads to several mathematical results that are useful in programming theory. For example, if it is mathematically possible to construct the 'fixed-point' operator for the group of mutually recursive functions that represent a loop in the original program, then one can remove the recursion, and hence the loop, resulting in a simpler program.

One of the main reasons why Strachey's method is so useful is that he incorporates many ideas of structural and computational semantics (which will be discussed in section 3.6). For example, his 'abstract store' is practically the same as the 'state vector' of computational models, as is the general scheme of representing functions on and to 'abstract stores'. Hence it is not surprising that the two methods - being in essence not too different - share many of their advantages.

3.5 Semantics based on string rewriting rules

Because in the earlier years syntactic theory was much better developed than any theory of semantics, many people tried using syntactic notions in defining all of a programming language, including semantics. Although none of their attempts were eminently successful, I shall discuss them here for completeness, and because some definitions still use them: they have string rewriting rules to describe some syntactic 'preprocessing', midway between the parsing and the semantic operations.

3.5.1 Preprocessing semantics

Many,if not all, practical programming languages contain certain features which, strictly speaking, are redundant. The most well known example is that of the controlled loop: it is easy to write it out in detail using only conditional statements and jumps. So easy, that it is common to define the meaning of loop statements in terms of a string transformation into the language without such statements. See, for example, sections 4.6.4.2 and 4.6.4.3 of the Algol 60 report (Naur et al.,1963), or section 9.3 of the first Algol 68 report (van Wijngaarden et al.,1968). In Algol 60, then, the statement

```
for V := A step B until C do S
```

is mapped onto

```
        V := A
L1:     if (V-C) * sign(B) gt  0 then goto exhausted
        S ;
        V := V + B ;
        goto L1 ;
exhausted:
```

where V is a variable; A,B, & C are expressions; and S is a statement (The syntactic rules defining "variable"s, "expression"s, and "statement"s are included elsewhere in the Algol 60 report).

This technique of defining features of a language by mapping them onto constructs, in a subset, that have the same meaning, has been used extensively, such as by van Wijngaarden(1962,1964), Nivat and Nolin(1964), and Boyle and Grau(1970).

In the scheme of van Wijngaarden(1962,1964), a source program string of Algol 60 is thoroughly pre-processed to convert it into a program in a related but simple language. This string rewriting removes comments, for statements, function designators and type procedures, actual parameters which are expressions (in favour of all parameters being procedure identifiers), conditional expressions, switches, labels, and goto statements. Algol 60, reduced according to such transformations, is seen to contain only a few concepts: (following van Wijngaarden, 1964)

1) some arithmetical and Boolean operations
2) assignment
3) the procedure with or without parameters, call by value, and call by name
4) locality and "own" concept: blocks and declarations.

In his scheme the preprocessing needs only to be done once, before any of the program's data is processed. An alternative scheme, espoused by Boyle and Grau(1970), is to have dynamic text transformations too. That is, the decision to apply a rewriting rule to the program text may be made on the basis of the actual execution of the program. Although this complicates the rewriting, it does allow a purely syntactic treatment of how identifiers retain their values through the various possibilities of block and procedure nesting. With dynamic transformations, therefore, Boyle and Grau are able to follow closely the rewriting-semantics defined in the Algol 60 report concerning procedure invocations. There, invocations are defined by 'copy rules': rules which replace the procedure call by a textual copy of the procedure body, and which specify the renaming of identifiers if necessary to avoid clashes. What Boyle and Grau have done is to formalise these rules for copying and renaming so that they can write down algorithms for these run-time textual transformations. These algorithms can be applied to blocks as well, to rename all occurences of identifiers declared within them, so that the end result is a program text for interpretation which has no problems of identifier scopes because each variable used has

an identifier, which they call its 'canonical identifier,' uniquely its own. In the appendix of their paper (Boyle and Grau, 1970) they give an example of transforming a recursive function to canonical form. Since the factorial function always terminates for finite input, only a finite number of rewriting string transformations are necessary in their example: even for recursive programs.

Wegner(1970, section 5) also discusses the copy rule model for identifier accessing, and it has been often used to define the invocation of procedures and functions (see for example van Wijngaarden et al.,1968, section 8.6.2), following the idea of the Algol 60 report.

Nivat and Nolin(1964) describe another way of defining Algol 60 programs by transforming them into simpler programs. However, they convert them into sequences of simple assignments and condition-instructions which use an explicit program stack: much as any compiler would do to convert programs of a language that allows recursion into a low-level language to run on a von Neumann-like machine with linear store, indirect addressing, and all. And as Boyle and Grau(1970) remark, since Nivat and Nolin do not explicitly state the general algorithms for transforming Algol 60, it has not been shown whether their target language adequately supports the more difficult aspects of handling identifiers and procedures etc. Furthermore, Boyle and Grau are not convinced that pushdown stacks with explicit linkages, indirect addressing, etc., are exactly suitable for a simple language into which Algol 60 may be transformed in order to define it: they regard their own work as an attempt to avoid these (and other) complexities.

Considering again the general technique of string transformation rules to define the more complicated parts of a program semantics, the works of van Wijngaarden et al.,1968 (e.g. section 8.6.2: "calls"), Wegner(1970), and Boyle and Grau(1970) show the elegance of the method at least for the copy rule model. But not so elegant are some of the (very ingenious) devices van Wijngaarden(1964) used to remove functions, labels, and

jumps etc. For example, I consider to be only sophistry replacing all jumps uniformly by procedure calls (to procedures which never return) rather than have one extra primitive operation.

String transformations can thus lead to readable and rigorous models of certain aspects of programming language semantics. Some other methods - such as the computational definitions - will try to do more: they would consider in much more detail the precise operational steps. A practical implementation, also, cannot efficiently rely on text transformations at run-time - the rewriting rules discussed in this section provide only an abstract model for the operations underlying all implementations. So to some extent syntactic models (such as the copy rule model) and computational models are complementary rather than mutually exclusive. Wegner(1970), for example, treats the copy rule model as the primary definition, and then goes on to discuss a variety of equivalent implementations of that model.

3.5.2 Data-processing by string rewriting rules

Markov algorithms were originally introduced to help define and investigate problems in computability theory (Markov,1962), but the transformation scheme present in them has found several applications in defining programming languages. This is by regarding the program as guiding the rewriting of input strings, through all intermediate results, into output strings: the meaning of the program is held to be defined by these rewritings.

Applications to semantic theory were first made by van Wijngaarden(1962,1964), and Caracciolo(1963,1966), using an extension of the original Markov algorithms. In ordinary Markov algorithms there are transformation rules, the left and right sides of which are sequences of symbols over some given alphabet. To find out whether a rule is applicable to an input sequence of symbols, the sequence is scanned for the occurence of a subsequence which is identical to the left-hand side of the transformation rule. If there is such a subsequence, its first occurence is replaced by the right-hand side of

the rule concerned. For example, given the rules '0+1=1' and '1+1=2', the input string '1+0+1' will, on applying the first rule, become '1+1', and then '2' with the second rule being applied.

In the extended formalism, the transformation rules contain not only terminal symbols, but also linguistic variables. Van Wijngaarden(1964) uses Backus notation to represent these variables, and so he can define rules such as '<di1> + <ui2> 0 = <ui2><di1>', and '<ui1>+-<ui2> = <ui1> -<ui2>'. There is a corresponding extension to the algorithm for applying such rules, in conjunction with production rules to define which values are possible for the variables. By using a large number of these extended rewriting rules, with suitable variables, one can define the rewriting-semantics of data-processing operations. For example, van Wijngaarden(1964) gives 44 rules for the addition and subtraction of integers, as strings of decimal digits of arbitary length.

De Bakker(1969) describes how the ideas of van Wijngaarden(1964) were used as the base for de Bakker(1967):

"An almost complete definition of Algol 60 was given (the only feature not treated being real arithmetic), consisting of about 800 transformation rules. The meaning of an Algol 60 program is determined by the way in which it is transformed by these rules. Here another extension of the Markov algorithm scheme not yet discussed is of importance, viz., the possibility of having a dynamically growing list of rules. The execution of a particular Algol 60 program will lead to the extension of the list of language-defining rules with rules which reflect the meaning of this specific program. For instance, the occurence of the assignment statement a := 3 in a program causes the creation of a new rule a → 3 (omitting some details on locality), which will be applied each time the value of a is needed subsequently in the execution of the program. De Bakker(1967) also gives a precise definition of the formal system used, illustrated by several examples, and an implementation of an abstract machine for interpreting it."

There is the meta-rule, incorporated in the abstract machine, that if two or more rules are applicable at a given stage, then the rule nearest the end of the rule-list is used. Thus the variable list of rules is analogous to a machine's stack of values.

Overall, because the basic definition method is so simple, the total definition of a language like Algol 60 is very longwinded, the treatment of some features - such as go to statements, and identifier localities - requiring some ingenuity. And there must be an easier way than having 800 transformation rules, the list of which must be searched at each step.

3.6 Semantics with structures and computations on structures

This type of meaning is the one most often used in computer science. For computers are very much concerned with representations, structures, and operations on structures. On the whole, one is interested in formalisms which closely model, even if abstractly, the style of computation of actual digital computers, and in the detailed study of relations among representations and implementations.

3.6.1 Information structures

The primary concept in structural semantics is that of "information structure", or "state vector" : these are objects esed to describe the state of a machine (abstract or actual) at a given instant in the execution of its program. Following McCarthy(1962), one says that 'the meaning of a program is defined by its effect on the state vector'. Maurer(1972) calls this assertion the "state vector thesis", and formulates it, with respect to digital computers, as follows:

'There exists for each program a set V of variables, whose values at any stage of the computation depend only on their values at the previous stage; and the entire meaning of the program is determined if we know its effect f, where, if I is an assignment of some legal value to each variable in V before the program starts, f(I) is the corresponding assignment after the program has finished.'

This model has been applied by McCarthy(1964) to the description of a subset of Algol, and to a proof of correctness of a compiler for a simple class of arithmetic expressions (McCarthy and Painter, 1967).

When structural meanings are compared with other types, it is seen that they are the most general type which can be represented externally. For example, one can write down a tree, lattice, network, vector, or any other sort of structure, and define meanings by relations within, and transformations of, this structure. In mathematical logic, these types of structural meanings

are called "intentional", as against "extensional". That is, meanings are implicit as relations rather than explicit as true-or-false propositions. Although formal logic proof techniques cannot be readily applied to them, structural meanings have many practical advantages. Because the meaning of a particular feature is largely tacit - essentially being its relation to its context - with careful design it is much easier for us to understand the relevance of that feature : that part of its meaning which is given explicitly. For to describe its meaning using, say, propositional meanings, the context, i.e. the state of the program, must be explicitly represented as a precondition within each propositional formula. Proofs, both manual and automatic, would tend to be verbose and longwinded, and would easily obscure the relevance of the particular feature being considered.

Wegner(1970,1971) formalises these ideas of information structures and computations as follows:

An information structure model is a triple $M=(I,I^0,F)$ where I is a countable set of information structures (structured states), I^0, a subset of I, is a set of initial representations, and F is a finitely representable set of unary operations (primitive instructions) whose domain and range is a subset of I.

Most concern is with the special case of

A deterministic (sequential) information structure model is one which for all I_j in I has at most one element f in F applicable to I_j.

A computation in a deterministic information structure model $M = (I,I^0,F)$ is a sequence $I_0, I_1, \ldots$ of elements of I such that I_0 is in I^0, and for $j=0,1, \ldots$ $I_{j+1} = f(I_j)$ for some f in F.

A terminating computation is a computation which, for some integer n, generates an I_n to which no element f in F is applicable.

A complete programming language definition requires the sets I^0 and I of information structures to be specified in a 'syntactic metalanguage', and the set F of primitive instructions to be specified in the 'semantic metalanguage'. Syntactic definition mechanisms such as the BNF notation of the Algol 60 report in fact only accurately define

the set I^0 of initial representations, where the syntax tree of the program is regarded as the input, along with any data, to the computation. On the other hand, definitions such as those in the Vienna Definition Language VDL (Lucas,1968 , Wegner,1972), which are more oriented toward semantics, rather than just syntax, specify both the set I^0 of initial representations and the set I of computational states, using the one meta-language. The VDL metalanguage was specially designed to specify tree structures of the same style as the 'abstract syntax' described in sections 2.4.3ff. When applied to ALEX, there is the VDL description of all abstract syntaxes (initial representations) in Appendix C, and of all computational states in Appendix H. This unity of structural description metalanguages is one of the great advantages of programming language definitions using VDL, and makes the definition of a modern, complex language such as PL/I (Alber et al.,1969, Walk et al., 1969) somewhat less difficult to understand. Semantic languages will be discussed in the next section, but I remark here that one semantic metalanguage is used in both the syntactic translation (from concrete to abstract syntax) and the semantic computations. Compare, for example, Appendices D and L, which are the VDL definition of the translation and interpretation phases, respectively.

3.6.2 Semantic metalanguages

Semantic metalanguages are those in which the basic operations are specified: these are the operations 'F' of above that are the primitive instructions for transforming the current information structure into its successor, and so defining the computational semantics of the current section of the program being executed by means of its effect on the state structure.

3.6.2.1 The Vienna Definition Language

The Vienna Definition Language (VDL) specifies the state transitions in F in terms of conditional expressions of the form

$$f = p_1 \rightarrow a_1 ; p_2 \rightarrow a_2 ; \ldots ; p_n \rightarrow a_n$$

where for $i=1,2,\ldots,n$, p_i specifies a predicate to be satisfied by the current state I, and a_i specifies the state transition if p_i is the first true predicate of the conditional expression: for the first i, p_i implies $I' = a_i(I)$.

The actions a_i are either macro 'calls' to a sequence of other such conditionals f (these calls having parameters usually), or are one of a small number of primitive operations which return values, and may, as a 'side' effect, modify other components of the state. The macro instructions may be viewed as syntactic devices for easily grouping frequently used sequences of operations, such as procedure calls etc., into the tree structure, whereas the value-returning instructions perform the basic structural transformations which constitute the semantics in terms of the computation at the level of expression evaluation, assignment, storage allocation, etc. To construct and modify the tree structures - which are the heart of a VDL definition - there are used the u_0 and u operators, respectively, as has been described in section 2.4.3.1.

Having a metalanguage with the scope and precision of VDL was a decided step forward in semantic theory. Of the long-range goals mentioned in section 3.1, the invention of VDL has helped in work toward many of them. Allen(1972) has considered the use of computational definitions in investigating the properties of specific programs and of specific language constructs in their contexts. For this goal, he considers axiomatic definitions to be more useful than abstract interpreters, so he tries to derive exact axioms about parts of the state structure from the algorithmic model. These axioms, which are independent of any implementation model, are combined with further propositions about a given program and then used in proving more about the program. His work uses propositional semantics to talk about the particular program, but computational semantics is the standard starting point for defining all programs in a language.

Concerning the theory of implementation, having an abstract interpreter written in VDL at least indicates some practicability of implementation, although an abstract definition can take liberties and, for example, include implicit searches (e.g. rule I20 in Appendix 1), or other schemes inefficient in practice: e.g. the idea of stacking the whole environment and control trees in the 'dump' component. We should remember, however, that the abstract interpreter is more often only the formal standard: although it is a model of a possible implementation, usually it is the reference for the correctness of other implementation models. Thus Lucas(1968) considers another realisation of the PL/I block concept, and proves its equivalence to that of the standard (Walk et al.,1969). Wegner(1972, section 5) and Jones and Lucas(1971) also consider proofs of interpreter equivalence.

