INTRODUCTION TO PROGRAMMING SCIENCE - PART I:

SYNTAX AND SEMANTICS OF PROGRAMMING LANGUAGES

by

W. D. Maurer

Memorandum No. ERL-M368

December 1972

ELECTRONICS RESEARCH ABORATORY

College of Engineering
University of California, Berkeley
94720

Note

This memorandum consists of the first draft of Part I of a projected book, entitled, "Introduction to Programming Science."

Research sponsored by the National Science Foundation, Grant GJ-31612.

C O N T E N T S

I N T R O D U C T I O N

Programming science, as the term is used in this book, is the science, as opposed to the art, of constructing programs for digital computers in algorithmic languages. It is the "mathematical science of computation" whose development was predicted in [McCarthy 63].*

There have been many attempts, over the years, to construct a theory of algorithms, as contrasted with a theory of programs. Since an algorithm may be programmed in any one of various languages, but must be programmed in some way in order to be analyzed, all that is necessary in constructing a theory of algorithms is to construct some programming language or type of machine which is powerful enough to encompass all algorithms of the type being studied, and then to study an arbitrary algorithm as programmed in this way. Historically, the theory of Turing machine computability is the earliest of these theories; recently, this theory has been significantly extended by research on time and space limitations of computations (theory of computational complexity). This theory was later recast by Kaphengst, and again by Shepherdson and Sturgis, in a form in which the type of machine used more closely resembles a modern digital computer. Markov's theory of algorithms is another theory of this kind; although Markov uses algorithms of a specific type, whereas Turing uses machines, the essential characteristic remains, namely that an algorithm, in order to be analyzed in either of these ways, must be recast (either as a Markov algorithm or as a Turing machine program).

More recent theoretical studies have also often tended to follow this "algorithmic" philosophy. Thus in Floyd's work on program correctness, and in the work of Floyd's student, James King, on computer-aided program verification, it is assumed that all programs to be verified are of a particular, relatively simple form. C. A. R. Hoare has axiomatized the verification conditions which appear in proofs of program correctness; Hoare's axioms, however, are true in a programming language only in the absence of side effects and certain other general features. There are also at least two distinct theories of "program schemes," one due to Yanov and the other due to Luckham, Park, and Paterson; in either case, a particular algorithm to be analyzed must be cast in a special form in order for it to be an interpretation of such a program scheme.

In a theory of programs, the program, as well as the algorithm, is assumed to be given. Various questions may be asked about a program, such as whether it is correct, whether it terminates, or whether it is equivalent to some other program. We are then concerned with developing methods of proving such facts; we are concerned with formulating theorems which simplify the job of proving them or which facilitate the construction of efficient and easy-to-use computer aids to proving them.

Our present approach is divided into three parts. In the first part, we take up the syntax and semantics of programming languages. If a program P is written in a language L, and we wish to prove facts about P, we must know both the syntax and the semantics of L, and we must know them in a form to which our theory is applicable. The syntactic approach which we use is basically Backus-Naur Form (BNF). The semantic approach is of our own construction, based principally on the state vectors of McCarthy and the synthesized and inherited semantic attributes of Knuth.

In the second part, we introduce the fundamental methods of proving assertions about programs: correctness, termination, and equivalence. In the third part, we proceed to more advanced topics: the correctness of compilers, the correctness of self-modifying programs, the construction of computer aids to verification, and the semantics of data structures.

* * * * * * * * * * * * * * * * * *

This book has been very carefully constructed so as to be readable by people with a wide variety of backgrounds, both in mathematics and in programming.

To take mathematics first: Programming science is a mathematical science, and, like every other mathematical science (even mathematical logic), can be, and is in fact here, based on the concepts of set and function as primitive, undefined terms. Many mathematical sciences, such as group theory, are based on set theory but only in a trivial way; most group theory is done without much thought about abstract sets. This is definitely not the case in the theory discussed here. Sets, functions, restrictions of functions, composition of functions, and cartesian products are found in great profusion, and the semantic rules by which a programming language is described are full of references to finite and infinite sets, union and intersection of sets, functions as sets of ordered pairs, and so on. Nevertheless, this book has been constructed so as to be readable by those who have never before worked with a mathematical science, such as, for example, the majority of practicing programmers. For this reason, the usual section in which all of the mathematics necessary to understand a subject is neatly collected and summarized will not be found in this book. The definitions of set, function, restriction

of a function, and so on, are spread out over several early chapters; and each one is accompanied by an immediate application of it to the developing theory. Those who are familiar with higher mathematics may skim lightly over these sections, although they will need to understand thoroughly exactly how the set theory is used.

From the standpoint of programming, the prerequisites for reading the entire book are a knowledge of FORTRAN, ALGOL, at least one assembly language, and a smattering of higher-level languages, particularly LISP and SNOBOL. It is not, however, necessary to read the entire book in order to understand what programming science is about. Part I, with the exception of section 3-5 (which is not essential and may be omitted), requires only a reading knowledge of FORTRAN and ALGOL, and even this may be learned concurrently, if the reader is studying programming from an intuitive point of view at the same time. Part II requires even less than Part I in the way of programming knowledge; all the algorithms presented here should be obvious even to a non-programmer. Each of the sections of Part III has its own set of prerequisites, which should be relatively obvious upon reading the titles of the sections.

This book was written while the author was supported by National Science Foundation Grants GJ-821 and GJ-31612, and by the University of California as an Assistant Professor, including one summer during which the author held a Summer Faculty Fellowship given for the express purpose of research relating to this book.

P A R T O N E

S Y N T A X A N D

S E M A N T I C S O F

P R O G R A M M I N G

L A N G U A G E S

C H A P T E R O N E

C O N S T A N T S

1-1 Syntax of Constants

Consider the following strings of characters:

8

-12

99999999999999999999999

17B

0.27

.27

$621_{10}5$

10^5

50.

93B

Intuitively, each of these "looks like" a constant in some programming language. Some of them, however, will be considered, under certain circumstances, to be improperly formed. These circumstances involve the programming language being used, the particular interpreter, compiler, or other system being used to translate that language, and even the version number of this system as it is gradually modified and extended.

Programming science is concerned with exact statements about computer programs. The most fundamental exact statement we can make about a program is that it is properly formed. In order to make such a statement, we normally construct certain rules for well-formed

programs, and then require that any well-formed program must obey these rules. These are then called the syntax (or syntactic) rules of the given programming language or programming system.

The syntax rules of actual programming languages give a wide variety of answers to well-formedness questions even as applied to something as simple as a constant. Looking back at the ten examples above, we cannot find any one of them that is always, unequivocally, well-formed. The digit 8, for example, is not a constant in the assembly language of the PDP-8, where all integers not followed by D are taken as octal. In ALGOL,* -12 is a simple arithmetic expression, but it is not a constant, according to the ALGOL definition (for example, we cannot write 24/-12 in ALGOL, whereas for any constant $x \neq 0$ we can write 24/x). Many languages (although not ALGOL) specify 999999999999999999999999 to be improperly formed, because it will not fit into a computer word. The remaining examples either involve B (for "binary," not permitted in ALGOL), or else involve real numbers, which are, of course, not permitted on any computer or system which lacks floating-point or other real number handling facilities.

Most programming syntax rules are constructed in essentially the same way. This way was first widely publicized in connection with ALGOL, and is known, in its original form, as BNF (Backus-Naur Form; formerly, Backus "Normal" Form), honoring John Backus, who developed it, and Peter Naur, editor of the ALGOL report. BNF is a symbolic form of expressing certain types of syntax rules which may also be expressed in other ways. As an example, we shall now give a complete set of syntax rules for numbers. These rules are equivalent to those given in the ALGOL report.

---

*Throughout this book, "ALGOL," unless further qualified, refers to ALGOL 60 as modified in the Revised Report on the Algorithmic Language ALGOL 60, published in 1963.

A number is an unsigned number, optionally preceded by a sign (+ or -).

An unsigned number consists of a decimal number followed by an exponent part, where either the decimal number or the exponent part may be omitted.

The exponent part of an unsigned number consists of the subscript symbol $_{10}$ followed by an arbitrary (possibly signed) integer.

A decimal number consists of an unsigned integer followed by a decimal fraction, where either the unsigned number or the decimal fraction may be omitted.

A decimal fraction consists of a decimal point (.) followed by an arbitrary unsigned integer.

An integer is an unsigned integer, optionally preceded by a sign (+ or -).

An unsigned integer is an arbitrary sequence of digits (0, 1, 2, 3, 4, 5, 6, 7, 8, or 9).

The above rules are completely rigorous; in the following section, we shall see how they may be used to prove that 8, 99999999999999999999999, 0.27, .27, $621_{10}5$, and $_{10}5$ are properly formed unsigned numbers in ALGOL, that -12 is a properly formed number, and that 50. (for example) is not a properly formed number. However, as presented above, the syntax rules have the disadvantage that they are not very easily processable by computer themselves. In order to remedy this difficulty, we rewrite the rules in symbolic form, and it is this symbolic form that is known as BNF.

The variable quantities with which BNF is concerned (in the above rules, "number," "unsigned number," "exponent part," "decimal number," "decimal fraction," "integer," "unsigned integer," and "digit") are referred to, in the ALGOL report, as metalinguistic

variables. Each metalinguistic variable has a definition, in terms of other metalinguistic variables and (sometimes) characters, in this case 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, +, -, $_{10}$, and the period, all of which may be called "metalinguistic constants." In BNF, metalinguistic variables are enclosed in angle brackets < > , and the symbol ::= means "is defined as." Thus, for example, the definition of "decimal fraction" is given in the ALGOL report as

<decimal fraction> ::= . <unsigned integer>

In order to reduce confusion later on, we shall enclose all metalinguistic constants in quotes ' ' . Thus the above rule now reads

<decimal fraction> ::= '.' <unsigned integer>

Similarly, the definition of "exponent part" is

<exponent part> ::= '$_{10}$' <integer>

In both these rules, the words "followed by" are represented by a blank.

Another symbol used in BNF is the vertical line | , which means "or." This allows us to define a metalinguistic variable as either of several alternatives. For example, "digit" is defined as

<digit> ::= '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'

Most of the rest of our rules for constants may now be expressed without any further notational conventions. Thus "number" may be defined by

<number> ::= <unsigned number> | '+' <unsigned number> | '-' <unsigned number>

In words: A number is either an unsigned number, or '+' followed by an unsigned number, or '-' followed by an unsigned number. This definition clearly expresses the meaning of "optionally preceded by a sign." Similarly, the definitions of "integer," "unsigned number," and "decimal number" may be given as

<integer> ::= <unsigned integer> | '+' <unsigned integer> | '-' <unsigned integer>

<unsigned number> ::= <decimal number> | <exponent part> | <decimal number> <exponent part>

<decimal number> ::= <unsigned integer> | <decimal fraction> | <unsigned integer> <decimal fraction>

There remains the definition of "unsigned integer." Since the ALGOL report was published, several authors have made extensions to BNF, one of which embodies the idea of "arbitrary sequence." Thus by writing

<unsigned integer> ::= <digit>*

we could express, in this expanded notation, the statement that an unsigned integer is an arbitrary sequence of digits. This includes the null sequence, which is probably not what we want; but we can easily cure this by writing

<unsigned integer> ::= <digit> <digit>*

In this book, however, we shall use the original notation of the ALGOL report, and write

<unsigned integer> ::= <digit> | <unsigned integer> <digit>

This is a recursive syntax rule; it defines unsigned integers in terms of other unsigned integers. Recursive syntax rules are often

harder to visualize than non-recursive ones; for the moment, we shall merely note the fact that we can always define an "alpha" to be an arbitrary sequence of "beta's" by writing

<alpha> ::= <beta> | <alpha> <beta>

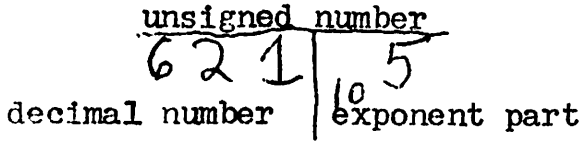
as we have done here. We can define the same thing by writing

<alpha> ::= <beta> | <beta> <alpha>

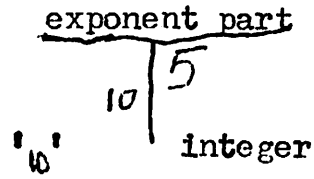
The distinction between these two methods will be taken up further in section 1-5.

1-2 Derivation Trees

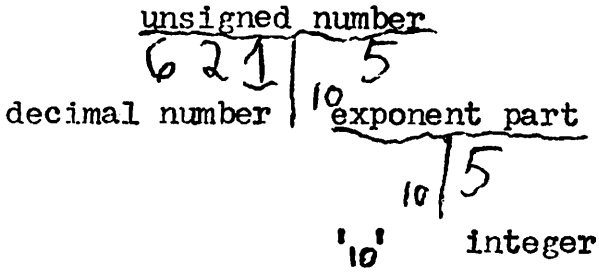
Using syntax rules such as those discussed above, we may analyze the syntax of any constant and break it down into its component parts. Let us consider, for example, the constant $621_{10}5$. This is short for 621×10^5, or 62,100,000. Its exponent part is $_{10}5$, and it also contains the decimal number 621. We may express this relationship by writing



This makes it easy to see that $621_{10}5$ is an unsigned number, according to the rule for unsigned numbers, provided that 621 is really a decimal number and that $_{10}5$ is an exponent part. Using the rule for exponent parts, we may write

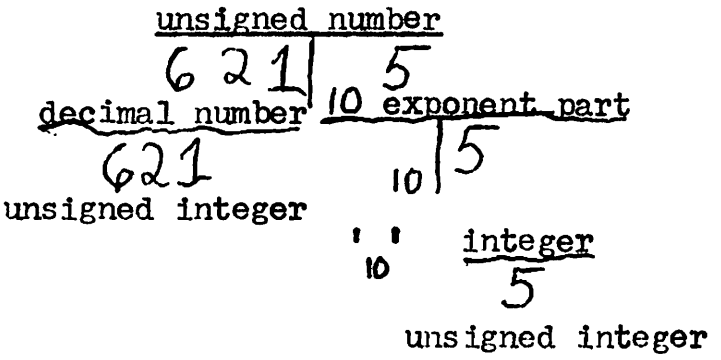


in analogy with the above. By putting these two diagrams together, we obtain

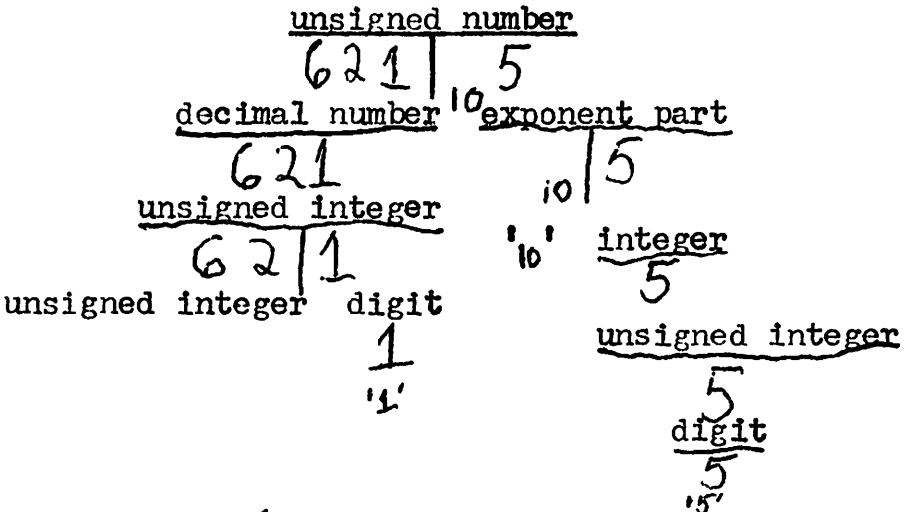


Let us carry this analysis further. Both 621 and 5 are clearly unsigned integers. One of our rules says that any unsigned integer is a decimal number, which is what we want to know concerning 621.