Another aspect of implementation theory is considered by Lucas(1970). While VDL uses the "tree" as its basic structure, an implementation can often well use structures such as stacks, directories, storage arrays, etc. Lucas (1970) looks at these various 'software devices', as he calls them, and considers whether they could be usefully

included as primitives in the definitional method. It is hoped that these formal studies of machine structures (see section 3.6.3.1, and also Bekic and Walk,1971) will help in the mechanised design of implementations and perhaps of compilers.

However,VDL definitions are not the ultimate in semantic descriptions. One immediate, and usually overriding, objection is that the Vienna Definition language is very difficult to read and understand. The VDL definition of PL/I is not widely known for its elegance, clarity, readability and down-to-earth sanity. This problem has been discussed before, in section 2.6, and it was found that one of the difficulties with VDL is that it violates what may be called a 'principle of proportional understanding'. That is, it requires a lot of work in coming to know the VDL scheme and the structures it works with, before even the simplest question about a programs meaning can be answered.

3.6.2.2 The BASIS/1 definition of PL/I for ECMA/ANSI.

To try to overcome this major deficiency of VDL definitions, a more recent definition of PL/I (the ECMA/ANSI BASIS/1 definition,version 12, July 1974) has used an English-like metalanguage in which to write the algorithms etc. for the semantics. The operations are expressed in a semi-formal programming language which uses the grammatical flexibility of ordinary English prose, while at the same time attaching precise meaning to certain words and phrases, in order that the flow of control and the tree manipulations be well defined. For example, the instructions "replace", "let","append","attach", and "delete" have specified computational meanings for tree operations, and instructions such as "perform", "go to", "if", "for each", and "case" etc., are used to direct the flow of control along the steps constituting the defined operations (see section 1.2.3 of the BASIS/1-12 document for their exact specifications).

It may appear that using English as a semantic metalanguage is a retrograde step in progressing towards completely formalised language definitions. Certainly

such definitions cannot be mechanically checked, converted, implemented, or otherwise automatically processed, but we should remember that no-one has yet succeeded in mechanically processing the definition of a whole programming language written in any other semantic metalanguage either. To date, nearly all uses of a formal definition have involved passing through personal knowledge before they can be checked, implemented, etc. To the formalist, this merely indicates the primitive state of semantic theory, but to those who want to read, understand, and use the definition in practical ways it indicates that some compromise of complete formalisation would be profitable. Unlike the formalists, they do not see unbearable chaos or vagueness necessarily following this move; indeed, considerable precision is attainable, as in the BASIS/1 method introduced above, and used to define ALEX in Appendices J,K, and L. This precision is aided by the fact that the operations work on precisely defined objects (a completely formal notation is used to describe trees in accordance with their syntactic definitions e.g. as in Appendices J and K) and because the certain words and phrases mentioned earlier have precisely defined structural meanings. Perhaps the resulting definitions should be called 'semi-formal.'

3.6.2.3 The Algol 68 method

Another definition which uses a semi-formal English metalanguage is that of the Algol 68 report (van Wijngaarden et al.,1968,1974). The degree of formality in the English of this report is much the same as in the BASIS/1 definition of PL/I for ECMA/ANSI, with a large number of words and phrases being specifically defined: sunch as "environ","notion", "proto-", "meta-", and "hyper-notions", "designate", "produce", "elaborate", "envelop", and many others. However, for two reasons pricipally, the BASIS/1 definition is both more precise and more readable than the 'semantics' sections of the Algol 68 report. More precise, because as well as having a formally defined syntactic structure to work from (i.e. abstract syntax, or I^0 of section 3.6.1),

the set of all possible machine states (all the I values of section 3.6.2) is also formally defined (e.g. see Appendix K for the set of machine states allowable during the interpretation of ALEX programs). The Algol 68 report, on the other hand, only describes a hypothetical machine and its structures using a semi-formal English, with concepts such as "objects", "names", "values", "to refer to", etc., for data structures, and "scopes", "locales", "environs", and "scenes" for program structures. While such natural-language descriptions are desirable and useful when a new abstract model is being introduced, it would help the definition in many ways if these computer structures were formally described as well. Such descriptions are not difficult because they are usually small compared with the later full semantic specifications, and because large data structures are becoming common in programming.

Secondly, the BASIS/1 definition is more readable than the Algol 68 report because it is set out more like a computer program. Both definitions intend to give rigorous algorithms for the interpretation of programs along with their data, but the former method gives a much more explicit exposition of these algorithms. It uses computer-programming-like features, such as subroutine operations with parameters and values being returned, and its layout of steps and cases within operations - while similar in some ways to that of the Algol 68 report - uses less English, and is easier to follow.

3.6.2.4 Linked-Forest Manipulation Systems as semantic metalanguages

In all of the computational-metalanguages described above, in section 3.6.2, one has had the capacity of defining arbitarily complicated functions, transformations, etc., on the information structures. This is because these semantic metalanguages are organised very much like computer programs, with no restriction on the complexity of tests, expressions, loops and procedure calls and so on that can be included. This power and complexity is almost certainly necessary when defining an intricate and epic language such as PL/I (e.g. the BASIS and VDL definitions), and to a lesser extent when defining Algol 68 (which is much more cleanly organised). However, our example language ALEX is simpler still, and it is possible to define the information-structure operations of its semantics using a metalanguage without the capacity for arbitarily complicated functions.

The metalanguage in question is Čulík's 'linked-forest manipulation systems' (LFMS), which have already been discussed, with respect to defining context-dependencies in the syntax, in section 2.4.3.2. In his 1973 paper, Čulík gives a LFMS definition of his dialect ALG of Algol 60, and this example has been followed where possible in the LFMS definition of ALEX semantics in Appendix N. This means that his scheme of handling identifiers, declarations, and data cells has been repeated: compared with the BASIS and VDL definitions, in this abstract syntax there are no declarations to be executed, for the translator has already made declarations of the program into 'cells' for the abstract syntax and for the machine execution. Variables are 'cell-designators' (or 'parameter-designators'), so they link directly to their values, without the need for environment or denotation directories of the VDL, and subsequently of the BASIS, definition. This simplifies many aspects - variables in function arguments, for example, keep pointing to their proper values even if

new variables with the same name are declared - but at the cost of destroying any distinction between abstract syntax, which is usually read-only, and machine structure, which can be changed at will. One structure is now used throughout the execution, so a procedure-call, for example, copies the procedure body and then adds it to this structure (Appendix N rule L16), and then one must carefully provide for selective deletion at exit (rule L24) so that the call statement can be reused. There is no simple guarantee of the integrity of the program: in complete contrast to the policy of the BASIS definition, according to which the abstract syntax is never changed, never copied even: only refered to by 'designators' in the program-state . The VDL definition does copy the syntax, because it does not have designators, only selectors, so it cannot have pointers to an arbitary statement, only to an arbitary immediate component of some given node (cf. the discussion in section 2.4.3.2).

In form, a LFMS definition, e.g. of Appendix N, looks much like a Markov algorithm (cf. section 3.5) : it is a list of transformation rules 's → t' where s & t are arbitary sequences of symbols standing for types of nodes and their substructures, however, rather than strings. Whenever the information structure has a sequence 's', its first occurence is replaced by the sequence of structures 't' (both sequences have the same number of structures in a valid rule). Rules can be selected nondeterministically, but a system of control words is employed to mostly tame the nondeterminism: words such as "EXEC" or "END" or "VAL" may be dynamically attached to any node, and the rules can test for their presence. Also an LFMS can use the same mechanism for sequencing the application of rules as is used in Rosenkrantz's (1969) 'Programmed Grammars' (section 2.3.3), but in the ALEX semantic definition this is only employed in rule L2 (Appendix N), to make sure the computation ends when the program has exited. These two measures do not eliminate all nondeterminism: it remains, for example , in rules L17 and L29, where parameters, and parts of expressions, respectively, are evaluated in parallel.

The power of an LFMS definition, compared with the Markov algorithms of section 3.5, is greatly enhanced by the fact that an LFMS operates on trees (and 'forests', which are set of trees), rather than linear sequences. And furthermore these are linked trees and forests : nodes can be 'designators' which point across the tree to designate any other node. Thus the tree structures , with designators, are very similar to those of the BASIS definition . But the algorithms which operate on the trees are quite different in style: whereas the BASIS definition resembles a computer program, with local variables and strict control of the flow of calculations, the LFMS definition uses a kind of free-format structure matching. Most of the time this structure matching, however, is straightforward because usually there is only one "EXEC" control word in the tree, and instructions require this control word to begin. Any other nodes are reached from the EXEC-labeled node, e.g. by designators, so there is no need for a very general set of structure-matching algorithms to interpret LFMS definitions.

As an example of an LFMS rule, consider rule L21 of Appendix N :

L21 variable=parameter$_1$-designator
& parameter$_1$: name,type,argument=expression →
expression-copy

The left-hand side of this rule is the precondition: here, a 'variable' is required which is a 'parameter' designator, the parameter designated having as components a "name" string, a 'type', and an 'argument' (identically an 'expression'). If this precondition is satisfied, this means that the "variable" node is replaced by a copy of the expression. As part of the ALEX definition, the rule means that a variable designating a call-by-name parameter representing an expression is replaced in its context by a copy of that expression (this closely follows the copy-rule model for call-by-name parameters).

Another example is the rule

L14 EXEC goto-stmt: label$_1$-designator
& label$_1$: stmt$_2$-designator →
goto-stmt:label$_1$ -designator & EXEC stmt$_2$

In the ALEX scheme, the control word "EXEC" is the precondition for a statement's starting execution, just as "END" is used to signal the completion of a statement so its successor can be EXECuted (e.g. rule L6), and as "VAL(x)" is the control word indicating that the subsumed expression has been evaluated to yield the value x. Hence the rule above transfers control (transfers EXEC) from the goto-statement to the statement designated by the label-value denoted by the label of the goto-statement, and leaves that statement in its normal state, ready for subsequent use. This rule, however, does not have the same effects as the corresponding rules of the VDL and BASIS definitions. While it is perfect for 'local' gotos, it gives very interesting results when applied to jumps which lead outside the current block or procedure. This is largely dictated by our inability to define more complex algorithms in the LFMS semantic metalanguage. Without perhaps making the whole definition very much more complicated, it is difficult to define the routines required for a procedure to delete itself when it exits,and also to delete any block copies between itself and the block containing the destination label. In the existing scheme it is the calling block which deletes the called procedure; the called procedure cannot delete itself, without the LFMS formalism being extended to allow more general structure-matching operations (e.g. notation to represent a variable tree-depth in the precondition of a rule).

3.6.3 Hypothetical machines

Many formal definitions of programming languages use some kind of (hypothetical) machine, to define constructively the detailed effects of a program in that language. For example, there are the definitions of PL/I (Alber et al.,1969 and ECMA/ANSI,1974), Algol 68 (van Wijngaarden et al.,1968,1974), Basic (Lee,1972), Euler (Wirth and Weber,1966), and a subset of Fortran (Maurer, 1972). These hypothetical machines, as well as any compilers or interpreters using them, should be more abstract than realistic: the definitions should not be involved with the many details of concrete machines: these are irrelevant to the original language because one wants the final definition to accurately reflect only that language, and not a host of other contingent details of some particular machine. This means that, whatever level the machine is programmed at, an interpreter, for example, should be independent of the idiosyncrasies of particular real machines if it is to be put forward as a standard definition. Garwick(1964) proposed defining languages by their compilers, but even so his "MIC" model used an assembly-language of very bland characteristics (his ideas, however, did not meet with a great deal of enthusiasm: partly for reasons similar to those in section 3.2.2).

A definition with a hypothetical machine cannot be regarded as complete unless some reasonably precise description of that machine is given too, otherwise the difficult problem of defining the basic actions is only postponed, without a solution being properly attempted. Of course, we must remember that any formal system has primitives that cannot be defined by the system and so must be specified elsewhere. And there are degrees of postponement: as the primitives are made more basic, the importance for the whole definition of specifying them formally, rather than intuitively, decreases.

The amount of detail given to define a hypothetical machine has varied widely. Normally one would not expect

a very extensive formal definition for it, following the principle that 'the metalanguage should be simpler than the language' (cf. section 3.2). It all depends on how novel the machine (or metalanguage) is, and how much work one is prepared to put into the whole definition. For example, the machine used by Wirth and Weber(1966) for defining Euler is sufficiently simple that it might be considered self-explanatory, but the abstract machine for supporting Vienna Definition Language programs, on the other hand, had many unconventional features, and in Lucas(1968) a substantial, if only semi-formal, description is given for the machine's representation and operation. However, Wegner(1972) does give a definition of VDL in VDL: there is then a kind of consistency check on any actual interpretation of the VDL primitives.

3.6.3.1 Machine structures

What kinds of structures are built into the machine to represent states during the program's execution or interpretation? Is the machine essentially cellular, linear, tree-like, or does it support arbitary structures? Ideally these should match the kinds available in the language being defined: so that, for example, the Basic definition of Lee(1972) presupposes a linear storage, whereas the Fortran definition of Maurer(1972) needs only cellular storage (he does not include arrays in his subset); and the more complex Algol 68 uses general 'objects', 'multiple values', 'structured values', and 'names' in its hypothetical machine, so that general structures can be constructed however the users' programs require.

There is not always, however, a perfect match. For example, the PL/I machine of the VDL definition (Alber et al.,1969), though essentially tree-structured, was required to support "goto" operations, and handle array attributes such as "aligned", "connected", and other archaic legacies of the IBM 360 architecture. The definition had great difficulty in dealing with

these conditions - which are essentially linear and not hierarchical - within the architecture of a tree-structured machine. It got around the problem by giving only several axioms which must be satisfied by any implementation, and leaving open the details after that. The definition then makes 'unknown' in the standard the result of many operations, for example, those on arrays declared with the "defined" construct (cf. the "equivalence" construct in Fortran), even for simple cases that would give the same result in all implementations. For example, the PL/I segment

```
DECLARE A(10),
        C(3) DEFINED A(5);
C = 6 ;
```

results, according to Walk et al.(1969), in all of the array "A" being made "unknown", following the rule that "since no relationships between parts of value representations are defined, an assignment simply makes all those parts unknown which are not independent of the part to which the assignment is made (with the exception of this part itself)." (Alber et al.,1969, section 4.2.2). In the above example, this means that since no relations between the "A" and "C" values are defined in the standard definition, the assignment makes all the "A" values "unknown" except those which happen to be identical to "C" values.