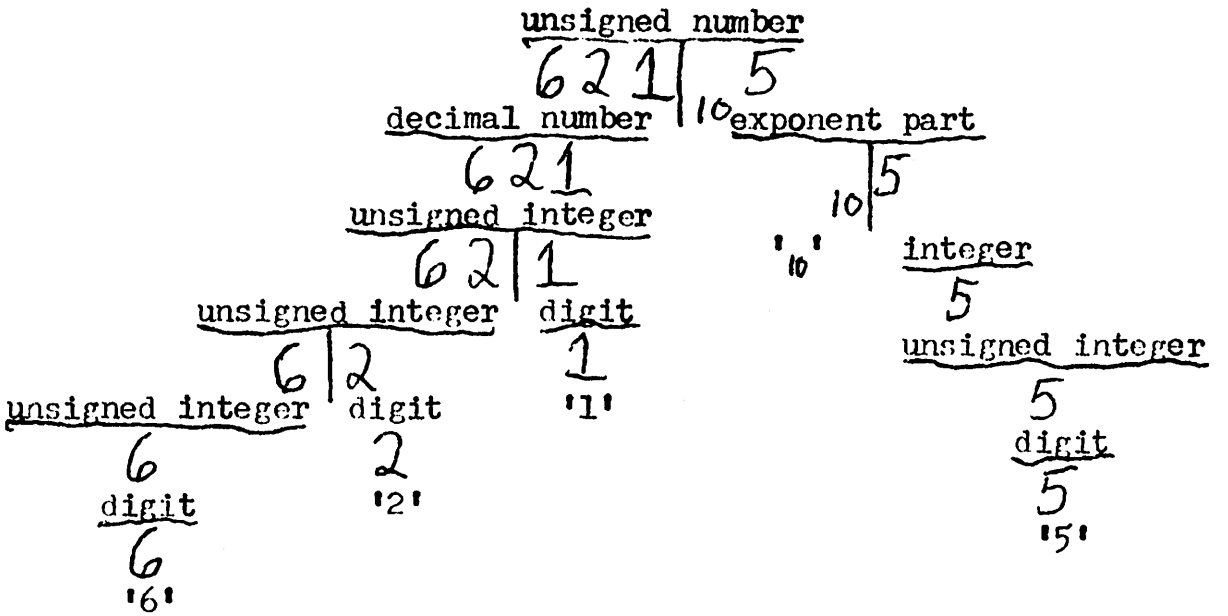
Another rule says that any unsigned integer is an integer, which takes care of 5. Expanding our diagram to incorporate these two facts, we obtain



Now let us consider our recursive rule for unsigned integers. In words, an unsigned integer is either a digit or an unsigned integer followed by a digit. The unsigned integer 5 is clearly of the first kind, since 5 is a digit. The unsigned integer 621 is of the second kind; it is the unsigned integer 62 followed by the digit 1. Thus we can further expand our diagram:

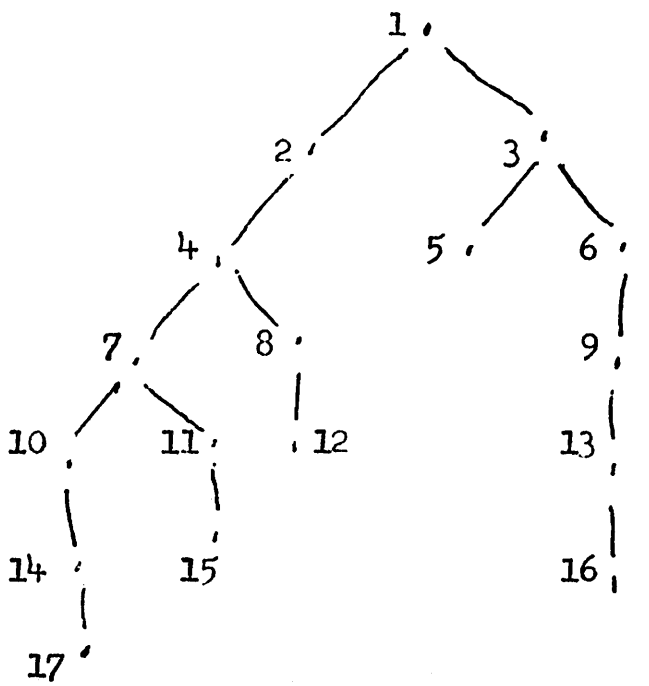


The unsigned number 62, in turn, is the unsigned number 6 followed by the digit 2. The unsigned number 6 is itself a digit. By incorporating these last two facts, we obtain a diagram which gives, in graphical form, a complete proof of the fact that $621_{10}5$ is an unsigned number according to our rules:



In the same way, if we started with any unsigned number (or integer, or unsigned integer, etc.), we could make a diagram like this one to show how it is constructed.

In order to analyze further this type of diagram, let us redraw the diagram above in a more abstract form:



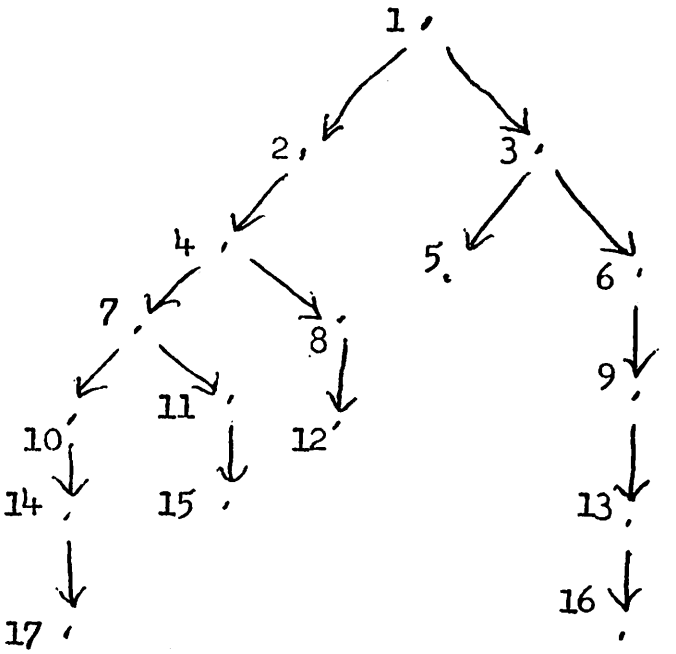
This is called a graph. It consists of points, called nodes, and line segments which join the nodes, called links. The correspondence

between the original diagram and the abstract diagram may be shown by means of the following table:

NODE NUMBER	METALINGUISTIC VARIABLE	CHARACTER STRING	METALINGUISTIC CONSTANT (IF ANY)
1	unsigned number	$621_{10}5$	-----
2	decimal number	621	-----
3	exponent part	$_{10}5$	-----
4	unsigned integer	621	-----
5	-----	-----	'$_{10}$'
6	integer	5	-----
7	unsigned integer	62	-----
8	digit	1	-----
9	unsigned integer	5	-----
10	unsigned integer	6	-----
11	digit	2	-----
12	-----	-----	'1'
13	digit	5	-----
14	digit	6	-----
15	-----	-----	'2'
16	-----	-----	'5'
17	-----	-----	'6'

Each link in a graph such as this one arises in a standard way from the definition of some metalinguistic variable. For example, consider the link from node 1 to node 3. This link is present because the definition of "unsigned number," which corresponds to node 1, contains a reference to "exponent part," which corresponds to node 3. Let us represent this relationship by an arrow pointing from node 1 to node 3. If we redraw the graph again, replacing all of its

links by arrows in this way, we obtain:

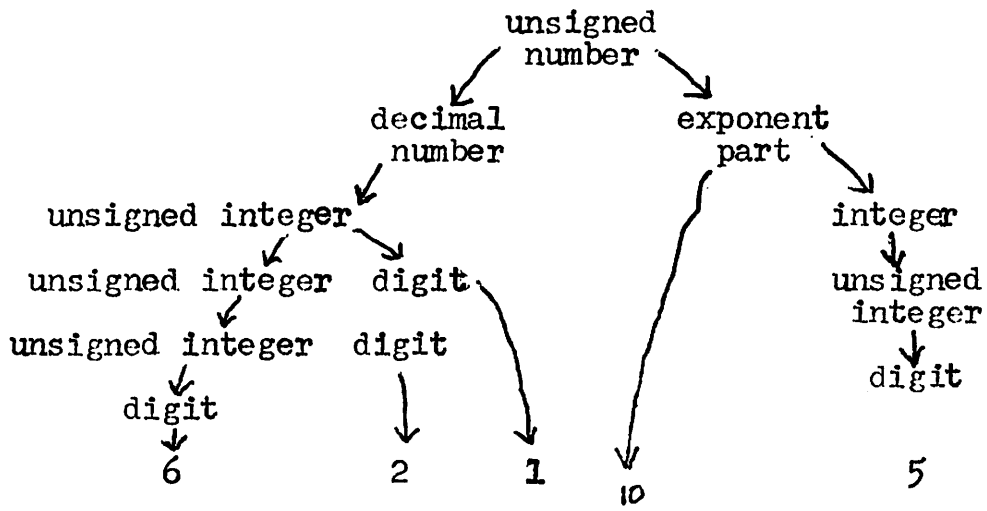


This is called a directed graph. In each case, if an arrow points from node X to node Y, then node X corresponds to some metalinguistic variable V, and node Y corresponds to a metalinguistic variable or constant which appears in the definition of V as it applies to the particular string being analyzed.

Every directed graph may have initial nodes and terminal nodes. An initial node is one with no arrows pointing to it; a terminal node is one with no arrows leading out of it. If we think of a train running along a directed graph in the direction of the arrows, then a terminal node is the "end of the line," so to speak, whereas an initial node is the point from which a journey logically starts. In the above graph, there is one initial node, namely node 1, while the terminal nodes are nodes 5, 12, 15, 16, and 17.

We may notice immediately that the terminal nodes of a graph which is constructed as above correspond precisely to the metalinguistic constants. This follows, in fact, from the method of construction. A metalinguistic variable presumably has a definition,

which gives rise to at least one arrow in the graph leading out of the corresponding node. Thus it follows immediately that this node cannot be terminal; it may or may not be initial. A metalinguistic constant, on the other hand, does not have a definition, and therefore the corresponding node has no arrows leading out of it; that is, it is a terminal node. These considerations allow us to label each terminal node with a constant and each non-terminal node with a variable, as follows:



This is called the derivation tree of the string $621_{10}5$ as an unsigned number. It is the final form of diagram which we will associate with a character string in this way. In order to explain the term "derivation tree," it is necessary first to explain two other notions. A path in any graph is a sequence of nodes with a connected sequence of links between them, such as the nodes 1, 2, 4, 7, 11, and 15 in our example. A cycle in a graph is a path which leads back to where it started. Our example of a graph does not have any cycles, and, for this reason, it is called a tree. All derivation trees are trees in this sense; the derivation of $621_{10}5$, for example, refers to the proof, as we have given it, that $621_{10}5$ is an unsigned number.

The correspondence between metalinguistic constants and variables, on the one hand, and terminal and non-terminal nodes on deri-

vation trees, on the other, was noticed very early in the mathematical study of syntax. Indeed, the term "metalinguistic variable" has almost completely disappeared. Instead of metalinguistic constants, we speak of terminal symbols (usually, simply terminals); instead of metalinguistic variables, we speak of nonterminals. This standard terminology will continue to be used throughout this book.

1-3 Semantics of Constants

There are various ways of determining the value of a constant. For example, consider the unsigned integer 621, and let us suppose that we are regarding this as an octal integer. How do we find its value? One way is to find the value (as an octal integer) of 62, multiply by 8, and add 1. The point of this method is that 621 is an unsigned integer precisely because it consists of 62 (unsigned integer) followed by 1 (digit). Thus, if we use this rule, there is an immediate relation between the syntax of unsigned integers and the calculation of their values.

In formal terms, let the unsigned integer U followed by the digit D be defined to be an unsigned integer, called I. Furthermore, let v_U, v_D, and v_I, denote the value of U, of D, and of I, respectively. We wish to derive an equation for v_I in terms of v_U and v_D. In this case it is

$$v_I = 8 \cdot v_U + v_D$$

If we were considering 621 as decimal instead of octal, the rule would be

$$v_I = 10 \cdot v_U + v_D$$

Such equations will be called semantic rules. Syntax is concerned with form; semantics is concerned with meaning. It might seem that the meaning of a constant resides entirely in its value, but this is not the case, as we may see by considering decimal fractions. Let the decimal fraction F consist of the period (.) followed by the integer I. What is the value of F in terms of the value of I? The answer is that we obtain the value of F by dividing the value of I by 10^N, where N is the number of digits in the integer I, or

the length of I. Unless the length, as well as the value, of I is part of its meaning, we cannot derive the value of a decimal fraction. We may write

$$v_F = v_I/10^{n_I}$$

where n_I is the length of I; but in that case we must give a separate semantic rule for n_I. Going back to our previous conventions, if n_U is the length of U and n_I is the length of I, we obviously have

$$n_I = n_U + 1$$

An unsigned integer consisting of a single digit may then be declared to have length 1.

Is the value of a real number enough to specify its meaning? In this case, as it turns out, there is no need to know the length, as a character string. We may, however, need to know the type, that is, whether the real number is in fact an integer. The type of a decimal number, according to the syntactic rules we have given, may be determined from the following table:

Kind of decimal number	Type
Unsigned integer only	Integer
Decimal fraction only	Real
Unsigned integer followed by decimal fraction	Real

Similarly, the type of an unsigned number is determined as follows:

Kind of unsigned number	Type
Decimal number only	Same as the type of the decimal number
Exponent part only	Real
Decimal number followed by exponent part	Real

Normally it is not necessary to know the type of a number for the purposes of the semantics of constants; but the type may be needed for other semantic purposes. For example, the value of the expression 2x3.1415926535 is found by performing real multiplication on the constant 3.1415926535 and the result of converting the integer 2 to real form. (Expressions will be taken up more generally in Chapter 2.) If by real multiplication we mean floating point multiplication, this will not always give the same result as does integer multiplication, even for integral arguments.

Length, value, and type are examples of attributes. We shall take the point of view that any character string in a language which has a meaning at all has that meaning expressed in terms of one or more attributes. The process by which the attributes of a string are constructed in terms of the attributes of its substrings, as we have been doing, is called synthesis, and our attributes are said to be synthesized. (There is another sort of attribute, associated with a different kind of semantic rule, which we shall introduce in section 2.4.)

We shall now give notational conventions for semantic rules which give values of synthesized attributes, when these are associated with a syntactic definition of a language as given in BNF. For this purpose, we slightly alter the form of a BNF rule. Where there are no alternatives -- that is, where the vertical line (|) is not used -- we insert a lower-case letter after every nonterminal, to serve as a label. Thus the rule

<decimal fraction> ::= '.' <unsigned integer>

might become

<decimal fraction> f ::= '.' <unsigned integer> i

In words: The decimal fraction f consists of a period (.) followed by the integer i.

Where the vertical line is used n times, there are n+1 alternatives, and each of these causes a label to be given to the nonterminal at the left of the ::= sign. This label, followed by ::= and preceded by ; , takes the place of the vertical line for every alternative except the first. Thus the rule

<integer> ::= <unsigned integer> | '+' <unsigned integer> | '-' <unsigned integer>

might become

<integer> x ::= <unsigned integer> u; y ::= '+' <unsigned integer> v; z ::= '-' <unsigned integer> w

In words: The integer x consists of the unsigned integer u;

the integer y consists of a plus sign (+) followed by the unsigned integer v;

the integer z consists of a minus sign (-) followed by the unsigned integer w.

Semantic rules are now given which make reference to the labels just given. For the value of the decimal fraction f, we shall not write v_f, as before, but f^v. The purpose of this superscript convention is mainly to reduce the awkwardness of more complex expressions. Thus if a_x be replaced by x^a, and b_c by c^b, then a_{b_c}, for example, becomes $(c^b)^a$, which may as well be written c^{ba}. Similarly

$$a_{b_{c_d}} \qquad \text{and} \qquad v_{x_1+x_2}$$

become d^{cba} and $(x_1+x_2)^v$ respectively (assuming that x_1 and x_2 are unaffected by the convention). In general, only one level of superscription is necessary, thus reducing printing costs as well.

The semantic rule which we have given for decimal fractions would now be written

<value> $f^v = i^v/\exp(10, i^n)$

Notice that we use exp(x, y) instead of x^y in order to use superscripts for one purpose only. The term <value> is included here for explanatory purposes only. We shall write the semantic rule or rules directly under the corresponding syntactic rule, thus:

<decimal fraction> f ::= '.' <unsigned integer> i
<value> $f^v = i^v/\exp(10, i^n)$

Similarly, the rules for exponent parts now read

<exponent part> e ::= '$_{10}$' <integer> i
<value> $e^v = \exp(10, i^v)$

Where there is more than one alternative in a rule, we must have separate semantic rules for each; these are separated by semicolons. As an example, we give the rules for unsigned integers:

<unsigned integer> x ::= <digit> u; y ::= <unsigned integer> z <digit> v
<value> $x^v = u^v$; $y^v = 10 \cdot z^v + v^v$
<length> $x^n = 1$; $y^n = z^n + 1$

In words, an unsigned integer is either:

(1) a single digit, in which case its value is the value of that digit and its length is 1; or

(2) an unsigned integer z followed by a digit v, in which case its length is one more than the length of z, and its value is found

by taking the value of z, multiplying by 10, and adding the value of y.

The interaction between a rule like this, which defines "unsigned integer," and a rule such as the one for decimal fractions, in which unsigned integers are used, bears further study. The superscripted letters i^v and i^n in the rule for the value of a decimal fraction refer to the value and the length, respectively, of the integer i. Although i^v and i^n do not occur in the rules which define unsigned integers, we note that i, in the decimal fraction rule, clearly denotes an unsigned integer; this then tells us to look for the superscript v in the definition of unsigned integers if we want the proper interpretation of i^v, and similarly for i^n. In fact, we do find both v and n defined here as superscripts. Also note (compare, for example, the definitions of decimal fraction and exponent part) that there is nothing wrong with repeating labels (in this case, the label i) from one syntax rule to another; each label has validity only for the semantic rules associated with that particular syntax rule.

The remaining syntactic and semantic rules for constants may now be written as follows:

<digit> a ::= '0'; b ::= '1'; c ::= '2'; d ::= '3'; e ::= '4';
f ::= '5'; g ::= '6'; h ::= '7'; i ::= '8'; j ::= '9'

<value> $a^v = 0$; $b^v = 1$; $c^v = 2$; $d^v = 3$; $e^v = 4$; $f^v = 5$;
$g^v = 6$; $h^v = 7$; $i^v = 8$; $j^v = 9$

<integer> x ::= <unsigned integer> u; y ::= '+' <unsigned integer> v; z ::= '-' <unsigned integer> w

<value> $x^v = u^v$; $y^v = v^v$; $z^v = -w^v$

<decimal number> x ::= <unsigned integer> u; y ::= <decimal fraction> f; z ::= <unsigned integer> v <decimal fraction> g

<value> $x^V = u^V$; $y^V = f^V$; $z^V = v^V + g^V$

<type> x^t = integer; y^t = real; z^t = real

<unsigned number> x ::= <decimal number> u; y ::= <exponent part> p; z ::= <decimal number> v <exponent part> q

<value> $x^V = u^V$; $y^V = p^V$; $z^V = v^V \cdot q^V$

<type> $x^t = u^t$; y^t = real; z^t = real

<number> x ::= <unsigned number> u; y ::= '+' <unsigned number> v; z ::= '-' <unsigned number> w

<value> $x^V = u^V$; $y^V = v^V$; $z^V = -w^V$

<type> $x^t = u^t$; $y^t = v^t$; $z^t = w^t$

1-4 Consequences of the Definitions

What facts can we prove, having set up a syntactic and semantic definition such as the one above? We shall find that a number of more or less arbitrary choices have effectively been made.

A number may be signed. This innocent-looking statement is the cause of a great deal of trouble when we start analyzing expressions. Suppose, for example, that we allow the sum of any two numbers to be an expression. This means that expressions such as 5+7 and 18-32 are allowed, which is all right, but it also means that +5++7 and -18--32 are expressions, which we may not want. It is quite permissible, of course, to allow multiple minus signs in a language, and to interpret, for example, ---3 as -(-(-3)). In ALGOL, however, this is not allowed, and thus the sum of two numbers cannot, in general, be regarded as an expression. What ALGOL does is to ignore completely the syntactic definition of "number" which has just been made. Any two unsigned numbers may be added, subtracted, etc., to produce an expression, while constructions like -7-4 (which is allowed) are handled in other, special ways.

An exponent part may stand alone. In FORTRAN, we may write X = 1.0E8 to set X equal to one hundred million; but X = E8 would set X equal to the value of the variable E8. In ALGOL, however, the character $_{10}$ cannot be used in a variable name, and so it is perfectly permissible to write X:=$_{10}$8 , for example.

A decimal point must be followed by at least one digit in a decimal fraction. If the preceding fact seems too general, this one may seem too restrictive. If .125 is an acceptable substitute for 0.125 (which it is), why isn't 37. an acceptable substitute for 37.0? In fact, in many other languages, it is; but not in ALGOL. In order

to include such constructions, we might add a fourth alternative definition of decimal numbers, so that the definition reads

<decimal number> w ::= <unsigned integer> a; x ::= <unsigned integer> b '.'; y ::= <decimal fraction> f; z ::= <unsigned integer> c <decimal fraction> g

<value> $w^V = a^V$; $x^V = b^V$; $y^V = f^V$; $z^V = c^V + g^V$

<type> w^t = integer; x^t = real; y^t = real; z^t = real

Negative exponents are permitted. Compare carefully the definitions of decimal fractions and of exponent parts. In the first case, the period must be followed by an unsigned integer; we do not want 37.-5, for example, to be a legal decimal number. In the second case, however, any integer may be used, and thus $6.023_{10}+23$ and $6.63_{10}-34$, for example, are allowable unsigned numbers.

Leading zeros are permitted. If a programmer wishes to write JANUARY:=01; FEBRUARY:=02; and so on, this is permitted under the above rules. To prohibit unsigned integers from having leading zeros (not advocated by this author as good language design), we might proceed as follows (syntactic definitions only):

<non-zero digit> ::= '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'

<non-zero integer> ::= <non-zero digit> | <non-zero integer> '0' | <non-zero integer> <non-zero digit>

<unsigned integer> ::= <non-zero integer> | '0'

Logical constants have not been provided for. In some languages, such as APL and PL/I, there is no difference between a logical constant and any other kind; "true" is represented by 1 and "false" by 0. In ALGOL, however, the keywords true and false are used, and we employ here a definition which has no relation to that of an integer:

<logical value> t ::= true; f ::= false

<value> t^V = true; f^V = false

Octal integers, followed by the letter B, are not allowed, as they would be in many versions of FORTRAN. In order to allow these, we might introduce the following rules:

<octal digit> a ::= '0'; b ::= '1'; c ::= '2'; d ::= '3';
e ::= '4'; f ::= '5'; g ::= '6'; h ::= '7'

<value> $a^V = 0$; $b^V = 1$; $c^V = 2$; $d^V = 3$; $e^V = 4$; $f^V = 5$; $g^V = 6$; $h^V = 7$

<octal integer> x ::= <octal digit> d 'B'; y ::= <octal digit> e <octal integer> z

<length> $x^n = 1$; $y^n = z^n + 1$

<value> $x^V = d^V$; $y^V = z^V + e^V \cdot \exp(8, z^n)$

Decimal digits may then be defined in terms of octal digits:

<digit> a ::= <octal digit> d; b ::= '8'; c ::= '9'

<value> $a^V = d^V$; $b^V = 8$; $c^V = 9$

This example illustrates several further facts. The rule for octal integers is a recursive one; it defines the octal integer 257B, for example, as the octal digit 2 followed by the octal integer 57B. The value of 257B is found by adding the value of 57B to $2 \cdot 64$ (that is, $2 \cdot \exp(8, 2)$, since the length of 57B is 2), which is, of course, the value of 200B. Note that the letter B does not figure in the length of such an octal integer; also, of course, d^V, in the rule for the value of an octal integer, refers to any of the quantities a^V through h^V in the definition of an octal digit, and not specifically to d^V there.

No limits are placed on the sizes of numbers. In programming terminology we might say, "arbitrary precision is permitted," for both integers and real numbers. Of all the rules set forth above, this is the only one that is methodically broken in ALGOL implementations. Not only is it broken (which is clearly necessary in a

world containing only a finite total number of tape reels), but it is broken in different ways for different implementations, corresponding normally (but not always) to the size of a computer word. It is thus important to know how to specify, within the context of a language description, a particular way in which to break the rules.

Let us first consider integers. Suppose that α is the smallest possible negative number that will fit into a word on some given computer, while β is the largest positive number. (For a one's complement computer, normally $\alpha = -\beta$; for a two's complement computer, normally $\alpha = -(\beta+1)$.) If the value of the integer **i** is denoted by i^V, then we must have

$$\alpha \leq i^V \leq \beta$$

We shall take the simplest possible course and add this relationship directly to the description of our language, enclosing it in parentheses to indicate that it is a necessary condition. In the case of the definition of unsigned integers given above, we would write

<unsigned integer> x ::= <digit> u; y ::= <unsigned integer> z <digit> v

<value> $x^V = u^V$; $y^V = 10 \cdot z^V + v^V$

$(\alpha \leq y^V \leq \beta)$

<length> $x^n = 1$; $y^n = z^n + 1$

Note that only y^V, and not x^V, needs to be restricted in this way, because clearly $-9 \leq x^V \leq 9$, while α and β will normally have absolute value greater than 9.

For floating point numbers, the corresponding condition is harder to express. Of course, there is a largest possible floating point number on any given computer, and likewise a smallest possible positive floating point number (corresponding, on most computers, to

the bit pattern which is the same as that representing the integer 1). We could require that the absolute value of any floating point number lie between these two limits, in much the same way as we have done above. But this is not the only condition which a legal floating point number must satisfy, because of the question of finite accuracy. A legal floating point number must be representable as a positive or negative power of 2 (or of 16, on the IBM 360) times an integer between certain upper and lower limits. Whenever a constant which would otherwise be legal and of type real does not satisfy these requirements (such as 0.1, for example), its value must be defined in some way as a legal floating point number. Often there is more than one reasonable way of doing this; for example, a choice between truncating and rounding off always presents itself. It is these considerations which make the proof of assertions about programs involving floating point numbers much more difficult than corresponding assertions about integers; this subject will be taken up further in Chapter 5.

At this point we may ask ourselves: Why make any such restrictions at all? We are, after all, dealing with a mathematical abstraction, which may be carefully distinguished from the actual situation on a given computer in a way somewhat analogous to that in which the ALGOL reference language and the ALGOL hardware representations are distinguished in the ALGOL report. Eventually, of course, it will be necessary for us to prove assertions about programs which run on real computers, and for this we must have a way of describing the mathematical effects of floating-point operations and the actual correspondence between real constants and their floating-point equivalents. For the moment, however, let us concentrate on the ideal case in which real numbers and integers may be arbitrary.

1-5 Alternative Definitions

The importance of BNF as a way of describing languages arises from its generality; it is not restricted to languages with ALGOL-like rules. As an example of this, we give rules for the syntax of constants in a language resembling FORTRAN. We emphasize that FORTRAN is not as strictly defined as ALGOL, and consequently not all FORTRAN systems will handle constants in exactly this way.

<octal digit> a ::= '0'; b ::= '1'; c ::= '2'; d ::= '3';
e ::= '4'; f ::= '5'; g ::= '6'; h ::= '7'

<value> $a^V = 0$; $b^V = 1$; $c^V = 2$; $d^V = 3$; $e^V = 4$; $f^V = 5$; $g^V = 6$; $h^V = 7$

<digit> x ::= <octal digit> d; y ::= '8'; z ::= '9'

<value> $x^V = d^V$; $y^V = 8$; $z^V = 9$

<octal integer> x ::= <octal digit> d 'B'; y ::= <octal digit> e <octal integer> z

<length> $x^n = 1$; $y^n = z^n + 1$

<value> $x^V = d^V$; $y^V = z^V + e^V \cdot \exp(8, z^n)$

<decimal integer> x ::= <digit> d; y ::= <digit> e <decimal integer> z

<length> $x^n = 1$; $y^n = z^n + 1$

<value> $x^V = d^V$; $y^V = z^V + e^V \cdot \exp(10, z^n)$

<unsigned integer> x ::= <octal integer> o; y ::= <decimal integer> d

<value> $x^V = o^V$; $y^V = d^V$

<integer> x ::= <unsigned integer> u; y ::= '+' <unsigned integer> v; z ::= '-' <unsigned integer> w

<value> $x^V = u^V$; $y^V = v^V$; $z^V = -w^V$

<real number without exponent> x ::= <decimal integer> a '.';
y ::= <decimal integer> b '.' <decimal integer> c;
z ::= '.' <decimal integer> d

<value> $x^V = a^V$; $y^V = b^V + c^V/\exp(10, c^n)$; $z^V = d^V/\exp(10, d^n)$

<unsigned real number> x ::= <real number without exponent> r;
y ::= <real number without exponent> s 'E' <integer> i

<value> $x^V = r^V$; $y^V = s^V \cdot \exp(10, i^V)$

<real number> x ::= <unsigned real number> u; y ::= '+' <unsigned real number> v; z ::= '-' <unsigned real number> w

<value> $x^V = u^V$; $y^V = v^V$; $z^V = -w^V$

<unsigned double precision number> x ::= <real number without exponent> r 'D' <integer> i

<value> $x^V = r^V \cdot \exp(10, i^V)$

<double precision number> x ::= <unsigned double precision number> u; y ::= '+' <unsigned double precision number> v; z ::= '-' <unsigned double precision number> w

<value> $x^V = u^V$; $y^V = v^V$; $z^V = -w^V$

<complex number> c ::= '(' <real number> r ',' <real number> s ')'

<value> $c^V = r^V + i \cdot s^V$

<logical constant> t ::= '.TRUE.'; f ::= '.FALSE.'