Goto operations are another example of a mismatch between the structures implied by the language and those supported by a machine such as that of VDL definitions. In fact, many would argue against the use of goto-statements in well structured programs because they destroy the correspondence between the linguistic structure of the program and the dynamic structure of the computation : these are exactly the same reasons why goto-operations are difficult to support in a tree-structured machine. For such a machine, the tree structure implies that the effect of any particular node is defined in terms of the effects of the subphrases of that node: this is the usual structure

for VDL definitions (see for example Appendices C and I). However, as Lucas(1970) puts it:

"Goto statements have the following consequence: Parts of an expansion may become irrelevant if one of the subphrases happens to be a goto statement. One possibility to make an expansion irrelevant is to delete it. In order to be able to show the deletions, the parts potentially to be deleted must be parts of the state. Thus, we keep copies of program-parts in the state for the sole purpose of being able to delete them." (p45)

ALEX does have goto statements similarly, and to manage them I have used a scheme closely following the PL/I definition, of which Lucas was talking just above, and which is described in Alber et al.(1969, section 9.6). According to the scheme, each block component contains a tree structure "control information" (see specifications S1 & S4 of Appendix H), each level of which contains a copy of the program text, copied from the abstract syntax, as well as an index indicating which statement or group within this program segment is currently active. A goto statement, even a local one, will involve trimming this tree sufficiently (rule I20: 'goto-2' of Appendix I) so that it can be built up again (I22: 'goto-3') to point to the destination statement somewhere else in this block (I23: 'goto-4'). In this way, one can support even the difficult case mentioned earlier (section 3.4.1) of

if b then begin ...; l:s ; ... end else goto l ;

Here the 'building up' of the "control information" is into the begin-block, so that the statement s is reached. This building up involves copying the text for that block, and an index to s, into the "control information" tree. One cannot simply point the program control to s because one must arrange for a normal flow of control out of these levels of statement nesting that have been built up.

Ideally, the operation of a tree-structured machine should follow the tree structure of the abstract syntax

of its program, and there should need only be one read-only copy of this program since high-level-language programs should not alter themselves. The mismatch of structures resulting from having to support goto operations on such machines means that there have to be copies of the program in the dynamic machine states, where they can be changed or deleted, etc., and that one cannot have an elegant interpreter operating with simple recursive functions. The EPL interpreter of Lucas et el.(1968) does not support gotos, and its flow of control is much more simple than that of the ALEX interpreter in Appendix I, because it can use recursive functions everywhere, even within conditional statements such as the example above: it has no goto statements to break the match between the structures of the machine and of its programs.

3.7 Conclusion

The semantics found in this thesis to be most suitable, in terms of the aims listed in chapter 1, for defining full-scale programming languages is that using structures and computations on structures. For these structural meanings a number of semantic metalanguages have been devised, the most comprehensive being the Vienna Definition Language, the language of the Algol 68 reports, and the language of the BASIS/1 definition of PL/I. The Vienna Definition Language is the most mathematically oriented of the three, striving to represent all structures, functions, and primitive operations by an algebraic notation. While still regarding rigour as extremely important, the later two languages aim also at readability: they use English much more, but they do give certain words and phrases precise computational meanings. Comparing these two, I consider, for reasons discussed in section 3.6.2.3, that the semi-formal English of the BASIS/1 definition is to be prefered, where precision and readability are concerned, to that of the Algol 68 reports.

Where the emphasis is not on such comprehensive languages, and one does not have to define arbitarily complicated functions and conditions, the Linked-Forest Manipulation Systems can give concise, even elegant, definitions, e.g. of ALEX semantics in Appendix N. Note that this language does not have arrays, loop statements, or label expressions, neither does it have any features requiring intricate or specialised semantics such as formatted input-output or mathematical library functions etc., and in Appendix N this simplicity of language is well matched to the capabilities of the Linked-Forest Manipulation Systems.

The other metalanguages for semantics I have not found to be so successful for the situations considered in this study. Natural semantics, for example those in natural languages or in compilers on real computers, although perhaps where our ideas start and where implementations end, are not the most suitable means for formal definitions. This is because one would prefer definitions which are complete, precise, and readable, and other metalanguages,

especially those mentioned earlier in this section, are improvements in most of these respects.

The other types of semantics do not have the full power of computational semantics, and have correspondingly restricted applications. Propositional semantics is most useful for defining the effects of particular programs (i.e. not the set of all programs of some language), and even then only certain classes of programs (e.g. programs which can be proved to terminate) can be properly treated. The range of applicability, though, is being increased by further research, so one should not be bound too much by one's first impressions.

Pure functional semantics is another type not suitable for defining certain classes of programs: those with unavoidable assignments or transfer statements. However, unctional semantics can be extended, e.g. as in Strachey(1964) or in Aiello et al.(1974), to include these more computational features. But then one finds that, in effect, one is using computational semantics, and one benefits accordingly. There is not a sharp distinction between functional and computational semantics in this case: the same written expression, e.g. in the lambda calculus, can be interpreted either computationally or functionally.

Semantics based on string rewriting rules can only with great difficulty be extended to include all the constructions of a programming language such as Algol 60. They are most useful for defining, by syntactic preprocessing, certain features of a language in terms of more basic statements: such as the definition in the Algol 60 report of "for" statements, or the various parameter mechanisms. For nearly all other aspects of programming languages, the other types of semantics, especially the structural semantics, have been found, in this study, to be better in terms of the criteria adopted for their comparison.

References

AIELLO, L. et al.(1974)
The semantics of Pascal in LCF. AI memo 221, Stanford University.

ALBER, K. et al.(1969)
Informal introduction to the abstract syntax and interpretation of PL/I. TR 25.099 IBM Lab. Vienna.

ALLEN, C.D.(1972)
Derivation of axiomatic definitions of programming languages from algorithmic definitions. SIGPLAN notices 7,1 pp15-26.

ASHCROFT, E.A.(1972)
Program correctness methods and language definition. SIGPLAN notices 7,1 pp51-57.

ASHCROFT, E.A. and Z. Manna(1971)
Formalization of properties of parallel programs. in Machine Intelligence 6, Edinburgh University Press.

BAKER, J.L.(1972)
Grammars with structured vocabulary. Information and Control 20 pp351-395.

de BAKKER, J.W.(1967)
Formal definition of programming languages. Mathematical Center Tracts Vol. 16, Mathematisch Centrum, Amsterdam.

de BAKKER, J.W.(1969)
Semantics of programming languages. pp173-227 in Tou(ed.) Advances in Information Systems Science, Vol 2, Plenum P.,1969.

BARRON, D.W.(1963)
The main features of CPL. Computer Journal 6 pp134-143.

BAUER, H.R. et al.(1968)
Algol W language description. Stanford University Rep. CS 110.

BEKIC, H. and K. Walk(1971)
Formalization of storage properties. pp28-61 in Engeler(ed.) Symposium on Semantics of Programming Languages, Springer-Verlag, 1971.

BOYLE, M. and A. Grau(1970)
An algorithmic semantics for Algol 60 identifier denotation. J. ACM. 17,2 pp361-382.

CARACCIOLO, A.(1963)
Some remarks on the syntax of symbolic programming languages. C. ACM. 6,8 pp456-460.

CARACCIOLO, A.(1966)
Generalized Markov algorithms and automata. pp115-130 in Automata Theory, E.R. Caianiello(ed.), Academic P.,1966.

CHURCH, A.(1941)
The calculi of lambda-conversion. Ann. Math. Stud., No. 6, Princeton Univ. Press, 1941,1951.

ČULÍK, K.(1973)
A model for the formal definition of programming languages. Int. J. Computer Mathematics 3 pp315-345.

DUNCAN, F.G.(1964)
Our ultimate metalanguage. pp295-299 in Steel(ed.)(1966)

ECMA/ANSI (1974)
The BASIS/1 definition of PL/I. Version 12, July 1974.

FELDMAN, J. and D. Gries(1968)
Translator writing systems. C. ACM. 11 pp77-113.

FLOYD, R.W.(1967)
Assigning meanings to programs. pp19-32 in Proc. of Symposium in Applied Math. Vol 19, Amer. Math. Soc.

GARWICK, J.V.(1964)
The definition of programming languages by their compilers. pp139-147 in Steel(ed.)(1966)

GREIBACH, S. and J. Hopcroft(1969)
Scattered context grammars. J. Comp. Syst. Sci. 3 pp233-247.

HOARE, C.A.R.(1969)
An axiomatic basis for computer programming. C.ACM. 12 pp576-580.

HOARE, C.A.R.(1971a)
Procedures and parameters: An axiomatic approach. pp102-116 in Engeler(ed.)(1971).

HOARE, C.A.R.(1971b)
Proof of a program: FIND. C. ACM. 14,1 pp39-45.

IBM(1970)
PL/I language specifications. Form no. GY33-6003-2.

HOPCROFT, J.E. and J.D. Ullman(1969)
Formal languages and their relation to automata.
Addison-Wesley, Reading, Mass.,1969.

JONES, C.B. and P. Lucas(1971)
Proving correctness of implementation techniques.
pp178-211 in Engeler(ed.)(1971).

KNUTH, D.E.(1968)
Semantics of context-free languages. Math. Syst. Theory
2,2 pp127-145.

KNUTH, D:E.(1971)
Examples of formal semantics. pp212-235 in Engeler(ed.)(1971).

KOSTER, C.H.A.(1969)
Syntax-directed parsing of Algol 68 programs.
pp61-69 in Peck(ed.) Proc. Informal Conf. on Algol 68
Implementation, Univ. Brtish Columbia, 1969.

KOSTER, C.H.A.(1971a)
Affix grammars. pp95-109 in Peck(ed.) Algol 68 Implementation,
North-Holland,1971.

KOSTER, C.H.A.(1971b)
A compiler-compiler. MR 127/71, Mathematisch Centrum, Amsterdam.

LANDIN, P.J.(1964a)
The mechanical evaluation of expressions.
Computer Journal 6,4 pp308-320.

LANDIN, P.J.(1964b)
A formal description of Algol 60. pp266-294 in Steel(ed.)(1966).

LANDIN, P.J.(1965)
A correspondence between Algol 60 and Church's lambda notation.
C. ACM. Vol 8, nos. 2 pp89-101, 3 pp158-165.

LEDGARD, H.F.(1974)
Production Systems: or, Can we do better than BNF?
C. ACM. 17,2 pp94-102.

LEE, J.A.N.(1972)
The formal definition of the BASIC language.
Computer Journal 15,1 pp37-41.

LONDON, R.(1971)
Experience with Inductive Assertions for proving programs
correct. pp236-251 in Engeler(ed.)(1971).

LONDON, R.(1972)
A correctness proof of the Fisher-Galler algorithm using Inductive Assertions. pp125-135 in Rustin(ed.)(1972).

LUCAS, P.(1968)
Two constructive realizations of the block concept, and their equivalence. TR 25.085 IBM Lab. Vienna.

LUCAS,P.(1970)
On the semantics of programming languages and software devices. pp41-57 in Rustin(ed.)(1972).

LUCAS, P. et al.(1968)
Method and notation for the formal definition of programming languages. TR 25.087 IBM Lab. Vienna.

McCARTHY, J.(1962)
Towards a mathematical science of computation. IFIP 1962 pp21-28.

McCARTHY, J.(1964)
A formal description of a subset of Algol. pp1-12 in Steel(1966).

McCARTHY, J. et al.(1962)
The LISP 1.5 programming manual. MIT Press.

McCARTHY, J. and J. Painter(1967)
Correctness of a compiler for arithmetic expressions. in Proc. of Symposium in Applied Math. Vol 19, Amer. Math. Soc.

MANNA, Z.(1969)
Properties of programs and the first-order predicate calculus. J. ACM. 16 pp244-255.

MANNA, Z.(1970)
The correctness of nondeterministic programs. Art. Intell. J. 1 pp1-26.

MANNA, Z.(1971)
Mathematical theory of partial correctness. pp252-269 in Engeler(ed.)(1971).

MANNA, Z. and A. Pnueli(1970)
Formalization of properties of functional programs. J. ACM. 17 pp555-569.

MANNA, Z. and R.J. Waldinger(1971)
Towards automatic program synthesis. pp270-310 in Engeler(ed.)(1971).

MARKOV, A.A.(1962)
The theory of algorithms. Israel Program for Scientific Translations, Jerusalem, 1962.

MAURER, W.D.(1972)
A semantic extension of BNF. Int. J. Computer Math. 3, pp157-176.

MENDELSON, E.(1964)
Introduction to mathematical logic. Van Nostrand, New York.

MILGRAM, D.L. and A. Rosenfeld(1971)
A note on scattered context grammars.
Information Processing Letters, 1 pp47-50.

NAUR, P.(1964)
discussion on p158 of Steel(ed.)(1966).

NAUR, P. et al.(1963)
Revised report on the algorithmic language Algol 60.
C. ACM. 6,1 pp1-17.

NIVAT, M. and N. Nolin(1964)
Contribution to the definition of Algol semantics.
pp148-159 in Steel(ed.)(1966).

ROSENKRANTZ, D.J.(1969)
Programmed grammars and classes of formal languages.
J. ACM. 16,1 pp107-131.

SCHWARTZ, T.(1970)
Semantic definition methods and the evolution of programming languages. pp1-23 in Rustin(ed.)(1972).

SINTZOFF, M.(1967)
Existence of a van Wijngaarden syntax for every recursively enumerable set. Annales de la Societe Scientifique de Bruxelles,81,115-118.

STEARNS, R.E. and P.M. Lewis(1969)
Property grammars and table machines.
Information and Control 14, pp524-549.

STRACHEY, C.(1964)
Towards a formal semantics. pp198-220 in Steel(ed.)(1966).

WALK, K. et al.(1969)
Abstract syntax and interpretation of PL/I.
TR 25.098 IBM Lab. Vienna.

WEGNER, P.(1968)
Programming languages, information structures, and machine organisation. McGraw-Hill, New York.

WEGNER, P.(1970)
Programming language semantics. pp149-248 in Rustin(ed.)(1972).

WEGNER, P.(1971)
Operational semantics of programming languages.
pp128-141 in SIGPLAN notices 7,1.

WEGNER, P.(1972)
The Vienna Definition Language.
Computing Surveys 4,1 pp5-63.

van WIJNGAARDEN, A.(1962)
Generalised Algol. pp17-26 in Annual Review on Automatic Programming R. Goodman(ed.) Vol 3, Pergamon Press, 1963.

van WIJNGAARDEN, A.(1964)
Recursive definition of syntax and semantics.
pp13-24 in Steel(ed.)(1966).

van WIJNGAARDEN, A. et al.(1968)
Report on the algorithmic language Algol 68.
Nunerische Mathematik 14 pp79-218.

van WIJNGAARDEN, A. et al.(1974)
Revised report on the algorithmic language Algol 68.
TR 74-3 Computer Science Dept., University of Alberta.

WIRTH, N. and H. Weber(1966)
Euler: A generalisation of Algol, and its formal definition.
C. ACM. Vol 9, nos. 1 pp11-25, 2 pp 89-99.

Bibliography

sources which have been read, but not explicitly referenced in the text.

AHO, A.V.(1968)
Indexed grammars. J. ACM. 15,4 pp 647-671.

AHO, A.V. and J.D. Ullman(1972)
The theory of parsing, translation, and compiling. Vol I. Prentice-Hall, 1972.

BAKER, J.L.(1970)
Acceptors from van Wijngaarden grammars. Univ. of Washington, Computer Science Group, TR 70-02-10.

BASILI, V.R. and A.J. Turner(1973)
A hierarchical machine model for the semantics of programming languages. SIGPLAN notices 8,11 pp 152-164.

BEECH, D.(1970)
A structural view of PL/I. Computing Surveys 2,1, pp 33-64.

BEECH, D. and M. Marcotty
Unfurling the PL/I standard. SIGPLAN notices 8,10, pp 12-43.

BERRY, D.M.(1971)
Block structure: retention or deletion? 3rd SIGACT Symposium on the Theory of Computing, May 1971.

BOBROW, D.G.(1966)
Symbol manipulation languages and techniques. IFIP Working Conference 1966. North-Holland, Amsterdam,1968.

BOHM, C.(1964)
CUCH as a formal and description language. pp179-197 in Steel(ed.)(1966)

BOHM, C. and W. Gross(1966)
Introduction to the CUCH. pp35-65 in Automata Theory, Caianello(ed.), Academic Press,1966.

BROOKER, R.A.(1963)
The compiler compiler. pp 229-275 in Annual Review in Automatic Programming, Vol 3, R. Goodman(ed.), Pergamon Pr.,1963.

BURSTALL, R.M.(1967)
Semantics of assignment. pp3-20 in Machine Intelligence 2, Dale & Michie(eds.) Oliver and Boyd, 1967.

BURSTALL, R.M.(1970)
Formal description of program structure and semantics in first-order logic. pp79-98 in Machine Intelligence 5, Meltzer and Michie (eds.) Edinburgh Univ. Press.

de CHASTELLIER, G. and A. Colmerauer(1969)
W-grammars. Proc. ACM National Conf., 1969, pp 511-518.

CHEATHAM, T.E. and K. Sattley(1964)
Syntax-directed compiling. AFIPS Proc. Eastern JCC, 25, pp31-57.

COHEN, K. and J.H. Wegstein(1965)
AXLE, an axiomatic language for string transformations. C.ACM. 8, pp 657-661.

ELGOT, C.C. and A. Robinson(1964)
Random-access, stored-program machines. J.ACM. 11, pp365-399.

ENGELER, E.(1967)
Algorithmic properties of structures. Math. Systems Theory 1, pp 183-195.

ENGELER, E. (ed.)(1971)
Symposium on the semantics of programming languages. Springer-Verlag, 1971.

FELDMAN, J.A.(1966)
A formal semantics for computer languages. C.ACM. 9,1, pp3-9.

HERRIOT, R.G.(1973)
GLOSS: A high level machine. SIGPLAN notices, 8,11, pp 81-90.

IRONS, E.T.(1961)
Syntax-directed compiler for Algol 60. C.ACM. 4,1, pp 51-55.

JOHNSTON, J.B.(1971)
The contour model of block-structured processes. SIGPLAN notices, 6,2, pp 55-82.

LANDIN, P.J.(1963)
A lambda-calculus approach. pp97-141 in Advances in Programming and Non-numerical Computation, L. Fox(ed.) Pergamon Press, New York, 1966.

LANDIN, P.J.(1966)
The next 700 programming languages. C.ACM. 9, pp157-166.

LEDGARD, H.F.(1972)
Embedding Markov normal algorithms within the lambda-calculus.
Int. J. Computer Math. 3, pp131-140.

LEE, J.A.N.(1972)
Computer semantics. Van Nostrand, 1972.

LEE, J.A.N.(1973)
VDL - A definitional system for all levels.
pp41-48 in Proc. First Ann. Symp. Computer Architecture,
Lipovski and Szygenda (eds.) IEEE & ACM,1973.

LEWIS, P.M. and R.E. Stearns(1968)
Syntax-directed transduction. J.ACM. 15,3, pp 465-488.

LONDON, R.L.(1970)
Bibliography on proving the correctness of computer programs.
pp569-580 in Machine Intelligence 5, Meltzer and Michie(eds.),
Edinburgh Univ. Press, 1970.

LUCAS, P. and K. Walk(1969)
On the formal definition of PL/I. Ann. Rev. Aut. Progr. 6,105-181.

LUCKHAM, D.C. et al.(1970)
On formalized computer programs.
J. Computer & System Sciences, 4, pp220-249.

McCARTHY, J.(1965)
Problems in the theory of computation. IFIP 1965, Vol 1, 219-222.

MAURER, W.D.(1966)
A theory of computer instructions. J.ACM. 13,2, pp 226-235.

MAYER, O.(1972)
Some restrictive devices for context-free grammars.
Information and Control 20 pp 69-92.

NARASIMHAN, R.(1967)
Programming languages and computers: A Unified Metatheory.
pp189-244 in Advances in Computers 8, Academic Press, 1967.

PRATT, T.(1971)
Pair grammars, graph languages, and string-to-graph translations.
J. Computer & System Sciences, 5, pp 560-595.

RUSTIN, R.(ed.)(1972)
Formal semantics of programming languages.
Prentice-Hall,1972. Eight papers of Courant Computer Science
Symposium No. 2, Sept. 1970.

SCOTT, D.(1970)
Lattice theory, data types, and semantics. pp65-106 in Rustin(ed.)
(1972)

STEEL, T.B.(ed.)(1966)
Formal language description languages. Proc. IFIP 1964 Conf. North-Holland, 1966,1971.

URSCHLER, G.(1969)
Translation of PL/I into abstract syntax.
TR 25.097 IBM Lab. Vienna.

WILNER, W.T.(1970)
Formal semantic definition using synthesised and inherited attributes. pp 25-39 in Rustin(ed.)(1972).

WIRTH, N.(1963)
A generalisation of Algol. C.ACM. 6, pp547-554.

ZEMANEK, H.(1966)
Semiotics and programming languages. C.ACM. 9, pp 139-143.

Appendix A

Concrete syntax of ALEX: a context-free covering grammar

```
program   ::=  block
block   ::=  begin  declaration-list ; stmt-list  end
declaration-list   ::=  declaration /
                        declaration-list ; declaration
declaration   ::=  variable-declaration / procedure-
                   declaration / function-declaration
variable-declaration   ::=  type  id-list
id-list   ::=  id / id-list , id
procedure-declaration   ::= procedure  id  parameters ; stmt
function-declaration   ::=  type  function  id  parameters ;
                              stmt  returns  expression
type   ::=  integer / real / logical
parameters   ::= ( parameter-list) /  null
parameter-list   ::=  parameter / parameter-list, parameter
parameter   ::=  value-option  type  id
  value-option   ::=  value /  null

stmt-list   ::=  stmt  /  stmt-list ; stmt
stmt   ::=  unlabelled-stmt  /  id : stmt
unlabelled-stmt   ::= assignment-stmt / goto-stmt / if-stmt /
                      procedure-call / block /
                      compound-stmt  / null
assignment-stmt   ::=  id  :=  expression
goto-stmt   ::=  goto  id
if-stmt   ::= if  expression  then  stmt  else-part fi
else-part   ::=  else  stmt  /  null
procedure-call   ::=  id /  id ( argument-list )
argument-list   ::=  argument / argument-list , argument
argument   ::=  expression
compound-stmt   ::=  begin  stmt-list  end
null   ::=

expression   ::=  constant / var/ function-call /
                  binary  /  unary
constant   ::=  integer-value  /  real-value  / logical-value
logical-value   ::=  true / false
```

```
real-value    ::=      (not further specified)
integer-value    ::=          "
var    ::=  id
binary    ::=  ( expression  binary-op  expression )
unary    ::=  unary-op  expression
function-call    ::=  id  /  id ( argument-list )
binary-op    ::=       (not further specified)
unary-op    ::=                  "
id    ::=  letter  /  id  letter  /  id  number
letter    ::= a/b/c/d/e/f/g/h/i/j/k/l/m/n/o/p/q/r/s/t/u/v/w/x/y/z
number    ::= 0/1/2/3/4/5/6/7/8/9
```

Appendix B

Abstract representation of the concrete syntax in VDL

(see section 2.4.3.1)

AR1 is-c-progr = is-c-block

AR2 is-c-block = (⟨s_1: is-BEGIN⟩,
⟨s_2: is-c-decllist⟩,
⟨s_3: is-SEMIC⟩,
⟨s_4: is-c-stlist⟩,
⟨s_5: is-END⟩)

AR3 is-c-decllist = (⟨s-del: is-SEMIC⟩,
⟨s_1: is-c-decl⟩, ...)

AR4 is-c-decl = is-c-var-decl v is-c-proc-decl v is-c-func-decl

AR5 is-c-var-decl = (⟨s_1: is-c-type⟩,
⟨s_2: is-c-varlist⟩)

AR6 is-c-varlist = (⟨s-del: is-COMMA⟩,
⟨s_1: is-c-var⟩, ...)

AR7 is-c-var = is-c-id

AR8 is-c-type = is-INT v is-LOG v is-REAL

AR9 is-c-proc-decl = (⟨s_1: is-PROC⟩,
⟨s_2: is-c-id⟩,
⟨s_3: is-c-parlist v is-null⟩,
⟨s_4: is-SEMIC⟩,
⟨s_5: is-c-st⟩)

AR10 is-func-decl = (⟨s_1: is-c-func-type⟩,
⟨s_2: is-c-id⟩,
⟨s_3: is-c-parlist v is-null⟩,
⟨s_4: is-SEMIC⟩,
⟨s_5: is-c-st⟩,
⟨s_6: is-RETURNS⟩,
⟨s_7: is-c-expr⟩)

AR11 is-c-func-type = (⟨s_1: is-c-type⟩,⟨s_2: is-FUNC⟩)

AR12 is-c-parlist = (⟨s_1: is-LEFT-PAR⟩,
⟨s_2: (⟨s-del: is-COMMA⟩,
⟨s_1: is-c-param⟩, ...) ,
⟨s_3: is-RIGHT-PAR⟩)

```
AR13 is-c-param = (<s1: is-VALUE v is-null>,
                   <s2: is-c-type>,
                   <s3: is-c-id>)
AR14 is-c-stlist = (<s-del: is-SEMIC>,
                    <s1: is-c-st>, ...)
AR15 is-c-st = is-c-ul-st v is-c-lab-st
AR16 is-c-lab-st = (<s1: is-c-id>,
                    <s2: is-COLON>,
                    <s3: is-c-st>)
AR17 is-c-ul-st = is-c-assign-st v is-c-cond-st v is-c-proc-cal
                  v is-c-block v is-c-goto-st v
                  is-c-compound-st v is-null
AR18 is-c-assign-st = (<s1: is-c-id>,
                       <s2: is-BECOMES>,
                       <s3: is-c-expr>)
AR19 is-c-goto-st = (<s1: is-GOTO>,
                     <s2: is-c-id>)
AR20 is-c-cond-st = (<s1: is-IF>,
                     <s2: is-c-expr>,
                     <s3: is-THEN>,
                     <s4: is-c-st>,
                     <s5: is-c-else-part>,
                     <s6: is-FI>)
AR21 is-c-else-part = (<s1: is-ELSE>,
                       <s2: is-c-st>) v is-null
AR22 is-c-compound-st = (<s1: is-BEGIN>,
                         <s2: is-c-stlist>,
                         <s3: is-END>)
AR23 is-c-proc-call = (<s1: is-c-id>,
                       <s2: is-c-arglist v is-null>)
AR24 is-c-arglist = (<s1: is-LEFT-PAR>,
                     <s2: (<s-del: is-COMMA>,
                           <s1: is-c-arg>, ...) ,
                     <s3: is-RIGHT-PAR>)
AR25 is-c-arg = is-c-expr
```

AR26 is-c-expr = is-c-constant v is-c-var v is-c-func-call v
is-c-bin v is-c-unary

AR27 is-c-constant = is-c-log v is-c-int v is-c-real

AR28 is-c-log = is-TRUE v is-FALSE

AR29 is-c-int (not further specified)

AR30 is-c-real "

AR31 is-c-var = is-c-id

AR32 is-c-bin = (⟨s_1: is-LEFT-PAR⟩,
⟨s_2: is-c-expr⟩,
⟨s_3: is-c-binary-op⟩,
⟨s_4: is-c-expr⟩,
⟨s_5: is-RIGHT-PAR⟩)

AR33 is-c-unary = (⟨s_1: is-c-unary-op⟩,
⟨s_2: is-c-expr⟩)

AR34 is-c-func-call = (⟨s_1: is-c-id⟩,
⟨s_2: is-c-arglist⟩)

AR35 is-c-binary-op (not further specified)

AR36 is-c-unary-op "

AR37 is-c-id = (all the 'id' of Appendix A)

AR38 is-BEGIN(begin)
is-END(end)
is-PROC(procedure)
is-FUNC(function)
is-INT(integer)
is-REAL(real)
is-LOG(logical)
is-RETURNS(returns)
is-IF(if)
is-THEN(then)
is-ELSE(else)
is-FI(fi)
is-TRUE(true)
is-FALSE(false)
is-VALUE(value)
is-GOTO(goto)
is-COMMA(,)
is-SEMIC(;)
is-COLON(:)
is-BECOMES(:=)
is-LEFT-PAR(()
is-RIGHT-PAR())

Appendix C

Abstract syntax for the VDL definition of ALEX

```
A1  is-progr = is-block
A2  is-block = (<s-decl-part: is-decl-part>,
                <s-st-list: is-st-list>)
A3  is-decl-part = ( {<id: is-attr> : is-id(id)} )
A4  is-attr = is-var-attr v is-proc-attr v is-func-attr v
              is-label-attr
A5  is-var-attr = is-type
A6  is-proc-attr = (<s-param-list: is-param-list>,<s-st: is-st>)
A7  is-func-attr = (<s-param-list: is-param-list>,<s-st: is-st>,
                    <s-func-type: is-type>,<s-result: is-expr>)
A8  is-param = (<s-value-opt: is-VALUE v is-null>,
                <s-type: is-type>,
                <s-par: is-id>)
A9  is-type = is-INT v is-REAL v is-LOG
A10 is-label-attr = (<s-st-loc: is-int-list>)

A11 is-st = is-assign-st v is-cond-st v is-goto-st v is-null v
            is-proc-call v is-st-list v is-block
A12 is-assign-st = (<s-left-part: is-var>,
                    <s-right-part: is-expr>)
A13 is-goto-st = (<s-label: is-id>)
A14 is-proc-call = (<s-id: is-id>,<s-arg-list: is-arg-list>)
A15 is-cond-st = (<s-choice: is-expr>,
                  <s-then-st: is-st>,<s-else-st: is-st>)
A16 is-arg = is-expr

A17 is-expr = is-const v is-var v is-binary v is-unary
              v is-func-call
A18 is-const = is-log-c v is-int-c v is-real-c
A19 is-log-c = is-TRUE v is-FALSE
A20 is-var = is-id
A21 is-binary =(<s-rd1:is-expr>,<s-op:is-binary-op>,
                <s-rd2:is-expr>)
A22 is-unary = (<s-rd:is-expr>,<s-op:is-unary-op>)
A23 is-func-call = is-proc-call
A24 is-int-c, is-real-c, is-unary-op, is-binary-op
        are not further specified
A25 is-INT(INT), is-REAL(REAL) etc.
```

Appendix D

Translation from concrete to abstract syntax

T1 translate(t) =
is-c-progr(t) → trans-block(t,null)
T → error

T2 trans-block(cb,decs) =
PASS: u_0(⟨s-decl-part:decpart⟩,⟨s-st-list:
trans-stlist(s_4(cb),syms,decpart,
u_0(⟨s-st-loc:null⟩))⟩)
decpart: trans-decllist(s_2(cb),syms,newdecs)
syms: u(decs;{⟨id:attr⟩ for id(newdecs)=attr})
collect-declarations(cb,newdecs)
newdecs: null