<value> t^V = true; f^V = false

Note that this is still very much an ideal description of FORTRAN. No limits are placed on the size of integers or the precision of real numbers; even more obviously, there is no difference between the value of a real number and the value of the corresponding double precision number. In order to describe such a difference on a particular computer, we might define two functions, fapprox and

dapprox, for that computer, such that fapprox(x) is that floating point number and dapprox(x) is that double-precision number which is closest in value to the value of the real number x. The semantic rule for the value of a real number is then replaced by

$$\langle\text{value}\rangle\ x^V = \text{fapprox}(u^V);\ y^V = \text{fapprox}(v^V);\ z^V = \text{fapprox}(-w^V)$$

and the rule for the value of a double precision number by

$$\langle\text{value}\rangle\ x^V = \text{dapprox}(u^V);\ y^V = \text{dapprox}(v^V);\ z^V = \text{dapprox}(-w^V)$$

Note that by doing this we give actual values (floating point and double precision numbers) to strings of type <real number> and <double precision number>, but ideal values (real numbers) to strings of type <unsigned real number>, <unsigned double precision number>, and <real number without exponent>. If, instead, we gave all quantities actual values, we would encounter the usual error propagation problems associated with multiplication and division in these rules. For the value of a string such as 3.168, for example, we want that floating point number whose value is as close as possible to 3.168, and this is not necessarily the floating point number obtained by dividing 168 by 1000, using floating point division, and adding the result to 3.0, using floating point addition.

The rules given above for octal integers are the same as those of the preceding section; the rule for decimal integers is given in analogy to the rule for octal integers. As we remarked at the end of section 1.1, the syntactic rules

<alpha> ::= <beta> | <alpha> <beta>

and

<alpha> ::= <beta> | <beta> <alpha>

are equivalent; each defines an "alpha" as an arbitrary sequence of "beta's." The decimal integer rule above is of the second kind, as contrasted with the earlier rule, which was of the first kind. The semantic rules are also equivalent, although they are quite dissimilar. Note also that the octal integer syntax rule is a special case of the use of

<alpha> ::= <beta> <gamma> | <beta> <alpha>

to define an "alpha" as an arbitrary sequence of "beta's," followed by a single "gamma." Similarly, the rule

<alpha> ::= <gamma> <beta> | <alpha> <beta>

defines an "alpha" as an arbitrary sequence of "beta's," preceded by a single "gamma"; while either of the two equivalent rules

<alpha> ::= <beta> | <beta> <gamma> <alpha>

or

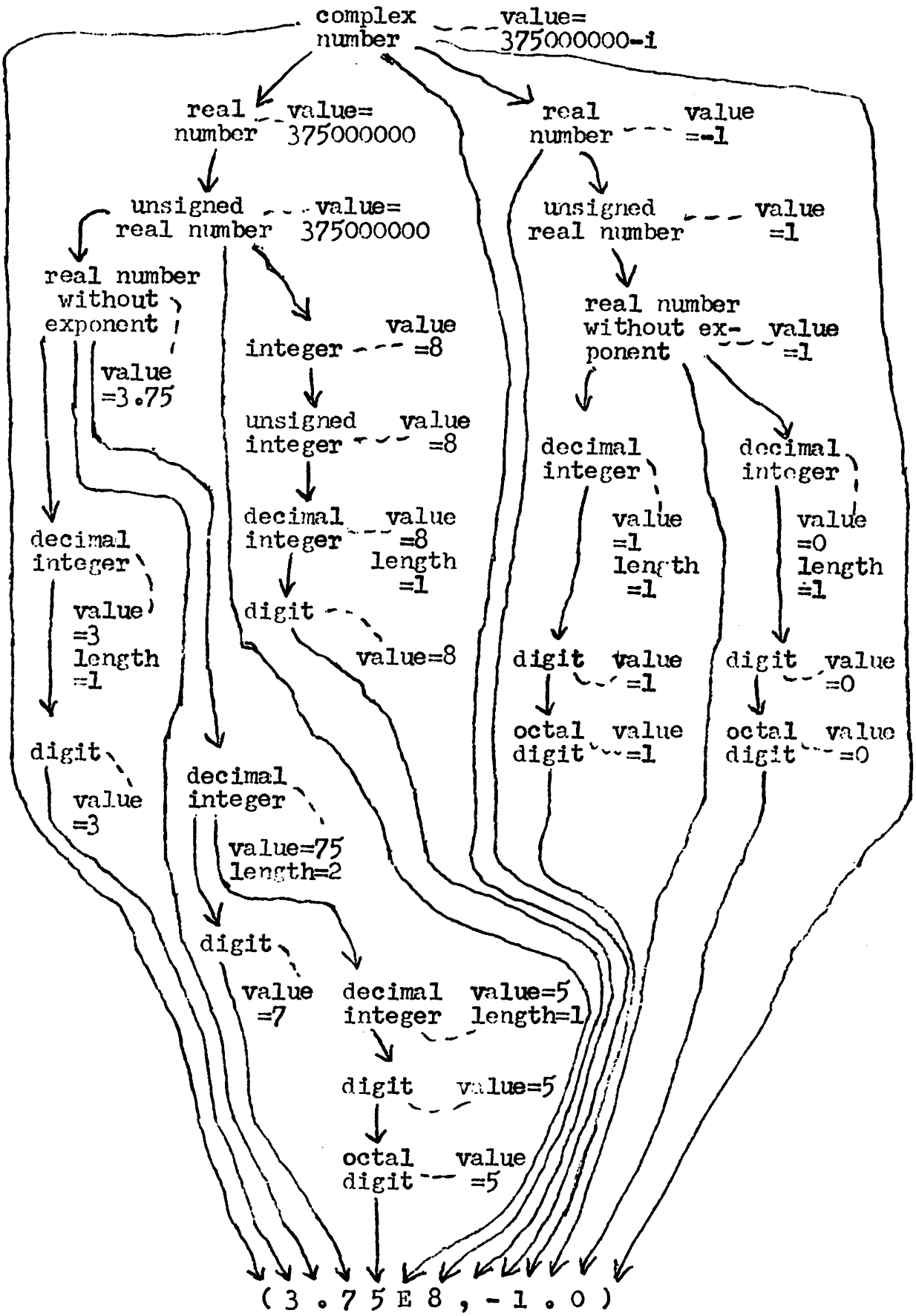
<alpha> ::= <beta> | <alpha> <gamma> <beta>

define an "alpha" as an arbitrary sequence of "beta's" separated by "gamma's," that is, in the order

beta gamma beta gamma ..... gamma beta

Several examples of these general rules are treated in the next four chapters.

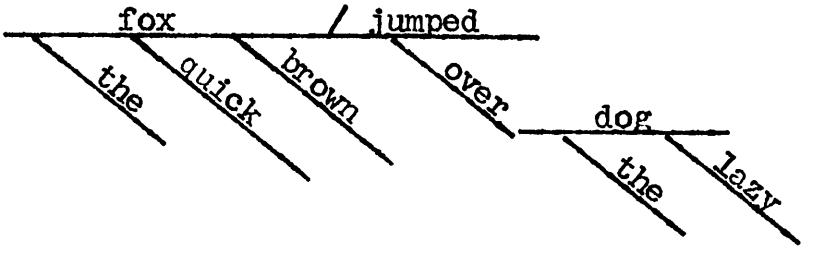
As an example of a derivation tree in FORTRAN, we give the derivation of (3.75E8,-1.0) as a complex number according to the FORTRAN rules given above. To each nonterminal in this tree, we have attached its semantic attributes:


complex number
value= 375000000-i
real number
value= 375000000
real number
value =-1
unsigned real number
value= 375000000
unsigned real number
value =1
real number without exponent
value =3.75
integer
value =8
real number without ex- ponent
value =1
unsigned integer
value =8
decimal integer
value =1 length =1
decimal integer
value =0 length =1
decimal integer
value =8 length =1
digit
value=8
digit
value =1
digit
value =0
octal digit
value =1
octal digit
value =0
decimal integer
value =3 length =1
digit
value =3
decimal integer
value=75 length=2
digit
value =7
decimal integer
value=5 length=1
digit
value=5
octal digit
value =5
(3 . 7 5 E 8 , - 1 . 0)

NOTES

The notation which came, later, to be called BNF was developed by John Backus [Backus 59] for use in the various conferences which were concerned with the development of ALGOL. BNF is actually an alternate form of a notation first used by Chomsky in describing the grammar of natural languages. The first version of ALGOL, later to be called ALGOL 58, was described in [Perlis and Samelson 59]; the version known as ALGOL 60 was published in three places in 1960 [Naur et al. 60] and in revised form three years later [Naur et al. 63]. It is the revised form to which reference will be made in this book.

The study of what we refer to as derivation trees is very old, and antedates the computer by many years. At this time, of course, derivation trees were studied only for natural languages. The practice (discontinued some years ago in most elementary schools) of teaching children how to "diagram" a sentence -- so that "The quick brown fox jumped over the lazy dog" is represented, for example, by



-- corresponds, in natural languages, to derivation trees for programming languages.

Our treatment of attributes, and the terms "attribute" and "synthesized," follow the treatment of [Knuth 68] and [Knuth 71]. (See also the treatment of inherited attributes in section 2.4.) With respect to the subject of programming language semantics, there are

currently many differing schools of thought, which are excellently summarized in [de Bakker 69]. Of special interest are the state vector concept [McCarthy 63], which we will be using in the sequel; the formal definitions of interpreters according to the so-called "Vienna method" [Lucas and Walk 69]; the formal definitions of compilers using Floyd-Evans production language [Feldman and Gries 68]; and the EULER language [Wirth and Weber 66] and the ISWIM meta-language [Landin 66].

The notation which we use for attributes has been described in [Maurer 72].

EXERCISES

1. The following syntax rules are given in English. Express them in BNF. (These rules deal with statements, to be examined further in Chapter 3. They do not correspond to the rules of ALGOL.)

a. An unlabelled statement is either an assignment statement, a conditional statement, or a transfer statement.

b. A statement is an unlabelled statement, optionally preceded by a label.

c. A declaration section is an arbitrary sequence of declarations.

d. An assignment statement consists of a variable followed by := followed by an expression.

e. A conditional statement consists of the word IF followed by a Boolean expression followed by THEN followed by a statement, optionally followed by ELSE and another statement.

2. A commonly encountered extension of BNF involves the use of curly brackets { } to denote optional quantities. Thus the rules

<sign> ::= '+' | '-'

<integer> ::= {<sign>} <unsigned integer>

are equivalent to the rule for integers given in section 1-1. Rewrite the definitions of number, unsigned number, decimal number, and integer, from section 1-1, and the definitions of statement and conditional statement in the problem above, using this notation. You should not need any alternative signs except in two of the above cases.

3. Construct derivation trees, as defined in section 1-2, for the following character strings:

a. 736 (unsigned integer)

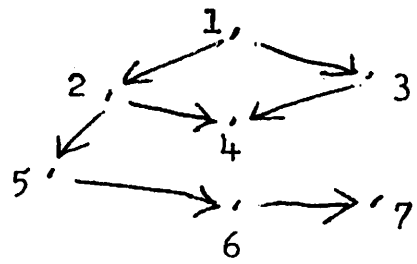
b. -5.7 (number)

c. 2_{10}-7 (unsigned number)

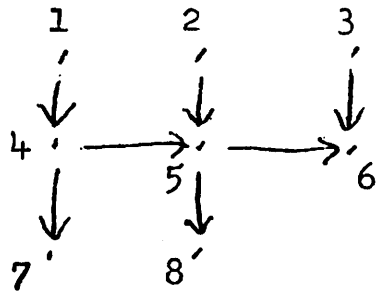
d. 1 (number)

4. Which nodes are initial and which are terminal in each of the following graphs?

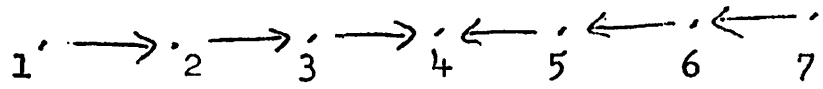
a.



b.



c.



5. Consider the syntactic rule

<unsigned integer> x ::= <digit> d; y ::= <unsigned integer> z <digit> e

Assume that digits have their usual values and that the superscript v is used for the value of a digit. What is the value of the unsigned integer 548217 under each of the following semantic rules for unsigned integers?

a. <value> $x^v = d^v$; $y^v = z^v + e^v$

b. <value> $x^v = d^v$; $y^v = z^v$

c. <value> $x^v = d^v$; $y^v = e^v$

d. <value> $x^v = d^v$; $y^v = z^v - e^v$

6. Complex constants in FORTRAN are denoted by pairs of real constants separated by a comma and enclosed in parentheses; thus (2.0, 3.0) is the constant normally denoted by 2+3i (or in electrical engineering, 2+3j). Assume that a definition of <real number> has already been given for FORTRAN real numbers, in which the superscript v denotes the value of a real number. Write a syntax rule for complex constants in this form, in terms of real numbers, and then give a semantic rule which gives the value, in the usual sense, of such a complex number, again using the superscript v.

7. (a) How would we modify the syntactic rules presented in section 1-1 in order to prohibit exponent parts from standing alone? That is, we wish $1_{10}5$ and $1.0_{10}5$ to be legal unsigned numbers, for example, but not $_{10}5$ by itself.

(b) Suppose that we wished to remove the restriction, in section 1-4, that a decimal point must be followed by at least one digit. Let us try to do this in a different way than is done in the text, by defining

<decimal fraction> x ::= '.'; y ::= '.' <unsigned integer> u

<value> $x^v = 0$; $y^v = u^v/\exp(10, u^n)$

and leaving all other definitions unchanged. Show that this change produces almost, but not quite exactly, the same effect as does the one given in the text.

(c) Give semantic rules to go with the syntactic definitions of non-zero digit, non-zero integer, and unsigned integer given in

section 1-4, which express the clearly intended meaning of these quantities in this context.

8. (a) Give a semantic rule, in terms of the others in this chapter, which defines the length of an integer to be the same as the length of the unsigned integer contained in it.

(b) Using this rule, express as a semantic condition (enclosed in parentheses as in section 1-4) the statement that the length of the integer in an exponent part must not exceed 2. (This is similar to a condition found in some versions of FORTRAN.)

(c) Give semantic rules, in terms of the others in this chapter, which define the length of <u>any</u> quantity to be the number of characters in it. (This rule conflicts with that of (a) above.)

(d) Using these rules, express as a semantic condition, as in (b) above, the statement that no number (as <number> is defined) can be too large to fit on a single 80-column card.

9. Give syntactic and semantic definitions, according to the style illustrated in the text, for:

(a) Binary integers (containing 1's and 0's, optionally preceded by -).

(b) Unsigned hexadecimal integers, using the hexadecimal digits 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, and F.

(c) Roman numerals (from 1 through 3,999).

10. Give derivation trees with all attributes attached, as at the end of section 1.5, for the following FORTRAN constructions as defined in that section:

(a) -377B (integer)

(b) -7.12E-4 (real number)

(c) 41.7D0 (double precision number)

(d) 183.4E10B (what is this?)

C H A P T E R T W O

E X P R E S S I O N S

2-1 Precedence

The class of arithmetic expressions such as A+B, TAU/BETA-GAMMAx2.0, and EXP(5.0-COS((G-H)xDELTA), 2.5)/(4.0+LN(SIGMA)) is the prototype for a rather loosely defined collection of classes of objects called expressions. An expression is made up of operators and operands, which have various properties depending on the kind of expression being studied. For arithmetic expressions, the operators are + - * / in FORTRAN, or + - x / ÷ in ALGOL (where real division / and integer division ÷ are distinguished from each other), or the like; parentheses are sometimes viewed as a special kind of operator. The operands are constants and variables; a function call (such as the use of EXP above) or an expression in parentheses (such as (4.0+LN(SIGMA))) may also be viewed as an operand.

Most operators are either binary or unary. A binary operator has two arguments; thus / (for division) is a binary operator, since one quantity must be divided by another one. A binary operator is normally written between its two operands; thus we write A/B or (W+X)/(Y-Z).* Unary operators have one argument; probably the most common use of a unary operator is the use of the minus sign to denote negatives. Thus -X is the negative of X. A unary operator is normally written before its argument. In theory, there is nothing to prevent expressions from being constructed with n-ary operators,

---

* Sometimes, in computing, this rule is not followed, and we write, for example, A B / instead of A/B. This is called Polish notation (see also section -).

for arbitrary n, and in fact the name of a function of n arguments is very often regarded as an n-ary operator.

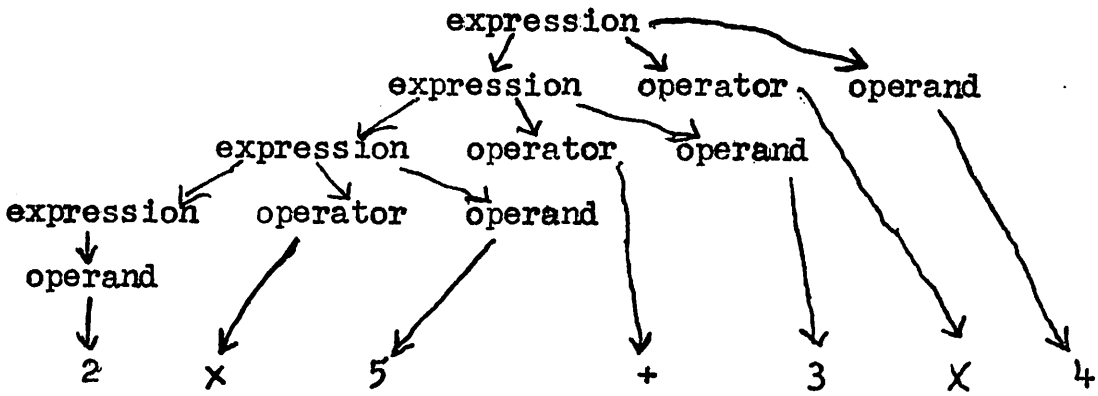
Let us now consider the problem of describing expressions in BNF. For simplicity, we use only the operators + and x, and we do not allow the unary + (that is, the convention according to which +α may be used instead of α, for any operand α). Under these conditions, it is clear that an expression is an arbitrary sequence of operands, separated by operators. This may be expressed in BNF as follows:

<expression> ::= <operand> | <expression> <operator> <operand>

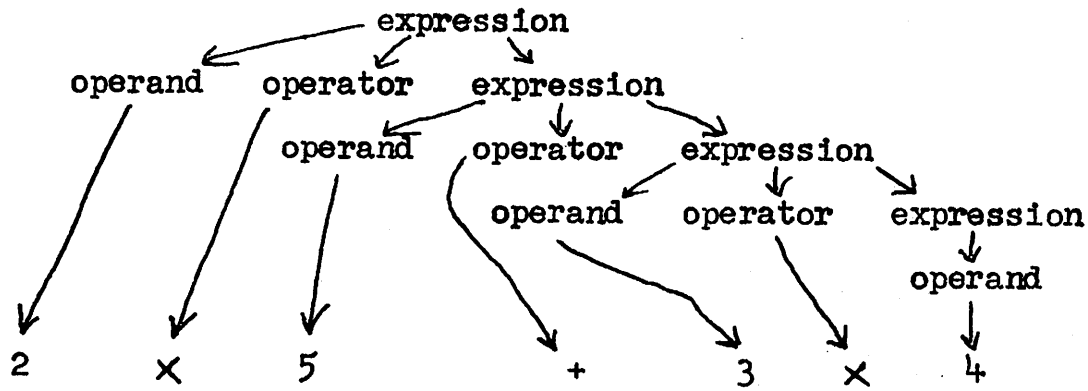
It might also be expressed as

<expression> ::= <operand> | <operand> <operator> <expression>

Using the first rule, the derivation tree of 2x5+3x4 would be



while using the second rule, it would be



The trouble with these BNF rules may be seen when we start considering the semantics. In the first case, the expression 2x5+3x4 is made up of the expression 2x5+3, the operator x, and the operand 4. The expression 2x5+3 has value 13, and if this is multiplied by 4, the answer is 52. In the second case, 2x5+3x4 is made up of the operand 2, the operator x, and the expression 5+3x4; this last expression has value 17, and if this is multiplied by 2, the answer is 34. This is actually the way in which values are calculated in the language APL, where this expression, in particular, would have the value 34; in most languages, however, the value of 2x5+3x4 is 10+12, or 22. Our problem is that we have not taken account, in our BNF rules, of the conventions according to which certain operations are performed before others. In this case, the normal convention directs multiplication to be performed before addition; so we would multiply 2 by 5, obtaining 10, then 3 by 4, obtaining 12, and finally add the results to get the value of 2x5+3x4. We cannot express this method of calculating values as a semantic rule which synthesizes the value of an expression from the values of the operand and subexpression which it contains according to either of the above syntactic rules.

We say that the operator x has a greater precedence than the operator +. Whenever there are precedence rules, one method of expressing them in BNF involves a different nonterminal for each level of precedence. Consider an expression as above, just after all multiplications have taken place; we may then say that the expression is made up of terms, each of which is the result of some multiplication. In the example above, the terms are 2x5 and 3x4. Each term, in turn, is made up of factors. What we do is to construct separate syntactic definitions of expression, term, and factor. An expression, in the sense above (that is, where the only operators are + and x, with the usual precedence) is a sequence of terms, separated by plus signs.

A term is a sequence of factors separated by multiplication signs. In our example, the factor is the primary expression type from which all expressions are built up; but if, for example, we add the exponentiation operator ↑, with the usual precedence rules -- namely, that all exponentiations are performed first, then all multiplications, and finally all additions, so that, for example, 2+5x4↑3x8+1 is interpreted as 2+(5x(4↑3)x8)+1 or 2563, with 4↑3 standing for 4^3 -- then a factor would be considered as a sequence of primary expressions separated by exponentiation signs.

All these rules may be expressed recursively. We may write

<expression> ::= <term> | <expression> '+' <term>
<term> ::= <factor> | <term> 'x' <factor>

for expressions involving only + and x; a factor is then the same as an operand. For expressions involving +, x, and ↑, we add the rule

<factor> ::= <primary> | <factor> '↑' <primary>

This last rule is taken directly from ALGOL; here a primary (short for "primary expression") is a variable, an unsigned number (recall the discussion of this in section 1.4), a function reference (that is, the name of a function, followed optionally by a list of arguments in parentheses), or an arbitrary arithmetic expression enclosed in parentheses.

Other binary operators are subject to similar rules. Subtraction is said to have the same precedence as addition, according to the normal rules of precedence. This means that as soon as all multiplications, divisions, etc., are performed, the additions and subtractions are performed in their natural order, from left to right. In particular, 15-4+3 is not 15-(4+3) or 8, but (15-4)+3 or 14. This affects the structure of our syntax rules if we wish later to add

semantic rules to them, as before. We may write

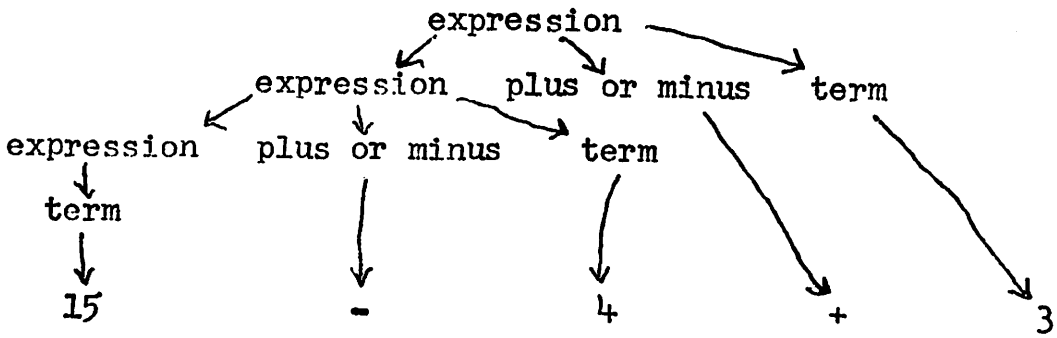
<plus or minus> ::= '+' | '-'

<expression> ::= <term> | <expression> <plus or minus> <term>

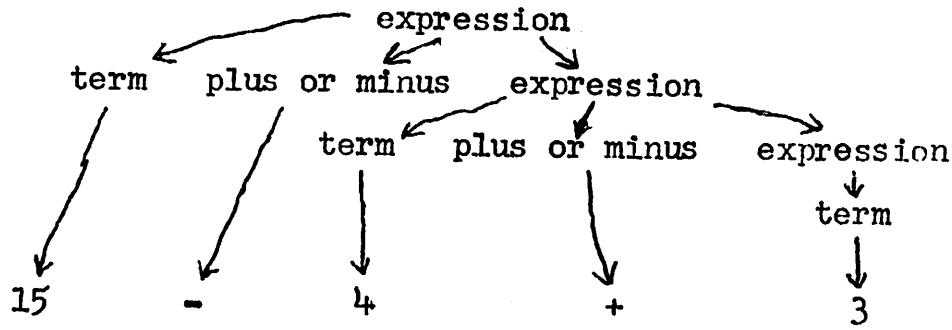
but not

<expression> ::= <term> | <term> <plus or minus> <expression>

as may be seen by constructing the corresponding derivation tree for 15-4+3; in the first case it is



while in the second case it is



Thus in the second case 15-4+3 consists of the term 15, the "plus or minus" - , and the expression 4+3; any semantic rule associated with this syntactic rule would interpret 15-4+3 as 15-(4+3). Notice that this problem does not arise with addition; it would cause no difficulty to replace our rule

<expression> ::= <term> | <expression> '+' <term>

for expressions with + and × only by

<expression> ::= <term> | <term> '+' <expression>

(since addition is associative).

In practice, ALGOL uses the slightly misleading term "adding operator" instead of "plus or minus." Similarly, "multiplying operator" covers multiplication and both kinds of division, and the ALGOL syntactic rules for terms are

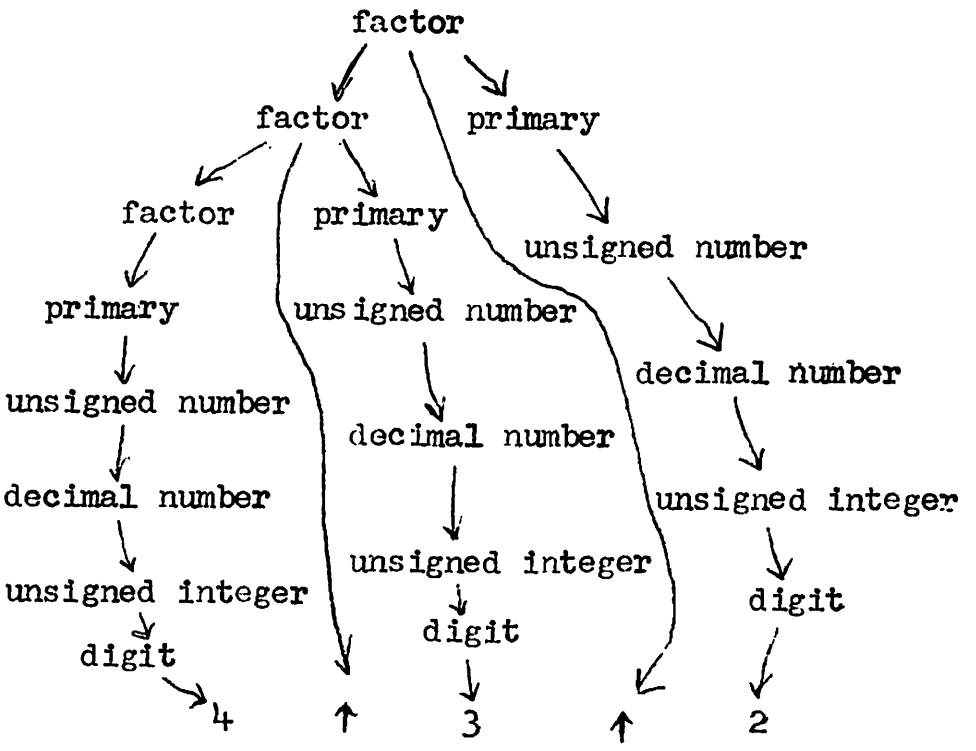
<multiplying operator> ::= '×' | '/' | '÷'

<term> ::= <factor> | <term> <multiplying operator> <factor>

The ALGOL rule for factors, involving exponentiation operators, namely

<factor> ::= <primary> | <factor> '↑' <primary>

has one slightly unnatural consequence. To see it, note that the derivation tree of 4↑3↑2 (for example), as a factor, is



In other words, 4↑3↑2 is the factor 4↑3, followed by ↑, followed by

the primary 2; it is interpreted as $(4\uparrow 3)\uparrow 2$, and semantically its value is thus 64^2 or 4096. Writing $4\uparrow 3\uparrow 2$ in ordinary exponential notation without parentheses, however, one obtains 4^{3^2}, which is normally interpreted as $4^{(3^2)} = 4^9 = 262144$. To reflect this result in a language such as ALGOL, we might write

<factor> ::= <primary> | <primary> '↑' <factor>

Further considerations of expression syntax, such as the handling of unary operators, conditional expressions, and logical or Boolean expressions, is postponed to section 2-5. Let us now turn to the calculation of values of expressions.

2-2 State Vectors

The value of an expression which is the sum of two terms is clearly, at least in simple cases, the sum of the values of the terms. We might, therefore, write

<expression> e ::= <term> v '+' <term> w

<value> $e^V = v^V + w^V$

as long as we could ignore all other kinds of expressions. But this semantic rule is lacking in one important sense: it does not express the fact that the value of an expression (as contrasted with the value of a number) is not a constant.