↑ the VDL order of execution

Pass 1: collecting all declarations

T3 collect-declarations(cb,newdecs) =
collect-labels-st(s_i(s_4(cb)),newdecs) for all i:s_i(s_4(cb)) ≠null
collect-decl(s_i(s_2(cb)),newdecs) for all i :s_i(s_2(cb))≠null

T4 collect-decl(decl,nd) =
is-c-proc-decl(decl) → add-dec(nd,s_2(decl),PROC)
is-c-func-decl(decl) → add-dec(nd,s_2(decl),FUNC)
is-c-var-decl(decl) → add-dec(nd,id,s_1(decl))
for all i : id=s_i(s_2(decl))≠null

T5 collect-labels-st(st,nd) =
is-null(st) → null
is-c-labst(st) → collect-labels-st(s_3(st),nd)
add-dec(nd,s_1(st),LABEL)
is-c-assign-st(st) v is-c-proc-call(st) v is-c-goto-st(st)
→ null
is-c-compound-st(st) → collect-labels-st(s_i(s_2(st)),nd)
for all i : s_i(s_2(st))≠null
is-c-block(st) → null
is-c-cond-st(st) → collect-labels-st(s_4(st),nd)
collect-labels-st(s_2(s_5(st)),nd)

T6 add-dec(newdecs,id,attr) =
newdecs : u(newdecs; ⟨id:attr⟩)
id(newdecs) ≠ null → error

Pass 2: translation, checking identifiers, filling label locs

```
T7 trans-decllist(decllist,syms,newdecs) =
    {trans-decl(s_i(s_2(decllist)),syms,newdecs)
       for all i : s_i(s_2(decllist))≠null}

T8 trans-decl(decl,syms,newdecs) =
     is-c-proc-decl(decl) →
        newdecs:u(newdecs; <s_2(decl): u_0(<s-param-list:pl>,
                                           <s-st:pb>)>)
         pb : trans-proc-body(s_5(decl),u(syms; pl))
          pl : trans-par-list(s_3(decl))
     is-c-func-decl(decl) →
        newdecs:u(newdecs;  s_2(decl):
                    u_0(<s-param-list:pl>,
                       <s-st:pb>,
                       <s-func-type:s_1(s_1(decl))>,
                       <s-result:re>)>)
         type≠s_1(s_1(decl)) → error
          re : trans-expr(s_7(decl),u(syms; pl),type)
           pb : trans-proc-body(s_5(decl),u(syms; pl))
            pl : trans-par-list(s_3(decl))
     is-c-var-decl(decl) → null % already in 'newdecs'
                                  by collect-decl

T9 trans-par-list(cpl) =
     is-null(cpl) → null
      not(∃i,j) (i≠j & s_i(s_2(cpl))=s_j(s_2(cpl))≠null) →
         u_0( {<elem(i): u_0(<s-value-opt: s_1(p)>,
                             <s-type     : s_2(p)>,
                             <s-par      : s_3(p)>)>
               : p=s_i(s_2(cpl)) for all i in 1:slength(s_2(cpl))})
      T → error

T10 trans-proc-body(bod,syms) =
      PASS: blk
       blk: u_0(<s-decl-part:decp>,<s-st-list:u_0(<elem(1):st>)>)
        decp=null → PASS: st    % no labels to worry about
         st: trans-st(bod,syms,decp,stloc)
          stloc: u_0(<s_1 : 1>)
           decp: null
```

```
T11 trans-stlist(t,syms,decp,stloc) =
      u0( {<elem(i): trans-st(si(t),syms,decp,u(stloc;<sd+1:i>))>
           : for all i in 1:slength(t)} )
      where: d=slength(t)

T12 trans-st(st,syms,decp,stloc) =
      is-c-lab-st(st) -> PASS: trans-st(s3(st),syms,decp,stloc)
                         decp: u(decp; <s1(st):
                                        u0(<s-st-loc:stloc>)>>
      is-null(st) -> null
      is-c-assign-st(st) -> trans-assign(s1(st),s3(st),syms)
      is-c-cond-st(st) -> trans-cond(s2(st),s4(st),s2(s5(st)),
                                     syms,decp,stloc)
      is-c-goto-st(st) -> PASS: u0(<s-label: lab>)
                          not is-label-attr(lab(syms)) -> error
      is-c-block(st) -> trans-block(st,syms)
      is-c-compound-st(st) -> trans-stlist(s2(st),syms,decp,
                                           stloc)
      is-c-proc-call(st) -> trans-proc-call(st,syms)

T13 trans-cond(v,then,else,syms,decp,stloc) =
      PASS: u0(<s-choice: c>,
               <s-then-st:trans-st(then,syms,decp,u0(l,<sd+1:1>)>
               <s-else-st:trans-st(else,syms,decp,u0(l, sd+1:2>)

      where: d=slength(stloc), l=stloc

T14 trans-assign(var,cexp,syms) =
      PASS: u0(<s-left-part:var>,<s-right-part:aexp>)
       not convertible(type,var(syms)) -> error
        aexp: trans-expr(cexp,syms,type)

T15 trans-proc-call(call,syms) =
     PASS: u0(<s-id: s1(call)>,<s-arg-list: al>)
      al : trans-arglist(s2(call),s-param-list(s1(call)syms),sym
       not is-proc-attr(s1(call)(syms)) -> error
```

```
T16 trans-arglist(apl,fpl,syms) =
      PASS: u0({ <elem(i): trans-arg(si(apl),si(fpl),syms)>
                 : for all i in 1:slength(apl) })
       slength(apl) ≠ slength(fpl) → error
        is-null(apl) & is-null(fpl) → null

T17 trans-arg(ap,fp,syms) =
      PASS : aexp
       not convertible(type,s-type(fp)) → error
        aexp : trans-expr(ap,syms,type)

T18 trans-expr(ex,syms,type) =      % return both type & abstract
      is-c-constant(ex) → ex                                 syntax
      is-c-var(ex) → PASS : ex
                      not is-var-attr(type) → error
                       type : ex(syms)
      is-func-call(ex) → trans-func-call(ex,syms,type)
      is-c-bin(ex) → PASS:u0(<s-rd1:a>,<s-rd2:b>,<s-op:s3(ex)>)
                      type: result-type2(ta,s3(ex),tb)
                       a: trans-expr(s2(ex),syms,ta)
                        b:trans-expr(s4(ex),syms,tb)
      is-c-unary(ex) → PASS:u0(<s-rd:a>,<s-op:s1(ex)>)
                        type: result-type1(s1(ex),ta)
                         a : trans-expr(s2(ex),syms,ta)

T19 trans-func-call(call,syms,type) =
      PASS: u0(<s-id:s1(call)>,<s-arg-list:al>)
       type : s-func-type(fa)
        al : trans-arglist(s2(call),s-param-list(fa),syms)
         not is-func-attr(fa) → error
     where: fa= s1(call)(syms)

T20 the following functions are not further specified:
      result-type1(op,arg) : type of op(arg)
      result-type2(arg1,op,arg2) : type of op(arg1,arg2)
T21 convertible(t1,t2) =
      t1=t2 → true
      is-INT(t1) & is-REAL(t2) → true
      T → false
```

T22 slength(x) =

$(\forall i)$ is-null($s_i(x)$) $\rightarrow$ 0

T $\rightarrow$ (ιi) (<u>not</u> is-null($s_i(x)$)

& $(\forall j)$ (j>i <u>implies</u> is-null($s_j(x)$)))

ι The iota operator is used in expressions of the form (ιx) p(x).

This expression denotes the value of x for which p(x) is true, but has no value if no value or more than one value in the range of x has the property p.

Appendix E

Definition of ALEX syntax by a Production System

programs

P PROGRAM b r:=null

b BLOCK begin $d_1;\ldots;d_m;s_1;\ldots;s_n$ end

if r_d:=B($d_1;\ldots;d_m;s_1;\ldots;s_n$)

r':=OVERRIDE(r_d,r)

DIFF.IDLIST(DOMAIN(r_d))

STATEMENT(s_1,r'),...,STATEMENT(s_n,r')

declarations

d DECLARATION vd / pd / fd

vd VARIABLE DECL. t $i_1,i_2,\ldots,i_m$ if m geq 1

t TYPE integer / real / logical

pd PROCEDURE DECL. procedure i pl ; s

if r_p:= PARAMS(pl) + B(s)

r':= OVERRIDE(r_p,r)

DIFF.IDLIST(DOMAIN(r_p))

STATEMENT(s,r')

fd FUNCTION DECL. t function i pl ; s returns e

if r_p:= PARAMS(pl) + B(s)

r':= OVERRIDE(r_p,r)

DIFF.IDLIST(DOMAIN(r_p))

STATEMENT(s,r')

B(t) = CT(e,r')

pl PARAMETER LIST. ($v_1t_1i_1$, ... ,$v_mt_mi_m$) / null if m geq 1

v VALUE OPTION value / null

statements

s STATEMENT us / i:s

us UNLABEL. STMT. ASGT-STMT / GOTO-STMT / IF-STMT /
PROC-CALL / BLOCK / null

GOTO-STMT goto i if r(i)=LABEL

ASGT-STMT	i:=e	if r(i)=INT v LOG v REAL CONVERTIBLE(CT(e,r),r(i))
IF-STMT	<u>if</u> e <u>then</u> s <u>fi</u> /	
	<u>if</u> e <u>then</u> s_1 <u>else</u> s_2 <u>fi</u>	if CONVERTIBLE(CT(e,r),LOG)
PROC-CALL	PROC / PARAM-PROC	
PROC	i	if r(i)=PROC()
PARAM-PROC	i(e_1, ... ,e_m)	if m geq 1 r(i)=PROC(pt_1,...,pt_n) , m=n CONVERTIBLE(CT(e_k),pt_k) for all k=1,...,m

<u>expressions and primaries</u>

e	EXPRESSION	c / i / f / bin / un
c	CONSTANT	ic / rc / lc
lc	LOGICAL CONST	<u>true</u> / <u>false</u>
rc	REAL CONSTANT	(not further specified)
ic	INTEGER CONSTANT	"
i	IDENTIFIER	lt / i lt / i dg
f	FUNCTION CALL	FUN / PARAM-FUN
	FUN	i if r(i)=FUN(-rt)
	PARAM-FUN	i(e_1, ... ,e_m) if m geq 1 r(i)=FUN(pt_1,...,pt_n-rt) , m=n CONVERTIBLE(CT(e_k),pt_k) for all k=1,...,m
un	UNARY EXPR	UNARY-OP e
bin	BINARY EXPR	(e_1 BINARY-OP e_2)
	BINARY-OP	(not further specified)
	UNARY-OP	"
dg	DIGIT	0 / 1 / 2 / 3 / 4 / 5 / 6 / 7 / 8 / 9
lt	LETTER	a / b / c ... /z

<u>computation of environments</u>

B	COMPUTED DECLS.	B(d_1;...;d_m;s_1;...;s_n) = B(d_1)+ ...+B(d_m)+ B(s_1)+ ...+B(s_n)
		B(i:s) = (i:LABEL) +B(s)
		B(null) = null

B(i:=e) = null

B(goto i) = null

B(PROC-CALL) = null

B(if e then s fi) = B(s)

B(if e then s_1 else s_2 fi) = B(s_1)+B(s_2)

B(begin d_1;...;d_m;s_1;...;s_n end) =
null if m geq 1
B(s_1)+...+B(s_n) if m=0

B(procedure i pl ; s) = (i:PROC(B(pl)))

B(t function i pl ; s returns e) =
(i : FUN(B(pl) - B(t)))

B(($v_1 t_1 i_1$, ... ,$v_m t_m i_m$)) =
B(v_1)B(t_1), ... ,B(v_m)B(t_m)

B(value) = VAL

B(integer) = INT

B(real) = REAL

B(logical) = LOG

B(t i_1,i_2, ... ,i_m) = (i_1:B(t))+...+(i_m:B(t))

PARAMS — PARAMS(($v_1 t_1 i_1$,...,$v_m t_m i_m$)) =
(i_1:B(t_1), ... ,i_m:B(t_m))

computed types of expressions

CT COMPUTED TYPE — CT(ic,r) = INT

CT(rc,r) = REAL

CT(lc,r) = LOG

CT(i ,r) = r(i) if r(i)= INT v REAL v LOG
rt if r(i)= FUN(pt_1,...,pt_n-rt)

CT(UNARY-OP e , r) (not further specified)

CT(e_1 BINARY-OP e_2,r) "

CONVERTIBLE — CONVERTIBLE(t_1,t_2) = t_1=t_2 v t_1=INT & t_2=REAL

miscellaneous predicates and functions

DOMAIN — DOMAIN(null) = null

DOMAIN((i_1:dal_1),...,(i_n:dal_n)) = i_1,...,i_n

\+ CONCATENATION — r+null = r

r+r' = r" if r=(x), r'=(y), and r"=(x,y)

DIFF.IDLIST — DIFF.IDLIST(i_1,...,i_n) = $\forall p \forall q \; i_p \neq i_q$ if p≠q

OVERRIDE — OVERRIDE(((i_1:r_1),...,(i_m:r_m)),((j_1:s_1),...)) =

((i_1:r_1),...,(i_m:r_m), list (j_q:s_q))

$\forall p \; i_p \neq j_q$

Appendix F

Definition of ALEX syntax by a van Wijngaarden grammar

```
program : new block.

NEST block : begin symbol, NEST new DEFS LABSETY declaration of
             DEFS, semicolon,
             NEST new DEFS LABSETY series defining LABSETY,
             end symbol.

NEST declaration of DEF DEFS : NEST declaration of DEF, semicolo
                                , NEST declaration of DEFS.
NEST declaration of ID has TYPE : TYPE symbol, ID token.
TYPE symbol, IDCLIST, semicolon, TYPE symbol, ID token :
  TYPE symbol, IDCLIST, comma, ID token.
NEST declaration of ID has proc(PARAMS) :
  procedure symbol, ID token, PARAMS paramlist, semicolon,
    NEST new PARAMS procedure body.
NEST declaration of ID has func(PARAMS-TYPE) :
  TYPE symbol, function symbol, ID token,
    PARAMS paramlist, semicolon,
      NEST new PARAMS procedure body,
        returns symbol, TYPE NEST new PARAMS expression.
NEST procedure body : NEST statement;
                      NEST new LABS statement defining LABS.
EMPTY paramlist : EMPTY.
PRMS paramlist: left paren, PRMS parms, right paren.
PRM PRMS parms: PRM parm, comma, PRMS parms.
ID has TYPE parm : TYPE symbol, ID token.
ID has value TYPE parm : value symbol, TYPE symbol, ID token.

NEST series defining LABSETY :
   NEST statement defining LABSETY ;
   where LABSETY is LABSETY2 LABSETY3,
      NEST statement defining LABSETY2, semicolon,
      NEST series defining LABSETY3.
```

statements

```
NEST statement defining LABSETY :
   where LABSETY is EMPTY, NEST statement;
   where LABSETY is ID has label LABSETY2,
      ID token, colon, NEST statement defining LABSETY2;
   NEST compound statement defining LABSETY;
   NEST conditional defining LABSETY.
NEST compound statement defining LABSETY :
   begin symbol, NEST series defining LABSETY, end symbol.
NEST conditional defining LABSETY :
   NEST if then part defining LABSETY, fi symbol;
   where LABSETY is LABSETY2 LABSETY3,
      NEST if then part defining LABSETY2,
      else symbol, NEST statement defining LABSETY3,
      fi symbol.
NEST if then part defining LABSETY :
   if symbol, logical NEST expression, then symbol,
      NEST statement defining LABSETY.
NEST statement : NEST assignment; NEST goto; NEST block;
                 NEST procedure call; EMPTY.

NEST goto : goto symbol, label NEST identifier.
NEST assignment : TYPE1 NEST identifier, becomes symbol,
                      TYPE2 NEST expression,
                   where TYPE2 is convertible to TYPE1.
NEST procedure call : proc(PARAMS) NEST identifier,
                       actual NEST parameters pack matching PARAMS

actual NEST parameters pack matching EMPTY : EMPTY.
actual NEST  parameters pack matching PRMS :
   left paren, actual NEST parameters matching PRMS,
   right paren.
actual NEST parameters matching PRM PRMS :
   actual NEST parameter for PRM, comma, actual NEST
   parameters matching PRMS.
actual NEST parameter for ID has VALUE option TYPE1 :
   TYPE2 NEST expression,
   where TYPE2 is convertible to TYPE1.
VALUE option : value; EMPTY.
```

expressions

```
TYPE NEST expression : TYPE constant; TYPE NEST identifier;
                       TYPE NEST binary; TYPE NEST unary;
                       TYPE NEST function call.
TYPE NEST function call :
   func(PARAMS-TYPE) NEST identifier,
   actual NEST parameters pack matching PARAMS.
TYPE NEST unary : UNARY-OP, TYPE2 NEST expression,
                  where TYPE is resulttype of UNARY-OP on TYPE2.
TYPE NEST binary : left paren, TYPE2 NEST expression,
                   BINARY-OP,  TYPE3 NEST expression,
                   right paren,
                   where TYPE is resulttype of BINARY-OP
                                 on TYPE2 and TYPE3.
logical constant : true symbol; false symbol.
integer constant and real constant are not further specified.
```

conversions and result types

```
where TYPE is convertible to TYPE : true.
where integer is convertible to real : true.
'resulttype of .. on .. and ..' is not further specified,
  but an example is
       resulttype of plus symbol on integer and real : real.
```

identifiers

```
MODE NEST identifier : ID token,
                       where ID has MODE from NEST.
PROP : ID has ATTRIBUTE.
where PROP from NEST new PROPSETY :
   where PROP is not in PROPSETY, where PROP from NEST;
   where PROP is alone in PROPSETY.
where PROP1 is alone in PROPS PROP2:
   where PROP1 is not in PROPS, where PROP1 is alone in PROP2;
   where PROP1 is alone in PROPS, where PROP1 is not in PROP2.
where PROP1 is not in PROPS PROP2 :
   where PROP1 is not in PROPS, where PROP1 is not in PROP2.
where ID has ATTRIBUTE is alone in ID has ATTRIBUTE : true.
where ID1 has ATTRIBUTE1 is not in ID2 has ATTRIBUTE2 :
   where ID1 differs from ID2.
```

```
predicates
   true : EMPTY.
   where NOTETY is NOTETY : true.
   where NOTETY1 ALPHA1 differs from NOTETY2 ALPHA2 :
      where NOTETY1 differs from NOTETY2;
      where ALPHA1 precedes ALPHA2 in ALPHABET;
      where ALPHA2 precedes ALPHA1 in ALPHABET.
   where ALPHA1 precedes ALPHA2 in
         NOTETY1 ALPHA1 NOTETY2 ALPHA2 NOTETY3 : true.
   where NOTION differs from EMPTY : true.
   where EMPTY differs from NOTION : true.

metaproductions
   NEST :: LAYER ; NEST LAYER.
   LAYER :: new DEFSETY LABSETY.
   DEFSETY :: DEFS ; EMPTY.
   DEFS :: DEF ; DEFS DEF.
   DEF :: ID has MODE.
   LABSETY :: LABS ; EMPTY.
   LABS :: LAB ; LABS LAB.
   LAB :: ID has label.
   ATTRIBUTE :: MODE ; label.
   MODE :: TYPE ; proc(PARAMS) ; func(PARAMS-TYPE).
   TYPE :: integer ; real ; logical.
   EMPTY ::.
   PARAMS :: PRMSETY.
   PRMSETY :: PRMS ; EMPTY.
   PRMS :: PRM ; PRMS PRM.
   PRM :: ID has TYPE ; ID has value TYPE.
   PROPSETY :: PROPS ; EMPTY.
   PROPS :: PROP ; PROPS PROP.
   PROP :: ID has ATTRIBUTE.

   ID :: LETTER ; ID LETTER ; ID DIGIT;
   IDCLIST :: ID ; ID, comma, IDCLIST.
   NOTETY :: NOTION ; EMPTY.
   NOTION :: CHAR ; NOTION CHAR.
   CHAR :: LETTER ; ( ; ) ; - .
   ALPHABET :: abcdefghijklmnopqrstuvwxyz()- .
   DIGIT :: 0 ;1 ; 2 ; 3 ; 4 ; 5;6 ; 7 ; 8 ; 9 .
   LETTER :: a;b;c;d;e;f;g;h;i;j;k;l;m;n;o;p;q;r;s;t;u;v;w;x;y;z.
   ALPHA :: CHAR
```

LETTER token : LETTER symbol.
ID LETTER token : ID token, LETTER symbol.
ID DIGIT token : ID token, DIGIT symbol.

representations

begin symbol	begin
end symbol	end
procedure symbol	procedure

function, if, fi, then, else, returns, goto, integer, real, logical, value, true, and false symbols similarly.

right paren	)
left paren	(
semicolon	;
colon	:
comma	,
becomes symbol	:=
plus symbol	+
etc.	
a symbol	a
b symbol	b
.	
z symbol	z
0 symbol	0
.	
9 symbol	9

Appendix G

Definition of ALEX syntax by a Linked-forest Manipulation System

Context-free component	Abstract tree component
1 program → block	program: block
1.1 USED	S(ERROR)F(STOP)
2 block → begin declaration-list; stmt-list end	block: declaration-list, stmt-list
2.1 DEC $type_1$:id_1 & DEC $type_2$:id_1	S(ERROR)F(NEXT)
2.2 $declaration_1$: DEC type:id_2 & variable: USED id_2 → $declaration_1$: DEC type:id_2 & variable=$declaration_1$-designator	
2.3 $parameter_1$:spec,type,id_2 & variable: USED id_2 → $parameter_1$:spec,type,id_2 & variable=$parameter_1$-designator	S(REPEAT)F(NEXT)
2.4 $declaration_1$ & $stmt_3$:(DEC label: id),$stmt_4$ → $declaration_1$,$declaration_2$=label:$stmt_3$-designator,id & $stmt_4$	S(REPEAT)F(NEXT)
2.5 goto-stmt: USEDid_1 & declaration=$label_2$: stmt-designator,id_1 → goto-stmt: $label_2$-designator & declaration	S(REPEAT)F(NEXT)
2.6 goto-stmt: USED id	S(ERROR)F(NEXT)
2.7 DEC $procedure_1$: id_2,$type\text{-}list_3$ & (procid: USED id_2),$argument\text{-}list_4$ → DEC $procedure_1$: id_2,$type\text{-}list_3$ & (procid: $procedure_1$-designator),$argument\text{-}list_4$, CHECK convertible (TYPES $argument\text{-}list_4$-copy, $type\text{-}list_3$)	S(REPEAT)F(NEXT)
2.8 DEC $function_1$: id_2,$type\text{-}list_3$ & (funcid: USED id_2),$argument\text{-}list_4$ → DEC $function_1$: id_2,$type\text{-}list_3$ & (funcid: $function_1$-designator),$argument\text{-}list_4$, CHECK convertible(TYPES $argument\text{-}list_4$-copy, $type\text{-}list_3$)	S(REPEAT)F(NEXT)
2.9 CHECK true → null	
2.10 CHECK false	S(ERROR)
2.11 declaration: DEC type: id → declaration=cell: value=undefined , type	S(REPEAT)F(NEXT)

2.12 DEC procedure → null S(REPEAT)F(NEXT)

2.13 DEC function → null S(REPEAT)F(NEXT)

2.14 id → null S(REPEAT)F(STOP)

3 declaration-list →
  declaration /
  declaration-list ;
  declaration
  declaration-list

4 declaration →
  type id-list
  declaration : type,
  id-list

4.1 declaration$_1$: type,(id-list: id$_2$,..) →
  declaration$_1$: type,(id-list: ,..) ,
  declaration$_3$: DEC type: id$_2$ S(REPEAT)F(STOP)

5 id-list → id /
  id-list,id
  id-list

6 declaration → procedure id
  parameters ;
  stmt
  declaration=procedure:
  parameters , stmt ,
  DEC procedure: id,
  TYPES parameters-copy

6.1 TYPES null → null

6.2 TYPES parameter-list → (TYPE parameter)-list

6.3 TYPE parameter: spec,type,id → type

6.4 Perform 2.1 to 2.14

7 declaration →
  type function id
  parameters;
  stmt returns expression
  declaration=function:
  parameters,stmt,type,
  expression ,
  DEC function: id,
  TYPES parameters-copy

7.1 Perform 6.1 to 6.4

8 type → integer/real/logical type = int / real / log

9 parameters → null /
  (parameter-list)
  parameters: null /
  parameter-list

10 parameter →
  value-option type id
  parameter: spec , type , id

11 value-option → value / null spec: value / name

12 parameter-list →
  parameter /
  parameter , parameter-list
  parameter-list

13 stmt-list → stmt / stmt-list;stmt	stmt-list
14 stmt → ulstmt / id: stmt	stmt = stmt / stmt: (DEC label: id),stmt
15 ulstmt → begin stmt-list end	stmt = stmt-list
16 ulstmt → if expression then stmt fi / if expression then $stmt_1$ else $stmt_2$ fi	stmt = if-stmt: expression,stmt,null / expression,$stmt_1$,$stmt_2$
17 ulstmt → id := expression	stmt = assignment-stmt: (variable: USED id),expression
18 ulstmt → goto id	stmt = goto-stmt: USED id
19 ulstmt → null	stmt = null
20 ulstmt → id / id(argument-list)	stmt=procedure-call: (procid: USED id),null / (procid: USED id), argument-list
22 argument → expression	argument = expression
21 argument-list → argument / argument-list,argument	argument-list
23 expression → true / false / real-value / integer-value	expression = constant = value = true/false/real-value/ integer-value
24 expression → id	expression = variable: USED id
25 expression → unary-op expression	expression = unary: unary-op,expression
26 expression →(expression$_1$ binary-op expression$_2$)	expression = binary: expression$_1$,binary-op, expression$_2$
27 expression → id / id(argument-list)	expression = function-call: (funcid: USED id),null / (funcid: USED id), argument-list

28 real-value, integer-value, unary-op, and binary-op are not further specified.

29 (extra rules for calculating TYPES of 2.7 & 2.8)

29.1 TYPES argument-list → (TYPE argument=expression)-list

29.2 TYPE true → log

29.3 TYPE false → log

29.4 TYPE real-value → real

29.5 TYPE integer-value → int

29.6 TYPE variable=declaration$_1$-designator
& declaration$_1$: DEC type
→type

29.7 TYPE unary: unary-op,expression →
result-type(unary-op, TYPE expression)

29.8 TYPE binary: expression$_1$, binary-op , expression$_2$ →
result-type2(binary-op, TYPE expression$_1$,
TYPE expression$_2$)

29.9 TYPE function-call: (funcid=function$_1$-designator),..
& function$_1$: parameter-list,stmt,type,expression
→ type

Appendix H

Machine states for the VDL definition

```
S1  is-state = (<s-env : is-env>,        % environment directory
                <s-c : is-c>,            % instruction control
                <s-ci : is-ci>,          % statement control
                <s-d : is-d>,            % dump: the stack
                <s-block-activation :
                   is-block-name>,       % unique block name
                <s-at : is-at>,          % attribute directory
                <s-dn : is-dn>,          % denotation directory
                <s-n : is-n>)            % unique-name index

S2  is-env = ({ <id:is-n> : is-id(id)} )

S3  is-c describes the set of all program control trees
                of instructions.

S4  is-ci = (<s-text : is-st v is-null>,
             <s-index : is-index>,       % index into text
             <s-ci : is-ci>,
             <s-c : is-c>)
             v is-null

S5  is-index = is-int v is-TRUE v is-FALSE

S6  is-d = (<s-env:is-env>,<s-c:is-c>,<s-ci:is-ci>,
            <s-block-activation:is-block-name>,
            <s-d : is-d>)
           v is-null

S7  is-block-name = is-n v is-null

S8  is-at = ({ <n:is-dntype> : is-n(n)} )

S9  is-dntype = is-INT v is-REAL v is-LOG v
                is-PROC v is-FUNC v is-THUNK v is-LABEL

S10 is-dn = ({ <n: is-value v
                   is-label-dn v
                   is-thunk-dn v
                   (<s-env : is-env>,
                    <s-attr : (is-proc-attr v is-func-attr)>)>
               : is-n(n)} )

S11 is-label-dn = (<s-block-activation:is-block-name>,
                   <s-st-loc : is-index-list>)

S12 is-thunk-dn =  (<s-env : is-env>,<s-attr : is-expr>)

S13 is-n = is-int
```

The initial state for any given program t for is-progr(t) is

$$I^0 = \mu_0(\langle s\text{-}c : \underline{int\text{-}proor}(t)\rangle, \langle s\text{-}n : 1\rangle).$$

States I whose control part s-c(I) is null are end-states.

Components of the current state I , s-env(I), s-c(I), etc., may be represented by the abbreviations ENV, C, CI, D, BA, AT, DN, and N.