We can, of course, construct a class of expressions built up from constants only. All we have to do is to restrict our notion of primary expression in such a way that variables are not permitted. This allows us to find the (constant) values of expressions such as 23+45/(6+4-1)-8 by semantic rules resembling the one above. Even if variables are allowed, there are some situations in which a variable has one and only one value in a given program. For example, the lines

```
TABLE   DA    20F
LTABLE  EQU   80
```

in BAL (IBM 360 Basic Assembler Language) mean: Reserve 20 full words (80 bytes) of memory to hold a table called TABLE, and set the value of the constant LTABLE (presumably, the length of the table) equal to 80. Under these conditions, the BAL assembler (and, ultimately, the linkage editor as well) will determine some start location for TABLE. If this location is 1000 (decimal), then the expression TABLE+LTABLE-4 in the line

```
P8      LA    7,TABLE+LTABLE-4
```

has constant value 1000+80-4 or 1076.

In most computing, however, the variables in an expression change their values as the program proceeds. Every time an expression is evaluated, the current value of each variable must be taken. We may say that the value of an expression is a function of the current state of the computation. If the current state of the computation is denoted by S, then we may write

<expression> e ::= <term> v '+' <term> w

<value> $e^V(S) = v^V(S) + w^V(S)$

in place of the simplified rule above. That is: If S is the current state, then the value of the value-function e^V, when applied to S, is calculated by adding the values of the value-functions v^V and w^V, applied to that same S. This is true whatever S may be.

Let us, first of all, note that things are not always this simple. Suppose, for example, that the terms v and w have side effects. A side effect will be defined, for the moment, as anything which causes one or more variables to change their values. In the process, the current state of the computation is changed to a new state. Thus a side effect may also be regarded as a function of the current state of the computation, but its value is another state, rather than an integer, real number, etc. (Side effects may arise from references to functions, discussed in Chapter 4.) Suppose that the terms v and w, in addition to values v^V and w^V, have side effects v^S and w^S. That is, if S is the current state, then $v^S(S)$ is the new state after the term v is evaluated; in fact, if we first evaluate v and then w, our state changes from S to $v^S(S)$ and then to $w^S(v^S(S))$. It is reasonable to define this change as the side effect of the expression e. Thus we may write

<expression> e ::= <term> v '+' <term> w

<side effect> $e^S(S) = w^S(v^S(S))$

<value> $e^V(S) = v^V(S) + w^V(v^S(S))$

Note that the semantic rule for the value of an expression has also been slightly altered. After v is evaluated, the current state is not S but $v^S(S)$, and thus $w^V(v^S(S))$, not simply $w^V(S)$, is the proper value of w (if evaluation proceeds from left to right).

Now we take up the question: What is the nature of S? What kind of object is the current state of a computation? One way of looking at S is as a function. If v is any variable, then S(v) is the current value of v. Such a function is sometimes called a content function, and S(v) is the (current) contents of v. Another way of regarding S is as a vector, that is, mathematically, an n-tuple of the form

$$(k_1, k_2, \ldots, k_n)$$

Here there are presumed to be n variables manipulated by the program -- call them $x_1, \ldots, x_n$ -- and the current value of x_i is k_i, $1 \leq i \leq n$. (When n = 2 or 3, such an n-tuple resembles the co-ordinate specification of an ordinary vector in the plane or in 3-space.) We might also write

$$(x_1 = k_1, x_2 = k_2, \ldots, x_n = k_n)$$

as a representation for S. This representation has the advantage that if the variables of the program are not called $x_1, \ldots, x_n$, we can include their actual names in the vector specification. Thus

$$(I = 12, J7 = 9, X = 3.8, Y = 0.0)$$

might specify the current state of a FORTRAN computation. Such a

vector is called a state vector. In the future, we shall refer to the current state of any computation as a state vector; in representing it, we shall use the function form and the vector form interchangeably, since they are clearly equivalent.

This is perhaps a good time to review some elementary facts about sets and functions, since these will be needed in the precise specification of state vectors. A set is a collection of objects; each of these is called a member of the set, or an element of the set. If x is a member of the set X, we say that x is in X, and we write $x \in X$. A function from one set to another is a way of associating an element of the second set with an element of the first. If A and B are two sets and f is a function from A to B, then we write $f: A \rightarrow B$; if a is an element of A, then f(a) is the element of B which f associates with a. For the functions of high-school algebra such as $f(x) = x^2 + 5x - 24$, we may write $f: R \rightarrow R$, where R is the set of all real numbers. For the state vector

$$S = (I = 12,\ J7 = 9,\ X = 3.8,\ Y = 0.0)$$

given above, we may write $S: A \rightarrow B$ (where S now denotes the corresponding content function), where I, J7, X, and Y are members of A, and 12, 9, 3.8, and 0.0 are members of B.

If the number of elements in a set is finite, the set may be called a finite set. Any finite set may be specified by listing its elements, separated by commas, and enclosed in curly brackets $\{\ \}$. Thus we may write

$$A = \{I,\ J7,\ X,\ Y\}$$

if we wish to specify the set A in the previous example as being the set of exactly four elements I, J7, X, and Y. The set of all elements with which a function associates values is called the domain of the

function; thus A is here the domain of S.

If P and Q are two sets, and every element of P is also an element of Q, then we write $P \subseteq Q$, or P is contained in Q. If $P \subseteq Q$, we may also write $Q \supseteq P$, or Q contains P. If both $P \subseteq Q$ and $P \supseteq Q$, then P and Q are the same set, and we write $P = Q$. The notation $P \subset Q$ is also used to express the fact that P is contained in Q; however, we shall use $P \subset Q$ to exclude the case that $P = Q$. That is, if $P \subset Q$, then $P \subseteq Q$ and $P \neq Q$. If f is a function from X to P, then f can also be regarded as a function from X to Q, if $P \subseteq Q$. For example, P might be the set

$$P = \{12, 9, 3.8, 0.0\}$$

and Q might be the set of all real numbers (including the integers). The state vector S cited above is a function from A to P, but we normally regard it as a function from A to Q -- that is, as a function which assigns some integer or real number to I, to J7, to X, and to Y. The set of all values of a function is called the range of the function; in this case P is the range of S. For any function $f: A \rightarrow B$, if the range of f is B, we say that f is onto B; if the range of f is merely contained in B, then f is into B. A set which is contained in B is called a subset of B.

If $X_1, X_2, \ldots, X_n$ are any n sets, their cartesian product $X_1 \times X_2 \times \ldots \times X_n$ is the set of all ordered n-tuples of the form $(x_1, x_2, \ldots, x_n)$, where $x_1 \in X_1$, $x_2 \in X_2$, $\ldots$, $x_n \in X_n$. If we write V_x for the set of all legal values of the variable x, then the cartesian product of all the V_x, for all variables x that we wish to consider, is effectively the set of all legal state vectors involving those variables.

2-3 Type Conversion

Let us now consider how to modify semantic rules for expressions in the presence of mixed-mode arithmetic. If A and B are real numbers, I and J are integers, and S is the current state vector, then the current value of the expression A*B+I*J is not normally obtained simply by adding the current value to A*B to the current value of I*J. On most computers, we must perform type conversion on I*J first, converting it from integer form to real form before multiplying.

Types and type conversion are handled in fundamentally different ways in different programming languages. The method used by ALGOL involves the introduction of a new technique of specifying semantic attributes, and it is therefore postponed until the next section. We shall now study several other ways of handling types.

Let us first consider FORTRAN II. A variable name in FORTRAN II is of type integer if its starts with I, J, K, L, M, or N, and of type real otherwise. We may represent this by

<integer letter> ::= 'I' | 'J' | 'K' | 'L' | 'M' | 'N'

<real letter> ::= 'A' | 'B' | 'C' | 'D' | 'E' | 'F' | 'G' | 'H' | 'O' |
'P' | 'Q' | 'R' | 'S' | 'T' | 'U' | 'V' | 'W' | 'X' | 'Y' | 'Z'

<alphanumeric character> ::= <real letter> | <integer letter> | <digit

<identifier> x ::= <real letter>; y ::= <integer letter>;
z ::= <identifier> w <alphanumeric character>

<type> x^t = real; y^t = integer; z^t = w^t

<length> x^n = 1; y^n = 1; z^n = w^n + 1

Note that the last of these four syntax rules is the only one which has semantic rules attached. A syntax rule with no associated semantic rules will be written, as above, in BNF without the modifications involving labeling and replacing vertical lines by semicolons. Even

if there are associated semantic rules, we use labeling only where necessary; thus <real letter>, <integer letter>, and <alphanumeric character> are unlabeled where they occur in the definition of <identifier>. The rule for length is given in order that the length of an identifier may be restricted by a semantic condition; thus writing

$$(w^n < 6)$$

under the rule for <length> would force all identifiers to have length less than or equal to 6.

By combining these rules with the FORTRAN rules of section 1-5, we may introduce <u>type</u> as a semantic attribute of expressions, terms, factors, and primary expressions. At one end, we have a rule such as

<primary> <u>x</u> ::= <unsigned integer> <u>i</u>; <u>y</u> ::= <unsigned real number> <u>r</u>; <u>z</u> ::= <identifier> <u>v</u>

<type> x^t = <u>integer</u>; y^t = <u>real</u>; $z^t = v^t$

<value> $x^V = i^V$; $y^V = r^V$; $z^V(S) = S(v)$

(We might also include other kinds of primary expressions, such as subscripted variables, function references, and general expressions in parentheses; see also section 2-5.) At the other end, our rule for expressions (for the moment without side effects, and composed of two terms only, separated by a plus sign) might become

<expression> <u>e</u> ::= <term> <u>v</u> '+' <term> <u>w</u>

<type> $e^t = v^t$

$(v^t = w^t)$

<value> $e^V(S) = v^V(S) + w^V(S)$

This rule <u>forbids</u> mixed mode arithmetic, as is done in FORTRAN II. It is not quite as rigorous as it might be, because the plus sign is

taken to stand for both real and integer addition. Let radd and iadd stand for real addition and integer addition, respectively, as functions; that is, radd(x, y) is the sum, in the usual sense, of the real numbers x and y, while iadd(i, j) is the sum, in the usual sense, of the integers i and j. (For an ideal language description, radd and iadd would be operations on all real numbers and all integers respectively; for an actual language description, radd would be floating point addition, with all of its attendant rounding, overflow, and underflow characteristics, while iadd would be one-word integer addition.) If we know, in advance, that the type of a term must be either real or integer, we may rewrite the above semantic rule for the value of an expression as

$$\langle \text{value} \rangle\ e^v(S) = f(v^v(S),\ w^v(S)),\ \text{where } f =$$
$$\underline{\text{if}}\ v^t(S) = \underline{\text{real}}\ \underline{\text{then}}\ \text{radd}\ \underline{\text{else}}\ \text{iadd}$$

(This is our first use of a conditional expression. Even to those unfamiliar with ALGOL, the meaning of f = if $v^t(S)$ = real then radd else iadd should be obvious; it is the same as that of

$$f = \begin{cases} \text{radd} & \text{if } v^t(S) = \underline{\text{real}} \\ \text{iadd} & \text{otherwise} \end{cases}$$

but a bit easier to analyze by computer in various ways.) If we wish to permit mixed mode arithmetic between real and integer quantities, with type conversion in the usual way, we may introduce the function comb(a, t, b, u, f, g), which combines the quantity a, of type t, with the quantity b, of type u (where t and u must be either integer or real) according to the real operator f or the integer operator g, as the case may be. The semantic rule above would then be replaced by

$$\langle \text{value} \rangle\ e^v(S) = \text{comb}(v^v(S),\ v^t,\ w^v(S),\ w^t,\ \text{radd},\ \text{iadd})$$

where the formal definition of the comb function is

comb(a, t, b, u, f, g) = if t = real then f(a, if u = real then b else float(b)) else if u = real then f(float(a), b) else g(a, b)

Here float(a) is the real number corresponding to the integer a.

Such rules for permitting or forbidding mixed mode arithmetic may be further refined by specifying, in a precise manner, the domain of the state vectors S upon which various quantities depend. The simplest approach is to specify that only those variables whose values might affect an expression appear in the domain to be determined. In fact, this domain may itself be made an attribute of an expression. If A/B is a term, with associated domain $\{A, B\}$ (that is, the set consisting of the two elements A and B), and B/C is a term with associated domain $\{B, C\}$, then the domain to be associated with the sum of these two terms, A/B+B/C, is clearly $\{A, B, C\}$ -- that is, the set of all elements which are in either (or both) of the above sets, or the union of these sets. If P and Q are any two sets, the union of P and Q is denoted by $P \cup Q$. Thus in this case

$$\langle \text{domain} \rangle\ e^d = v^d \cup w^d$$

is a reasonable semantic rule for domains, to be associated with the syntactic rule for expressions above.

Now suppose S is a state vector whose domain is e^d. We must be able to "reduce" S to state vectors whose domains are v^d and w^d respectively. If the expression e is A/B+B/C, for example, then the value-function e^V is applicable to state vectors with A-, B-, and C-components, but the value-function v^V associated with the term v = (A/B) is applicable to state vectors having A- and B-components only. Therefore, the expressions $e^V(S)$ and $v^V(S)$ in the semantic rule for values are incompatible with each other. If S is the state vector

$\{A=a, B=b, C=c\}$, and S is the argument of e^V, then the argument we want for v^V is clearly the state vector $\{A=a, B=b\}$. In general, viewing S as a content function with domain D, we want the function S' with domain D', where $D' \subseteq D$ (in this case $D' = \{A, B\}$ and $D = \{A, B, C\}$) which satisfies $S'(x) = S(x)$ for all $x \in D'$. This is known as the restriction of S to D', and written $S|D'$. Thus the semantic rule for values which forbids mixed mode arithmetic might be rewritten as

$$\langle value \rangle\ e^V(S) = f(v^V(S|v^d), w^V(S|w^d)), \text{ where } f =$$
$$\underline{\text{if}}\ v^t(S) = \underline{\text{real}}\ \underline{\text{then}}\ \text{radd}\ \underline{\text{else}}\ \text{iadd}$$

whereas the one which permits mixed mode might be written as

$$\langle value \rangle\ e^V(S) = comb(v^V(S|v^d), v^t, w^V(S|w^d), w^t, radd, iadd)$$

In either case, the type of an expression is also an attribute of it. If we define the FORTRAN converted-type function, ctype, as

$$ctype(t, u) = \underline{\text{if}}\ t = \underline{\text{integer}}\ \underline{\text{and}}\ u = \underline{\text{integer}}\ \underline{\text{then}}\ \underline{\text{integer}}\ \underline{\text{else}}\ \underline{\text{real}}$$

then

$$\langle type \rangle\ e^t = ctype(v^t, w^t)$$

is the indicated semantic rule relating types of expressions and types of terms, with superscript t in both cases. In general, ctype(t, u) is the type resulting from conversion, where t and u are the types being converted.