Appendix I

Interpretation instructions for the VDL definition

I1 int-progr(t) =

int-block(t)

for: is-progr(t) - see Appendix C

I2 int-block(t) =

s-d : stack

s-c : exit ;
int-st-list(s-st-list(t)) ;
int-decl-part(s-decl-part(t)) ;
update-env (s-decl-part(t)) ;

↑ the VDL order of execution

s-block-activation : unique-name

I3 unique-name =

PASS : $n_{s\text{-}n(I)}$

s-n : s-n(I) + 1

I4 update-env(t) =

null ;
{update-id(id,n) ;n:unique-name , for all id(t)≠null}

I5 update-id(id,n) =

s-env : μ(ENV ; <id:n>)

I6 int-decl-part(t) =

null ;
{int-decl(id(ENV),id(t)) for all id(t)≠null}

I7 int-decl(n,attr) =

is-var-attr(attr) → s-at:μ(AT;<n:attr>)

is-proc-attr(attr) → s-at:μ(AT;<n:PROC>)
s-dn:μ(DN;<n:
μ_0(<s-attr:attr>,
<s-env:ENV)>)>

is-func-attr(attr) → s-at:μ(AT;<n:FUNC>)
s-dn:μ(DN;<n:
μ_0(<s-attr:attr>,
<s-env:ENV>))>)

is-label-attr(attr) → s-at:μ(AT;<n:LABEL>)
s-dn:μ(DN;<n:μ_0(<s-st-loc:
attr>,<s-block-activation:
BA)>)>

```
I8  int-st-list(t) =
        stack-ci(1,t)

I9  stack-ci(index,t) =
        s-ci: μ0(<s-text:t>,<s-index:index>,
                 <s-ci:CI> ,<s-c:C>)
        s-c : continue

I10 continue =
        int-next-st ;
         int-st(take-st(s-index(CI),s-text(CI)))

I11 take-st(index,t) =
        is-st-list(t) & is-int(index) & index in (1:length(t))
            -> elem(index) t
        is-if-st(t) & is-TRUE(index) -> s-then-st(t)
        is-if-st(t) & is-FALSE(index)-> s-else-st(t)
        T -> error

I12 int-next-st =
        is-int(s-index(CI)) & s-index(CI) < length(s-text(CI))
            -> continue ;
               upd-index ;
        T -> s-ci : s-ci(CI)
             s-c  : s-c(CI)

I13 upd-index =
        s-ci: μ(CI; <s-index: s-index(CI)+1>)

I14 int-st(t) =
        is-assign-st(t) -> int-assign-st(t)
        is-cond-st(t)   -> int-if-st(t)
        is-goto-st(t)   -> int-goto-st(t)
        is-proc-call(t) & at_t=PROC -> int-proc-call(t)
        is-st-list(t)   -> int-st-list(t)
        is-block(t)     -> int-block(t)
        is-null(t)      -> null
     where: at_t = (s-id(t))(ENV))(AI)

I15 int-if-st(t) =
        stack-ci(convert(choice,LOG),t)
         choice:int-expr(s-choice(t))

I16 int-assign-st(t) =
        assign(n,v) ;
         v : int-expr(s-right-part(t)) ;
         n : eval-ref(s-left-part(t)) ;

I17 assign(n,v) =
        s-dn: μ(DN; <n: convert(v,n(AI))>)
```

```
I18 int-goto-st(t) =
      goto-1(s-label(t)(ENV)(DN))

I19 goto-1(label-dn) =
      is-null(BA) → error        % stop at outermost block
      s-block-activation(label-dn) = BA →
          goto-2(s-st-loc(label-dn))          % local goto
      T →  s-d: u(D; <s-c: goto-1(label-dn)>)
           s-c: exit                          % cut stack

I20 goto-2(stlocs) =
      (∃ list) length(list) ≥ 1 &
               ci-indexlist(s-ci(CI))^list = stlocs  →
          goto-3((ι list)(ci-indexlist(s-ci(CI))^list
                          = stlocs)
      T →  s-ci: s-ci(CI)
           s-c : goto-2(stlocs)

I21 ci-indexlist(ci) =
      is-null(ci) → null
      T → ci-indexlist(s-ci(ci))~<s-index(ci)>

I22 goto-3(indexlist) =
      is-null(tail(indexlist)) → goto-4(index_1)
      T → s-ci: μ_0(<s-text: s-st(st_1)>,<s-c : int-next-st>,
                    <s-ci: μ(CI;<s-index: index_1>)>)
           s-c : goto-3(tail(indexlist))
    where: index_1 = head(indexlist) ,
           st_1 = take-st(index_1,s-text(CI))

I23 goto-4(index) =
      s-ci: μ(CI; <s-index: index>)
      s-c:  continue

I24 int-proc-call(t) =
      length(arg-list_t) = length(p-list_t) →
       s-env: μ(env_t; { arg-dn(elem(i) p-list_t ,elem(i)
                                                 arg-list_t)
                       : 1≤ i≤ length(p-list_t)} )
       s-d  : stack
       s-c  : exit ;
              int-st(st_t)
      T → error
    where: n_t=(s-id(t))(ENV) , p-list_t=s-param-list(s-attr
                                              (n_t(DN))) ,
           env_t=s-env(n_t(DN)) , arg-list_t=s-arg-list(t) ,
           st_t =s-st(s-attr(n_t(DN)))
```

```
I25 arg-dn(fp,ap) =
      s-value-opt(fp)=VAL v is-const(ap) -> %call-by-value
          PASS: μ0(<s-par(fp):n>);
           assign(n,v) ;
            v: int-expr(ap) ;
            n: unique-name ;
      is-var(ap) ->                    % call-by-reference
          PASS: μ0(<s-par(fp): ap(ENV)>)
      T ->                             % make a thunk
          PASS: μ0(<s-par(fp): n>);
           s-at: μ(AT; <n:THUNK>);
            s-dn: μ(DN; <n: μ0(<s-attr:ap>,<s-env:ENV>) );
             n: unique-name

I26 stack =
      s-d: u0(<s-env:ENV>,<s-c:C>,<s-d:D>,<s-ci:CI>,
              <s-block-activation:BA>)

I27 exit =
      s-env : s-env(D)
      s-c   : s-c(D)
      s-d   : s-d(D)
      s-ci  : s-ci(D)
      s-block-activation : s-block-activation(D)

I28 int-expr(t) =
      is-binary(t) -> int-bin-op(s-op(t),a,b) ;
                       a: int-expr(s-rd1(t)) ;
                       b: int-expr(s-rd2(t));
      is-unary(t)  -> int-un-op(s-op(t),a) ;
                       a: int-expr(s-rd(t)) ;
      is-func-call(t) & at_t=FUNC ->
                      pass-value(n) ;
                       int-func-call(t,n) ;
                        n : unique-name
      is-var(t) & is-var-attr(n_t(AT)) & is-value(n_t(DN)) ->
                      PASS: n_t(DN)
      is-var(t) & is-var-attr(n_t(AT)) & is-thunk-dn
                                          (n_t(DN)) ->
                      pass-value(n) ;
                       int-thunk-val(t,n) ;
                        n : unique-name ;
      is-const(t) -> PASS: value(t)
      T -> error
    where: n_t=t(ENV) , at_t= ((s-id(t))(ENV))(AT) ,
           and value, int-bin-op, and int-un-op are
               not further specified.
```

```
I29 pass-value(n) =
        PASS: n(DN)

I30 int-func-call(t,n) =
        length(arg-list_t) = length(p-list_t) ->
         s-env: μ(env_t; {arg-dn(elem(i) p-list_t ,
                                 elem(i) arg-list_t)
                          : 1≤i≤length(p-list_t)} )

         s-d: stack
         s-c: exit ;
               assign(n,v) ;
                v : int-expr(expr_t) ;
                 int-st(st_t) ;

        T -> error
     where n_t=(s-id(t))(ENV) , dn_t=n_t(DN) , env_t=s-env(dn_t),
           p-list_t=s-param-list(s-attr(dn_t)) ,
           arg-list_t=s-arg-list(t) ,
           st_t=s-st(s-attr(dn_t)) , expr_t=s-expr(s-attr(dn_t)).

I31 int-thunk-val(t,n) =
        s-env: s-env(t(ENV)(DN))
        s-d  : stack
        s-c  : exit ;
                s-dn: μ(DN; <n:v>) ;
                 v: int-expr(s-attr(t(ENV)(DN))) ;

I32 eval-ref(t) =
        is-var(t) & is-value(t(ENV)(DN)) -> PASS: t(ENV)
        is-var(t) & is-thunk-dn(t(ENV)(DN)) ->
            PASS : n(DN);          % the returned name
             int-thunk-ref(t,n) ;
              n : unique-name ;
        T -> error                 % all other sorts of
                                     expressions

I33 int-thunk-ref(t,n) =
        s-env: env_t
        s-d  : stack
        s-c  : exit ;
                s-dn: μ(DN; <n:r>) ;
                 r: eval-ref(expr_t) ;
     where: n_t= t(ENV) , env_t = s-env(n_t(DN)) ,
            expr_t = s-attr(n_t(DN))

I34 length(L) =
        is-null(L) -> 0
        T -> (ιi) (elem(i,L)≠null & elem(i+1,L)=null)

I35 concatenation
        L_1 ⌢ L_2 = μ(L_1;{<elem(length(L_1)+i):elem(i,L_2)>
                           for 1≤ i ≤ length(L_2)} )
```

Appendix J

Abstract syntax for the BASIS definition of ALEX

(assumed to be produced from the concrete syntax of Appendix A by a translation procedure very similar to that of Appendix D)

A1 <program> ::= <block>

A2 <block> ::= <declaration-list><statement-list>

A3 <declaration> ::= <identifier><attribute>

A4 <attribute> ::= <variable-attr>/<procedure-attr>/
<label-attr> /<function-attr>

A5 <variable-attr> ::= <type>

A6 <procedure-attr> ::= <parameter-list><statement>

A7 <function-attr> ::= <parameter-list><statement>
<type><expression>

A8 <type> ::= INT / REAL / LOG

A9 <parameter> ::= <value-option><type><identifier>

A10 <value-option> ::= VALUE / <empty>

A11 <empty> ::=

A12 <label-attr> ::= <statement-designator>

A13 <statement> ::= <assignment-stmt>/<conditional-stmt>/
<goto-stmt>/<procedure-call>/<block>/
<statement-list>/<empty>

A14 <assignment-stmt> ::= <variable><expression>

A15 <goto-stmt> ::= <identifier>

A16 <conditional-stmt> ::= <choice><then-stmt><else-stmt>

A17 <choice> ::= <expression>

A18 <then-stmt> ::= <statement>

A19 <else-stmt> ::= <statement>

A20 <procedure-call> ::= <identifier><argument-list>

A21 <argument> ::= <expression>

A22 <expression> ::= <constant>/<variable>/<binary>/
<unary>/<function-call>

A23 <constant> ::= <value>

A24 <variable> ::= <identifier>

A25 <binary> ::= <expression><binary-operator><expression>

A26 <unary> ::= <unary-operator><expression>

A27 <function-call> ::= <identifier><argument-list>

A28 <binary-operator> and <unary-operator> are not further specified.

Appendix K

Machine states for the BASIS definition

M1 ⟨machine-state⟩ ::= ⟨program⟩⟨control-state⟩⟨program-state⟩

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

M3 ⟨operation⟩ ::= ..[1)]

M4 ⟨program-state⟩ ::= ⟨block-state-list⟩

M5 ⟨block-state⟩ ::= ⟨environment-table⟩
⟨statement-control⟩
⟨statement-designator⟩
⟨enclosing-block⟩
⟨returned-value⟩

M6 ⟨environment-table⟩ ::= ⟨identifier-possession-list⟩

M7 ⟨identifier-possession⟩ ::= ⟨identifier⟩⟨cell-designator⟩

M8 ⟨statement-control⟩ ::= ⟨operation-list⟩

M9 ⟨enclosing-block⟩ ::= ⟨block-state-designator⟩

M10 ⟨returned-value⟩ ::= ⟨value⟩/⟨cell-designator⟩/ null

M11 ⟨cell⟩ ::= ⟨denotation⟩⟨dn-type⟩

M12 ⟨dn-type⟩ ::= INT / REAL / LOG / LABEL /
PROC / FUNC / THUNK

M13 ⟨denotation⟩ ::= ⟨value⟩/⟨label-dn⟩/⟨procfun-dn⟩/
⟨thunk-dn⟩ / null

M14 ⟨value⟩::=TRUE/FALSE /⟨real-value⟩/⟨integer-value⟩

M15 ⟨real-value⟩ ::= ..[2)]

M16 ⟨integer-value⟩ ::= ..[2)]

M17 ⟨label-dn⟩ ::= ⟨block-state-designator⟩⟨statement-designator⟩

M18 ⟨procfun-dn⟩ ::= ⟨environment-table⟩⟨procfun-attr⟩

M19 ⟨procfun-attr⟩ ::= ⟨procedure-attr⟩/ ⟨function-attr⟩

M20 ⟨thunk-dn⟩ ::= ⟨environment-table⟩⟨expression⟩

Notes 1 The exact structure of ⟨operation⟩ is left unformalised and unspecified, but should be sufficient to represent all the operations and their local variables, etc., of Appendix L

2 Real-values and integer-values are not further specified.

3 A designator is a member of a class of objects which has the property that it can uniquely point to any node in either the ⟨machine-state⟩ or ⟨program⟩ trees, and is also capable of indicating when the node to which it points has been deleted.

Any tree which contains a single designator as a component is said to designate the tree to which the designator points.

Appendix L

Interpretation program for the BASIS definition

B1 interpret(p)
where p is a <program>

Step 1. Perform initialise-interpretation-state(p).
Step 2. Perform interpret-block.

B2 initialise-interpretation-state(p)
where p is a <program>

Step 1. Let bd be a <statement-designator> designating the <block> in p.
Step 2. Append to the machine-state the tree
<program-state>:
<block-state-list>:
<block-state>:
<statement-designator>: bd.

B3 interpret-block

Step 1. Let sd be a <statement-designator> of the first <statement> of the <block> designated by the current <statement-designator>.
Step 2. Perform stack(null,<operation> for prologue,sd).

B4 prologue

Step 1. For each <declaration>d of the current <block>, perform steps 1.1 to 1.3:
Step 1.1. Let id be the <identifier> immediately contained in d, and let attr be the <attribute> of id in d.
Step 1.2. Peform initialise-declaration(attr) to obtain a <cell>c; let cd be a <cell-designator> designating c.
Step 1.3. Append to the current <identifier-possession-list> an <identifier>:id <cell-designator>cd.
Step 2. Replace the current <statement-control> by an <operation> for continue.

B5 initialise-declaration(attr)
where attr is an <attribute>

Case 1. attr immediately contains a <variable-attr>
Step 1.1 Return a <cell>:<denotation>:null
<dntype>:attr.
Case 2. attr immediately contains a <procedure-attr>
Step 2.1 Let type be PROC.
Step 2.2 Let et be a copy of the current <environment-table>.
Step 2.3 Return a
<cell>:<denotation>:<procfun-dn>:
<environment-table>:et
<procfun-attr>:attr
<dn-type>:type.
Case 3. attr immediately contains a <function-attr>
Step 3.1 Let type be FUNC.
Step 3.2 Go to step 2.2

Case 4. attr immediately contains a <label-attr>
Step 4.1. Let lsd be the <statement-designator> in attr.
Let bsd be a <block-state-designator>
designating the current <block-state>.
Step 4.2. Return a
<cell>:<denotation>:<label-dn>:
<block-state-designator>:bsd
<statement-designator>:lsd
<dn-type>: LABEL.

B6 next-statement
This is normally the last action of each statement, and advances the current <statement-designator> to designate the next statement.

Step 1. Let st be the <statement> designated by the current <statement-designator>. Let stl be the <statement-list> which contains st, but does not contain any <statement-list> which also contains st.
Step 2. Let st2 be that immediate component of stl which either contains st or is exactly st. Then let st3 be that <statement> which immediately follows st2 as an immediate component of stl.
Step 3. If such a st3 is non-null, set the current <statement-designator> to designate it, and return. Otherwise
Step 4. Perform end-of-statement-list (stl).

B7 end-of-statement-list(stl)
where stl is a <statement-list>

Step 1. Let n be the node which immediately contains stl.
Case 1.1. n is a <block>, or is a <statement> contained in a <procedure-attr>.
Step 1.1.1 Perform exit.
Case 1.2. n is a <statement> contained in a <function-attr> fa.
Step 1.2.1. Let e be the <expression> in fa.
Step 1.2.2. Perform return-value(e).
Case 1.3. (otherwise)
Step 1.3.1. Set the current <statement-designator> to designate n.
Step 1.3.2. Perform next-statement.

B8 continue
This operation is the "driver" which initiates execution of the currently designated statement.

Step 1. Perform interpret-statement.
Step 2. Go to step 1.