Let us now pass from FORTRAN to APL, in which a single variable may have different types in the same job. We may, for example, have an assignment which sets the value of the variable A to be an array of 5 numbers, and later in the same job another assignment which sets A to be a 4-by-4 double array, or a 3-by-5-by 2 triple array, or even a character string. Under such conditions the type of an expression,

as well as its value, is a function of the current state of the computation. The APL expression A+B is valid when A and B are both arrays (of the same dimensions), in which case A and B are added componentwise; but it is also valid when A is an array (of any dimensions) and B is simply a number which is to be added to all components of A. Thus we may define a converted-type function, ctype, for APL, with the sample values

ctype(array(5,3), array(5,3)) = array(5,3)
ctype(scalar, array(5,3)) = array(5,3)
ctype(scalar, scalar) = scalar

Using this function, a syntactic rule for expressions and a semantic rule for their types, modeled after the treatment of expressions in APL, is

<expression> a ::= <term> t; b ::= <term> u <operator> o
<expression> c
<type> $a^t(S) = t^t(S)$; $b^t(S) = \text{ctype}(u^t(S), c^t(S))$

Here the syntactic rule has taken account of the right-to-left scan in the APL language (thus A×B+C means A×(B+C), for example).

In APL, the current type of any variable may be determined from its current value. This need not always be the case; for example, a variable may be taken, in some language, to have values which are bit patterns in a computer word, and normally it is not possible to tell whether such a bit pattern represents an integer or a floating-point number. In such a case, the model may be altered in either of two ways. We may associate with each variable v, such that our state vectors all have v-components, another variable t_v, the type of v, and give t_v-components to all state vectors as well. This reflects the fact that the type of a variable is in this case just as much a

variable quantity as the variable itself. The other method is to regard the values of a variable as pairs of the form (type, value), which are added, subtracted, etc., to form other such pairs; for example, in ALGOL, (integer, 5) + (real, 7) = (real, 12).

Finally, let us consider type conversion in SNOBOL.* Here the value of a variable is always a character string. If this string is numeric -- that is, if it consists solely of digits, possibly preceded by a minus sign -- then the arithmetic operations +, -, *, and / are applicable to it, and, in addition, it is automatically converted to one-word integer (rather than character string) form on some computers, provided that it fits into a single word. This, however, is an internal, implementation-dependent conversion, and has nothing to do with the formal definition of SNOBOL. In fact, SNOBOL may be formally defined without any type conversion at all. SNOBOL has, however, another related characteristic which affects the formal definition: <u>any</u> variable name may at any time be constructed by a SNOBOL program and used; that is, the total collection of variable names used by a given SNOBOL program cannot be determined by simply inspecting the statements of the program. Each such variable is assumed to have the null string as its initial value, and there are no restrictions on character strings which may be used as variable names (although strings which do not satisfy the rules for identifiers must be referenced indirectly). Consequently, a state vector in SNOBOL must have a component for <u>every</u> possible variable name. Most of these components are null, of course; in fact, in any given state vector, only a finite number of variables will have string values of non-zero length.

---

* The term "SNOBOL," throughout this book, will refer to SNOBOL 4.

2-4 Inherited Attributes

The FORTRAN II, APL, and SNOBOL languages studied in the previous section all have a common feature: they lack type declarations such as REAL I, J (FORTRAN IV) or real I, J (ALGOL) or DECLARE I, J FLOAT DECIMAL (PL/I). Such declarations require a fundamentally new method of obtaining the values of semantic attributes.

The process of synthesizing attributes which we introduced in section 1-3 has the property that, if it is used exclusively, the attributes of any string must all be derivable from the attributes of its substrings. Let us consider, for example, the string 12*34+56. This is an expression with two terms, 12*34 and 56. The attributes of 12*34+56 which we have studied are synthesized from attributes of its terms. The term 12*34, in turn, has attributes synthesized from those of its factors, in this case 12 and 34. Going even one level further, the length and value of 12, 34, and 56 are derived from those of 1, 2, 3, 4, 5, and 6, all of which are substrings of 12*34+56. In most languages, this causes no difficulty, because the value of 12*34+56 is intrinsic -- nothing in the given program other than this string can alter its value. There are, however, a few exceptions. In the assembly language of the PDP-8 computer, for example, there is a pseudo-operation which changes the normal number base from octal to decimal. If the pseudo-operation does not appear in a program, then 12*34+56, anywhere in that program, has the octal value 506, or the decimal value 326; if the pseudo-operation does appear (once, at the beginning of the program), then 12*34+56 has the decimal value 464. Thus the value of 12*34+56 cannot be determined from attributes of its substrings alone; that is, it cannot be an attribute of 12*34+56 under a system in which all attributes are synthesized.

The above is, admittedly, an unusual example; synthesis of attributes is usually enough, where constants are concerned. But the situation with respect to expressions containing declared variables is quite similar to this, and the difficulty it causes is even more fundamental: we cannot tell whether a string is a properly formed expression by looking at its substrings alone. If the expression

A*B+C

appears in a FORTRAN program containing the declaration

INTEGER A, B, C

then it is properly formed, whereas if the declaration were

LOGICAL A, B, C

then A*B+C would not be properly formed. If the declaration were

INTEGER A, C

and B were left as a real number, then A and C would both have to be converted from integer to real form before being multiplied and added, respectively. Thus we must find some new method of working with attributes which allows declarations and other "remote" constructions to affect them.

The simplest method, and the one which appears the most practical, is the reverse of the synthesis operation. Let us consider a typical syntax rule:

<alpha> ::= <beta> | <gamma> <delta>

A synthesized attribute of "alpha" would be calculated from attributes of "beta," "gamma," and "delta." Now suppose that we have an attri-

bute of "beta," of "gamma," or of "delta" which is calculated from one or more of the attributes of "alpha." Such an attribute is said to be inherited. If "beta" is defined by a syntax rule such as

<beta> ::= <epsilon> | <zeta>

then such an attribute may be further inherited; that is, an attribute of "epsilon" or of "zeta" may be determined from those of "beta." Continuing the process in this way, it is clear that attributes of any nonterminal occurring on a derivation tree may, in particular, be inherited from the nonterminal at the root of the tree. (The term "inherited" arises from considering derivation trees as if they were family trees, with the root of the tree as the father of the family and the other nonterminals as the sons, grandsons, etc.). Or, to put it another way, any substring y of a string x may inherit its attributes from those of x, if it appears in the derivation tree of x.

Let us first indicate how the use of inherited attributes solves the problem with declarations discussed above. All constants, expressions, and statements of a program P are substrings of P, and appear in the derivation tree of P. One of the attributes of P is normally a type function, or some generalization thereof. If the variables of P are I, J, K, A, and B, for example, where I, J, and K are integer variables and A and B are real variables, and t is the type function, then

t(I) = integer t(K) = integer t(B) = real
t(J) = integer t(A) = real

The type function is a synthesized attribute of P; it is determined from all the type declarations that occur in P, and, in the case of FORTRAN or PL/I, also from the names of undeclared variables which are used in P and thereby receive default declarations -- by the "I, J, K,

L, M, or N rule", for example. (How this is done is studied in greater detail in Chapter 5.) This type function is then inherited by each variable in the program. Strictly speaking, a variable inherits the type function from the expression in which it is contained; this expression in turn inherits the type function from the statement in which it is contained, and so on up to the program level. In a block structure language, each block has its own type function; for the moment, we shall regard this as an irrelevant complication, and regard the terms "program" and "block" as synonymous.

The type of a variable is now determined from the type function which is its inherited attribute. Let us modify the simplified definition of a primary from the preceding section:

<primary> x ::= <unsigned integer> i; y ::= <unsigned real number> r; z ::= <identifier> y

<type> x^t = integer; y^t = real; $z^t = z^y(v)$

<value> $x^v = i^v$; $y^v = r^v$; $z^v(S) = S(v)$

The only change is in the definition of the type of an identifier which is a primary. We have assumed that the type function associated with the primary z is denoted by z^y; the value of this function, when applied to the identifier y itself as argument, is the type of that identifier.

We must now give semantic conventions for denoting inherited attributes such as z^y in the above rule. We shall denote inherited attributes by superscripted letters, just as with synthesized attributes, and each inherited attribute of a nonterminal has a name which is enclosed in angle brackets, followed by the corresponding superscript. For the rule above, we would write

<primary> x ::= <unsigned integer> i; y ::= <unsigned real number> r; z ::= <identifier> v

<type function>^{y}

<type> x^{t} = integer; y^{t} = real; $z^{t} = z^{y}(v)$

<value> $x^{v} = i^{v}$; $y^{v} = r^{v}$; $z^{v}(S) = S(v)$

The name <type function>, and the superscript y, constitute the only information about type functions of primaries which is needed in the definition of a primary. The calculation of this type function will be carried out in a rule in which <primary> is used. For example, we might write

<factor> x ::= <primary> p; y ::= <factor> z '↑' <primary> q

<type function>^{y}

$p^{y} = x^{y}$; $z^{y} = y^{y}$; $q^{y} = y^{y}$

We have omitted all other semantic attributes of factors to concentrate on what happens to type functions. A factor, as well as a primary, has a type function, and the definition of this inherited attribute of factors is of the same form as the corresponding definition for primaries. The inherited attribute equation $p^{y} = x^{y}$ defines the type function of any primary which is a factor; the inherited attribute equation $q^{y} = y^{y}$ defines the type function of any primary which appears in a factor on the right side of the exponentiation symbol. In each case, the type function is the same as the one associated with the nonterminal next higher in the derivation tree. The same is true for the recursive equation $z^{y} = y^{y}$, which defines the type function of a factor contained in another factor.

2-5 General Expressions

Let us now take up where we left off at the end of section 2-3. The syntax rule for a primary in ALGOL is

<primary> ::= <unsigned number> | <variable> | <function designator> | '(' <arithmetic expression> ')'

We recall that unsigned numbers and variables are analogous, in that each can be prefixed with a unary operator (-3 and -A, for example). Function designators are what in FORTRAN would be called function calls -- a function name together with its arguments, if any, enclosed in parentheses, such as sqrt(a×a+b×b). The most important device in this rule, however, is the treatment of any arithmetic expression, enclosed in parentheses, as a primary. This rule alone embodies all of the usual characteristics of parentheses as used for grouping. Consider, for example, the expression 3×(8-(2-1))+5. The value of 8-(2-1) is 7, and this is calculated by first evaluating 2-1 and then treating 8-(2-1) as if the value of 2-1 were substituted for it. Thus (2-1) is treated as a primary expression, just as if it were the single character 1. Similarly, (8-(2-1)) is treated as if it were the single character 7 in finding the value of 3×(8-(2-1))+5, that is, 3×7+5 or 26.

Syntax rules for the unary operators + and - normally resemble the rules for integers and numbers given in section 1-1. Only a slight alteration in these rules is needed if we wish to allow unary operators to appear in sequence (such as ++5 or +-+18). For integers, for example, instead of writing

<integer> ::= <unsigned integer> | '+' <unsigned integer> | '-' <unsigned integer>

we write

<integer> ::= <unsigned integer> | '+' <integer> | '-' <integer>

What precedence should we give unary operators, when compared to the precedence of binary operators? If the unary + and - are given the highest precedence, so that they are performed before all binary arithmetic operators, then a unary operator may immediately follow a binary operator, so that 5+-4 or 8--7 are legal and have values equal to 1 in each case. Neither this behavior nor the use of several unary operators in succession, as above, is allowed in ALGOL. On the other hand, if we were to give the unary + and - the lowest precedence, then an expression like -5+4 would mean -(5+4) -- since the + would be done first -- rather than (-5)+4, which would be the normal algebraic meaning. The solution adopted in ALGOL is to give these two unary operators the same precedence as normal addition and subtraction, so that the ALGOL rule for simple arithmetic expressions reads

<simple arithmetic expression> ::= <term> | <adding operator> <term> | <simple arithmetic expression> <adding operator> <term>

since both "adding operators," + and -, are both binary and unary. Simple arithmetic expressions may then be combined, using if, then, and else, to form (general) arithmetic expressions according to the rule

<arithmetic expression> ::= <simple arithmetic expression> | <if clause> <simple arithmetic expression> 'else' <arithmetic expression>

where an "if clause" is defined by

<if clause> ::= 'if' <Boolean expression> 'then'

All of these rules may now be augmented by semantic attributes. We shall not, for the moment, give all the semantic attributes required by ALGOL, but shall confine ourselves to values and side effects of the type studied above, ignoring type conversion problems. Under these conditions, primaries, factors, terms, and simple and general arithmetic expressions may be described syntactically and semantically by the following simplified rules:

<adding operator> p ::= '+'; m ::= '-'

<binary function> p^b = plus; m^b = difference

<unary function> p^u = ident; m^u = neg

<multiplying operator> t ::= '×'; r ::= '/'; i ::= '÷'

<binary function> t^b = times; r^b = rdiv; i^b = idiv

<primary> w ::= <unsigned number> u; x ::= <variable> v;
y ::= <function designator> f; z ::= '(' <arithmetic expression> e ')'

<value> $w^V(S) \equiv u^V$; $x^V(S) = S(v)$; $y^V(S) = f^V(S)$; $z^V(S) = e^V(S)$

<side effect> $w^S(S) = S$; $x^S(S) = S$; $y^S(S) = f^S(S)$; $z^S(S) = e^S(S)$

<factor> x ::= <primary> p; y ::= <factor> z '↑' <primary> q

<value> $x^V(S) = p^V(S)$; $y^V(S) = \exp(z^V(S), q^V(z^S(S)))$

<side effect> $x^S(S) = p^S(S)$; $y^S(S) = q^S(z^S(S))$

<term> x ::= <factor> f; y ::= <term> z <multiplying operator> o <factor> g

<value> $x^V(S) = f^V(S)$; $y^V(S) = o^b(z^V(S), g^V(z^S(S)))$

<side effect> $x^S(S) = f^S(S)$; $y^S(S) = g^S(z^S(S))$

<simple arithmetic expression> x ::= <term> t; y ::= <adding operator> a <term> u; z ::= <simple arithmetic expression> s <adding operator> b <term> v

<value> $x^V(S) = t^V(S)$; $y^V(S) = a^u(u^V(S))$; $z^V(S) = b^b(s^V(S), v^V(s^S(S)))$

<side effect> $x^S(S) = t^S(S)$; $y^S(S) = u^S(S)$; $z^S(S) = v^S(s^S(S))$

<if clause> i ::= 'if' <Boolean expression> b 'then'

<value> $i^V(S) = b^V(S)$

<side effect> $i^S(S) = b^S(S)$

<arithmetic expression> x ::= <simple arithmetic expression> s; y ::= <if clause> i <simple arithmetic expression> t 'else' <arithmetic expression> z

<value> $x^V(S) = s^V(S)$; $y^V(S) =$ if $i^V(S)$ then $t^V(i^S(S))$ else $z^V(i^S(S))$

<side effect> $x^S(S) = s^S(S)$; $y^S(S) =$ if $i^V(S)$ then $t^S(i^S(S))$ else $z^S(i^S(S))$

It is assumed, in the above rules, that plus, difference, times, rdiv, idiv, and exp are respectively the addition, subtraction, multiplication, real division (fractional result), integer division (integer result with remainder), and exponentiation functions of two real variables, and that ident and neg are respectively the identity and negation functions of one real variable (recalling that type conversion is here being ignored). The symbol ≡ denotes, as usual, that the value of a function (in this case w^V in the rule for the value of a primary) does not depend on its argument. The value $b^V(S)$ of the Boolean expression b is assumed to be either true or false, depending on S; also, b is assumed to have a side effect $b^S(S)$.