B9 interpret-statement

Step 1. Let st be the node immediately contained by the <statement> designated by the current <statement-designator>.
Case 1.1. st is an <assignment-stmt>.
Perform interpret-assignment-statement(st).
Case 1.2. st is a <conditional-stmt>.
Perform interpret-if-statement(st).
Case 1.3. st is a <goto-stmt>.
Perform interpret-goto-statement(st).
Case 1.4. st is a <procedure-call>.
Perform interpret-procedure-call(st).

Case 1.5. st is a ⟨statement-list⟩.
Perform interpret-statement-list(st).
Case 1.6. st is a ⟨block⟩
Perform interpret-block.
Case 1.7. st is ⟨empty⟩
Perform next-statement.

B10 interpret-assignment-statement(ast)
where ast is an ⟨asignment-stmt⟩

Step 1. Let var be the ⟨variable⟩ & e the ⟨expression⟩ in ast.
Step 2. Perform evaluate-reference(var) to obtain cd.
Step 3. Perform evaluate-expression(e) to obtain ⟨value⟩ val.
Step 4. Perform assign(cd,val).
Step 5. Perform next-statement.

B11 assign(cd,val)
where cd is a ⟨cell-designator⟩ and val a ⟨value⟩

Step 1. Let c be the ⟨cell⟩ designated by cd.
Step 2. Let type be the ⟨dn-type⟩ of c.
Step 3. Perform convert(val,type) to obtain val2
Step 4. Replace the ⟨value⟩ component of c by val2.

B12 interpret-if-statement(st)
where st is a ⟨conditional-stmt⟩

Step 1. Let e be the ⟨expression⟩ of the ⟨choice⟩ of st.
Step 2. Perform evaluate-expression(e) to obtain val.
Step 3. Let t be convert(val,LOG)
Case 3.1. t is TRUE
Let s be the ⟨then-stmt⟩ of st.
Case 3.2. t is FALSE
Let s be the ⟨else-stmt⟩ of st.

Step 4. Set the current ⟨statement-designator⟩ to
designate the ⟨statement⟩ of s.

B13 interpret-goto-statement(gs)
where gs is a ⟨goto-stmt⟩

Step 1. Let id be the ⟨identifier⟩ that is in the ⟨label⟩ in gs.
Step 2. Perform id-ref(id), and obtain the designated
cell c, which must be of the form

⟨cell⟩:⟨denotation⟩:⟨label-dn⟩:
⟨block-state-designator⟩ bsn
⟨statement-designator⟩ lab
⟨dn-type⟩: LABEL.

Step 3. The ⟨block-state-list⟩ must contain a ⟨block-state⟩
designated by bsn (its corresponding block
contains the ⟨statement⟩ designated by lab).

Case 3.1. bs is the current ⟨block-state⟩
Perform local-goto(lab).

Case 3.2. bs is not the current ⟨block-state⟩
Step 3.2.1. Replace the ⟨statement-control⟩ of bs
by an ⟨operation⟩ for local-goto(lab).

Step 3.2.2. For each ⟨block-state⟩,b that occurs
after bs in the ⟨block-state-list⟩
(including the current block),
replace the ⟨statement-control⟩ of b
by an ⟨operation⟩ for exit.

B14 local-goto(dest)
where dest is a ⟨statement-designator⟩

Step 1. Replace the current ⟨statement-designator⟩ by dest.
Step 2. Perform continue.

B15 interpret-procedure-call(pc)
where pc is a ⟨procedure-call⟩

Step 1. Let id be the ⟨identifier⟩ and al the ⟨argument-list⟩ immediately contained in pc.
Step 2. Perform denotation(id) to give pd, which must be a
⟨procfun-dn⟩:⟨environment-table⟩,env
⟨procfun-attr⟩:⟨procedure-attr⟩:
⟨parameter-list⟩:pl;
⟨statement⟩:s.
Step 3. Perform install-arguments(al,pl,env).
Step 4. Let std be a ⟨statement-designator⟩ designating the ⟨statement⟩ s.
Step 5. Perform stack(env,⟨operation⟩ for interpret-statement,std)

B16 install-arguments(al,pl,env)
where al is an ⟨argument-list⟩, pl a ⟨parameter-list⟩, and env an ⟨environment-table⟩

Step 1. If length(al) does not equal length(pl) then error.
Step 2. Otherwise,
for each i from 1 to length(al) perform
Step 2.1. Let fp be the i'th component of pl, and
ap be the i'th component of al.
Step 2.2. Perform argument-dn(ap,fp), and
append the result to the ⟨identifier-possession-list⟩ contained in the environment-table env.

B17 argument-dn(ap,fp)
where ap is an ⟨argument⟩ and fp is a ⟨parameter⟩.

Step 1. Let aexp be the expression of ap.
Case 1.1. If aexp immediately contains a ⟨constant⟩, or the ⟨value-option⟩ component of fp is VALUE:

Step 1.1.1. Peform evaluate-expression(aexp), giving val. Let c be a

⟨cell⟩:⟨denotation⟩:⟨value⟩ val ;
⟨dn-type⟩:null.

Let cd be a ⟨cell-designator⟩ designating c.

Case 1.2. If aexp immediately contains a ⟨variable⟩, and id.
Perform id-ref(id) giving cd.

Case 1.3. (otherwise make a thunk)
Let env be a copy of the current ⟨environment-table⟩. Let c be a

⟨cell⟩:⟨denotation⟩:⟨thunk-dn⟩:
⟨environment-table⟩ env
⟨expression⟩ aexp
⟨dn-type⟩: THUNK.

Let cd be a ⟨cell-designator⟩ designating c.

Step 2. Return an ⟨identifier-possession⟩:
⟨identifier⟩ fp
⟨cell-designator⟩ cd.

B18 stack(env,op,std)
where env is an ⟨environment-table⟩, op an ⟨operation⟩, and std a ⟨statement-designator⟩

Step 1. Let bsd be a ⟨block-state-designator⟩ designating the current ⟨block-state⟩.
Step 2. Append to the ⟨block-state-list⟩ a

```
⟨block-state⟩: ⟨environment-table⟩ env
               ⟨statement-control⟩:
                 ⟨operation-list⟩:
                   ⟨operation⟩:  op ;;;
               ⟨statement-designator⟩: std
               ⟨enclosing-block⟩: bsd
               ⟨returned-value⟩: null.
```

B19 exit

Delete the current ⟨block-state⟩ from the ⟨block-state-list⟩.

B20 interpret-statement-list(stl)

Step 1. Set the current ⟨statement-designator⟩ to designate the first ⟨statement⟩ in stl.
Step 2. Perform continue.

B21 evaluate-expression(e)

Step 1. Let ec be the node immediately contained in e.

```
Case 1.1. ec is a ⟨binary⟩: ⟨expression⟩ e1
                            ⟨binary-op⟩ op ⟨expression⟩ e2:
            Perform evaluate-expression(e1) giving a,
                    evaluate-expression(e2) giving b.
            Return evaluate-binary-op(op,a,b)

Case 1.2. ec is a ⟨unary⟩: ⟨unary-operator⟩ op
                           ⟨expression⟩ e1.
            Perform evaluate-expression(e1), giving a.
            Return  evaluate-unary-op(op,a).

Case 1.3. ec is a ⟨function-call⟩
            Perform evaluate-function-call(ec)
            Return the current ⟨returned-value⟩.

Case 1.4. ec is a ⟨variable⟩: ⟨identifier⟩ id.

   Case 1.4.1. Denotation(id) is a ⟨value⟩ val
                 Return val.

   Case 1.4.2. Denotation(id) is a ⟨thunk-dn⟩ thdn
                 Perform evaluate-thunk-val(thdn).
                 Return the current ⟨returned-value⟩.
Case 1.5. ec is a ⟨constant⟩
            Return its ⟨value⟩.
```

B22 id-ref(id)
where id is an ⟨identifier⟩

Step 1. Search the current ⟨identifier-possession-list⟩ for the last ⟨identifier-possession⟩ ip containing id.
Step 2. Return the ⟨cell-designator⟩ also contained in ip.

B23 denotation(id)

Return the ⟨denotation⟩ component of the ⟨cell⟩ designated by id-ref(id).

B24 evaluate-function-call(fc)
where fc is a ⟨function-call⟩

Step 1. Let id be the ⟨identifier⟩ and al the ⟨argument-list⟩ immediately contained in fc.
Step 2. Perform denotation(id) to give fd, which must be a

```
⟨procfun-dn⟩:⟨environment-table⟩ env
              ⟨procfun-attr⟩:
                 ⟨function-attr⟩:
                    ⟨parameter-list⟩  pl
                    ⟨statement⟩       s
                    ⟨type⟩            t
                    ⟨expression⟩      re.
```

Step 3. Perform install-arguments(al,pl,env).
Step 4. Let std be a ⟨statement-designator⟩ designating the ⟨statement⟩ s.
Step 5. Perform stack(env,⟨operation⟩ for interpret-statement, std).

B25 evaluate-thunk-value(dn)
where dn is a⟨thunk-dn⟩

Step 1. Let et be the ⟨environment-table⟩, and e the ⟨expression⟩ in dn.
Step 2. Perform stack(et,⟨operation⟩ for return-value(e),null).

B26 return-value(e)
where e is an ⟨expression⟩

Step 1. Perform evaluate-expression(e) and attach the result to the ⟨returned-value⟩ component of the ⟨block-state⟩ designated by the current ⟨enclosing-block⟩.
Step 2. Perform exit.

B27 evaluate-reference(var)
where var is a ⟨variable⟩

Step 1. Let id be the ⟨identifier⟩ of var.
Case 1.1. If denotation(id) is a ⟨value⟩
Return id-ref(id).
Case 1.2. If denotation(id) is a ⟨thunk-dn⟩ thdn
Perform evaluate-thunk-reference(thdn).
Return the current ⟨returned-value⟩
(a ⟨cell-designator⟩).

B28 evaluate-thunk-reference(dn)
where dn is a ⟨thunk-dn⟩

Step 1. Let et be the ⟨environment-table⟩, and e the ⟨expression⟩ in dn.
Step 2. Perform stack(et,⟨operation⟩ for return-reference, null).

B29 return-reference(e)
where e is an ⟨expression⟩

Step 1. Perform evaluate-reference(e) and attach the result to the ⟨returned-value⟩ component of the ⟨block-state⟩ designated by the current ⟨enclosing-block⟩.
Step 2. Perform exit.

Note: evaluate-binary-op and evaluate-unary-op are not further specified.

Appendix M

Abstract syntax tree-structures for the LFMS definition of ALEX

AL1 program : block
AL2 block : declaration-list , stmt-list

AL3 declaration = cell / procedure / function / label
AL4 cell : value , type
AL5 value = real-value / integer-value / true / false / undef
AL6 type = int / real / log
AL7 procedure : parameter-list , stmt
AL8 function : parameter-list , stmt , type , expression
AL9 parameter : spec , type [, argument] / cell
AL10 spec = value / name
AL11 label : stmt-designator

AL12 stmt = assignment-stmt / if-stmt / goto-stmt /
procedure-call / stmt-list / block / null
AL13 assignment-stmt : variable , expression
AL14 if-stmt : expression , stmt , stmt
AL15 goto-stmt : label-designator
AL16 procedure-call : procid , argument-list
AL17 procid = procedure-designator
AL18 argument = expression

AL19 expression = constant / variable / unary / binary /
function-call
AL20 constant = value
AL21 variable = cell-designator / parameter-designator
AL22 unary : unary-op , expression
AL23 binary : expression, binary-op , expression
AL24 function-call : funcid , argument-list
AL25 funcid = function-designator

Notes 1. Real-value, integer-value, unary-op, and binary-op are not further specified.
2. undef is a terminal meaning that no value has been defined, and attempt to use it in an arithmetic expression will result in an error.
3. During execution, there will sometimes be further subnodes to call statements.

Appendix N

Transformation rules for the LEMS definition

L1 EXEC program: block → program: EXEC block

L2 program: END block → END program: block S(STOP).

L3 EXEC block: stmt-list → block: EXEC stmt-list

L4 block: END stmt-list → END block: stmt-list

L5 EXEC stmt-list: stmt .. → stmt-list: EXEC stmt ..

L6 .. END $stmt_1$,$stmt_2$ .. → .. $stmt_1$, EXEC $stmt_2$ ..

L7 stmt-list: .. END stmt → END stmt-list: .. stmt

L8 EXEC assignment-stmt: variable,expression →
assignment-stmt: variable, EXEC expression

L9 assignment-stmt: variable=$cell_1$-designator,VAL(x) expression
& $cell_1$: value=y , type →
END assignment-stmt: variable , expression
& $cell_1$: value=convert(x,type) , type

L10 EXEC if-stmt: expression,$stmt_1$,$stmt_2$ →
if-stmt: EXEC expression , $stmt_1$, $stmt_2$

L11 if-stmt: VAL(true) expression , $stmt_1$, $stmt_2$ →
if-stmt: expression , EXEC $stmt_1$, $stmt_2$

L12 if-stmt: VAL(false) expression, $stmt_1$, $stmt_2$ →
if-stmt: expression , $stmt_1$, EXEC $stmt_2$

L13 if-stmt: .. END stmt .. → END if-stmt: .. stmt ..

L14 EXEC goto-stmt: $label_1$-designator & $label_1$: $stmt_2$-designator →
goto-stmt: $label_1$-designator & EXEC $stmt_2$

L15 EXEC empty → END empty

L16 EXEC procedure-call:
procid=$procedure_1$-designator , argument-list
& $procedure_1$: parameter-list , stmt →
procedure-call:
procid , argument-list ,
MATCH (parameter-list-copy + argument-list-copy) ,
stmt-copy

L17 MATCH parameter-list → (MATCH parameter)-list

L18 MATCH value,type,argument=expression →
value,type, EXEC expression

L19 value,type, VAL(x) expression →
MATCHED cell: value=convert(x,type) , type

L20 MATCH name,type,argument → MATCHED name,type,argument

L21 variable=$parameter_1$-designator
& $parameter_1$: name,type,argument=expression →
expression-copy

L22 (MATCHED parameter)-list → MATCHED-PRMS

L23 MATCHED-PRMS , stmt → EXEC stmt

L24 procedure-call: procid , argument-list , END stmt →
END procedure-call: procid , argument-list

L25 EXEC constant=value=x → VAL(x) constant

L26 EXEC variable=cell$_1$-designator & cell$_1$: value=x , type →
VAL(x) variable

L27 EXEC unary: unary-op,expression →
unary: unary-op, EXEC expression

L28 unary: unary-op:op,VAL(x) expression →
VAL(op(x)) unary: unary-op: op ,expression

L29 EXEC binary: expression$_1$, binary-op:op , expression$_2$ →
binary: EXEC expression$_1$, binary-op:op , EXEC expression$_2$

L30 binary: VAL(x) expression$_1$,binary-op:op, VAL(y) expression$_2$
→ VAL(op(x,y)) binary: expression$_1$, binary-op:op ,
expression$_2$

L31 EXEC function-call:
funcid=function$_1$-designator , argument-list
& function$_1$: parameter-list,stmt,type,expression →
function-call:
funcid , argument-list ,
MATCH (parameter-list-copy + argument-list-copy),
stmt-copy , type , expression-copy

L32 END stmt , type , expression →
stmt , type , EXEC expression

L33 function-call:
funcid,argument-list,stmt,type, VAL(x) expression →
VAL(convert(x,type)) function-call: funcid,argument-list

-list
-designator } suffixes with special meanings
-copy