The calculation of the values and side effects of Boolean expressions in ALGOL is analogous to their calculation for arithmetic

expressions, as above. Each of the Boolean operators has a distinct precedence; the unary operator not ($\neg$) has the highest precedence, followed in order by and ($\wedge$), or ($\vee$), implies ($\supset$), and equivalence ($\equiv$). Recalling our discussion of the unary minus, we see that this convention allows for expressions such as $a \wedge \neg b$ and $a \vee \neg b$, where a and b are Boolean variables. This is in contrast to the situation with arithmetic expressions, where $a \times -b$, for example, is not allowed. (If $a \times -b$ means $a \times (-b)$, it is, of course, equivalent to $-a \times b$, which is legal, whereas $a \wedge \neg b$ and $\neg a \wedge b$ are clearly not equivalent.) Boolean expressions may be formed using if, then, and else, just like arithmetic expressions, and, in addition, two arithmetic expressions joined by a relation (such as $\leq$) form a Boolean expression. We give the ALGOL syntactic rules for Boolean expressions, together with simplified semantic rules resembling those above:

<relational operator> u ::= '<'; v ::= '≦'; w ::= '='; x ::= '≧'; y ::= '>'; z ::= '≠'

<binary operator> u^b = less; v^b = notgreater; w^b = equal; x^b = notless; y^b = greater; z^b = notequal

<relation> r ::= <simple arithmetic expression> x <relational operator> o <simple arithmetic expression> y

<value> $r^V(S) = o^b(x^V(S), y^V(x^S(S)))$

<side effect> $r^S(S) = y^S(x^S(S))$

<Boolean primary> t ::= <logical value> u; w ::= <variable> v; x ::= <function designator> f; y ::= <relation> r; z ::= '(' <Boolean expression> b ')'

<value> $t^V(S) \equiv u^V$; $w^V(S) = S(v)$; $x^V(S) = f^V(S)$; $y^V(S) = r^V(S)$; $z^V(S) = b^V(S)$

<side effect> $t^S(S) = S$; $w^S(S) = S$; $x^S(S) = f^S(S)$; $y^S(S) = r^S(S)$; $z^S(S) = b^S(S)$

<Boolean secondary> s ::= <Boolean primary> p; t ::= '¬'
<Boolean primary> q

<value> $s^V(S) = p^V(S)$; $t^V(S) = \underline{not}(q^V(S))$

<side effect> $s^S(S) = p^S(S)$; $t^S(S) = q^S(S)$

<Boolean factor> x ::= <Boolean secondary> s; y ::= <Boolean factor> z '∧' <Boolean secondary> t

<value> $x^V(S) = s^V(S)$; $y^V(S) = \underline{and}(z^V(S), t^V(z^S(S)))$

<side effect> $x^S(S) = s^S(S)$; $y^S(S) = t^S(z^S(S))$

<Boolean term> x ::= <Boolean factor> f; y ::= <Boolean term> z '∨' <Boolean factor> g

<value> $x^V(S) = f^V(S)$; $y^V(S) = \underline{or}(z^V(S), g^V(z^S(S)))$

<side effect> $x^S(S) = f^S(S)$; $y^S(S) = g^S(z^S(S))$

<implication> x ::= <Boolean term> t; y ::= <implication> z '⊃' <Boolean term> u

<value> $x^V(S) = t^V(S)$; $y^V(S) = \underline{implies}(z^V(S), u^V(z^S(S)))$

<side effect> $x^S(S) = t^S(S)$; $y^S(S) = u^S(z^S(S))$

<simple Boolean> x ::= <implication> i; y ::= <simple Boolean> z '≡' <implication> j

<value> $x^V(S) = i^V(S)$; $y^V(S) = \underline{equivalence}(z^V(S), j^V(z^S(S)))$

<side effect> $x^S(S) = i^S(S)$; $y^S(S) = j^S(z^S(S))$

<Boolean expression> x ::= <simple Boolean> s; y ::= <if clause> i <simple Boolean> t 'else' <Boolean expression> z

<value> $x^V(S) = s^V(S)$; $y^V(S) = \underline{if}\ i^V(S)\ \underline{then}\ t^V(i^S(S))\ \underline{else}\ z^V(i^S(S))$

<side effect> $x^S(S) = s^S(S)$; $y^S(S) = \underline{if}\ i^V(S)\ \underline{then}\ t^S(i^S(S))\ \underline{else}\ z^S(i^S(S))$

In these rules, and, or, implies, and equivalence are assumed to be Boolean-valued functions of two Boolean values, defined in the

usual way, while not(true) = false and not(false) = true. Also, the functions equal and less are defined by equal(x, y) = if x=y then true else false and less(x, y) = if x<y then true else false, while greater(x, y) = less(y, x), notequal(x, y) = not(equal(x, y)), notless(x, y) = not(less(x, y)), and notgreater(x, y) = not(greater(x, y)), for any two real numbers x and y (recalling again that type conversion has been omitted). Where arithmetic expressions require the definition of four nonterminals (primary, factor, term, simple arithmetic expression) because of the three precedence levels, Boolean expressions require these four plus two more (secondary, implication), because there are five precedence levels for Boolean operators.

A number of points about the combining of simple arithmetic and Boolean expressions using if, then, and else bear mention. The expression following else, in each case, is a general expression, and, in particular, may itself involve if. Thus

if a>b then c else if d=e then f else g

is legal, whether c, f, and g are all arithmetic or all Boolean variables. The expression between then and else, however, is a simple expression; thus

if a>b then if d=e then c else f else g

is not legal, although we may make it so by introducing parentheses, thus:

if a>b then (if d=e then c else f) else g

This is legal because any expression in parentheses is a primary, and thus a factor, term, etc., and ultimately a simple arithmetic or Boolean expression.

In the semantic rules involving if, then, and else, we note that the side effect of the if clause (or, equivalently, of the Boolean

expression it contains) is taken regardless of whether the value of the if clause is true or false; however, only one of the other side effects is taken in any case. This corresponds to evaluating either b or c in the expression if a then b else c, but not both. The value of such an expression is calculated after the side effect of a, but not after the side effect of b, even if the value of a is false.

NOTES

Many of the syntax rules in this section are taken directly from the ALGOL report [Naur et al. 63].

Much work has been done on the subject of precedence. If the syntactic rules of a language satisfy certain restrictions [Floyd 63], the language is called an operator precedence language, and there is a simple syntax-checking algorithm for it. Floyd's restrictions have been relaxed, and the corresponding syntax-checking algorithms extended, in [Wirth and Weber 66] and further in [McKeeman 66]. An excellent account of this work is given in [Feldman and Gries 68].

The state vector concept is fundamental to all work in programming science. It has been rediscovered at least eight times. [Podlovchenko 62] contains the first general account of state vectors as content functions (in Russian, sostoyaniya pamyati or "memory states"), although the specific abstract computer defined in [Kaphengst 59] has Maschinenstellungen, or "machine states." The term "state vector" is introduced in [McCarthy 63], and the term "content function" in [Elgot and Robinson 64]. State vectors are called "snapshots" in [Naur 66], "the content of the store" in [Strachey 66], and simply "states" in [Maurer 66]. Other writers use state vectors without giving them any special names; thus [Engeler 67] refers to a sequence of elements of an "underlying set" (of values), each of which is presumably the current value of a variable, while [Cooper 69] denotes by $\{L\}$ the (current) set of values of the registers in a program scheme. The "composite objects" of the so-called Vienna Definition Language [Lucas and Walk 69] are generalizations of state vectors.

The idea that the value of an expression is a function of the current state of the computation first appears in [McCarthy 63]; here the notation val(t, ξ) is used, where t is a term and ξ is the current

state vector. In [Strachey 66], if ϵ is an expression and σ is the content of the store (that is, the current state of the computation), then $R(\epsilon, \sigma)$ is the current value of ϵ, as it appears on the right side of an assignment, while $L(\epsilon, \sigma)$ is its current address, if any, which is its value when it appears on the left side. Some of our semantic rules for expressions appear in simplified form in [Burstall 70]; thus, if plus(e, e') is the expression which is the sum of the expressions e and e', and val(e, s) is the value of e, given the current state s, then Burstall gives the equation val(plus(e, e'), s) = val(e, s) + val(e', s).

Inherited attributes are the principal contribution of Knuth's paper mentioned earlier [Knuth 68]. The algorithm given in this paper to test whether a language definition involving synthesized and inherited attributes is inherently "circular" -- that is, whether the process of calculating the attributes goes on indefinitely and does not terminate -- is incorrect and has been corrected in [Knuth 71].

EXERCISES

1. Give the values of the expression 5*6-4*3, subject to the conditions that * denotes multiplication, - denotes subtraction, and:

(a) * and - have equal precedence.

(b) * has higher precedence than - .

(c) - has higher precedence than * .

2. Using the syntax rules

<expression> ::= <term> | <term> '+' <expression> | <term> '-' <expression>

<term> ::= <factor> | <term> '*' <factor> | <term> '/' <factor>

<factor> ::= 'A' | 'B' | 'C' | 'D' | 'E' | 'F'

give derivation trees for:

(a) A*B*C-D

(b) A-B/C-D

(c) A*B*C/D*E*F

(d) A+B+C/D/E+F

3. Let v(S) be the value of the expression 2*I+3*J-4 (in the usual sense) when the current state vector is S. Find the value of v(S) when S is:

(a) {I = 12, I3 = 6, I4 = 7, J = 0}

(b) {I = 12, J = 6, K = 7, L = 0}

(c) {A = 2.5, A0 = -2.5, I = 0, J = 0}

(d) {I = 17, J = 18}

4. How many different state vectors are there with 2^n components, each of which is capable of assuming 2^b distinct values? (Such state

vectors might express the current state of the core memory of a computer with an n-bit address field and b bits per full word.)

5. (a) Give rules for function names in FORTRAN II, each of which must start with a letter, must consist of letters and digits only, and must end with the letter F. Any function name starting with the letter X is of type integer; all other function names are of type real. (These rules were actually used in a very old version of FORTRAN II.) Give the type of each function name as a semantic attribute of it.

(b) Give rules for identifiers in an assembly language, such that each identifier must start with a letter and contain only letters, numbers, and dollar signs. Give the type of each such identifier as a semantic attribute, having two values, system and user; every identifier containing at least one dollar sign has type system, and all other identifiers have type user.

6. Consider the syntactic rule

<expression> a ::= <term> t; b ::= '+' <term> u; c ::= '-' <term> v; d ::= <expression> y '+' <term> w; e ::= <expression> z '-' <term> x

Give semantic rules to be associated with this rule, for the semantic attributes <value> and <type>, which

(a) forbid mixed mode,

(b) permit mixed mode,

in analogy with the semantic rules for values and types given with the syntactic rule

<expression> e ::= <term> v '+' <term> w

in section 2-3.

7. Which of the following situations require certain of the semantic attributes to be inherited (not necessarily the ones listed)?

(a) The type of an identifier is to be an attribute of it; assume that type declarations are not allowed, but that an identifier is of type integer if and only if its starts with any one of a certain set of letters specified in the first statement of the given program.

(b) Identifiers are as in FORTRAN II, and the type of an expression of the form IF b THEN x ELSE y is to be an attribute of it, where x and y are not required to be of the same type.

(c) The rank of an identifier is to be an attribute of it, where by the rank we mean the number of subscripts (zero for a non-subscripted variable), this rank to be determined by FORTRAN-style DIMENSION statements.

(d) An octal integer is defined, syntactically, to be an unsigned integer (in the usual sense, as, for example, in section 1-1) followed by the letter B. The value of an octal integer is to be an attribute of it.

(NOTE: In his paper on semantic attributes, Knuth shows that any syntactic and semantic definition involving both synthesized and inherited attributes is equivalent to one involving synthesized attributes only. However, the new attributes are not necessarily all the same as the old ones. In the above problems, certain attributes have been specified, and in some cases these attributes cannot be defined unless certain inherited attributes are used.)

8. Give semantic rules for type functions, according to the semantic conventions given in section 2-4, for the arithmetic expressions and associated non-terminals (not those associated with Boolean expressions) given in section 2-5.

9. Give derivation trees (syntax only) for the following arith-

metic expressions. Use the syntax rules of section 2-5, except that the rule

<variable> ::= 'A' | 'B' | 'C' | 'D'

is to be used instead of the rule for variables given in that section.

(a) A×B×C (term)

(b) A/B↑C×D (term)

(c) A×B+C-D (simple arithmetic expression)

(d) A-B×C×D (arithmetic expression)

10. Give derivation trees for the following Boolean expressions, as in problem 9 above:

(a) A∧B∨C∧D (Boolean term)

(b) A∨B∧C∨D (implication)

(c) ¬A∧¬B (simple Boolean)

(d) if A∨B then ¬C else D⊃E (Boolean expression)

C H A P T E R T H R E E

S T A T E M E N T S

3-1 Syntax of Statements

Syntactic rules for statements, in general, are much simpler than syntactic rules for expressions. As an example, we give a general syntactic rule for assignment statements:

<assignment> ::= <variable> ':=' <expression>

There is not much more to assignment statements, syntactically, than this. The symbol := may be replaced by = , as in FORTRAN or PL/I; the variable may be allowed to be subscripted; and we may want to distinguish two kinds of expressions, such as arithmetic expressions and Boolean expressions in ALGOL. Also, experimental languages have been constructed in which the assignment operator is treated much like the other operators in an expression, and expressions take the place of assignments. But in most practical cases, syntax rules for assignments are simple, and, in particular, non-recursive.

Multiple assignments, in which more than one assignment symbol (normally := or =) is present, involve what are called left part lists in ALGOL. The ALGOL assignment

A:=B:=C:=0.0

has A:=B:=C:= as its left part list; this is a sequence of left parts, in this case three of them, A:= , B:= , and C:= . In extended versions of FORTRAN, this would be written A=B=C=0.0, while in PL/I it would be

A,B,C=0.0 . (The statement A=B=C in PL/I sets A equal to (B=C), that is, to 1 if B=C and to 0 otherwise.) The ALGOL rule for left part lists is

<left part list> ::= <left part> | <left part list> <left part>

where a left part is either a variable identifier or a procedure identifier followed by := , that is,

<left part> ::= <variable identifier> ':=' | <procedure identifier> ':='

(Here the procedure identifier is assumed to be the name of a procedure which contains the given assignment statement, and refers to the quantity which will be treated as the value of the procedure when exit is made.) An ALGOL assignment is now a left part list followed by an expression, that is,

<assignment statement> ::= <left part list> <arithmetic expression> | <left part list> <Boolean expression>

since two kinds of expression must be distinguished in ALGOL.

Transfer statements are of various kinds. Most languages have a simple GO TO statement of the form

<GO TO statement> ::= 'GO TO ' <label>

or something similar; in FORTRAN, <label> is replaced by <statement number>. More complex GO TO statements involve the use of lists of labels separated by commas, defined by

<label list> ::= <label> | <label list> ',' <label>

In ALGOL, instead of labels, we have here general designational expressions, which may, for example, be constructed using if, then, and

else. The label list is here called a switch list, and is used in defining switch declarations such as switch s := a, b, c where a, b, and c are labels; thus go to s[i] would transfer control to c, for example, if i were equal to 3, or a if i were equal to 1. The ALGOL definition of a switch declaration is thus

<switch declaration> ::= 'switch' <switch identifier> ':=' <switch list>

where switch lists are defined by

<switch list> ::= <designational expression> | <switch list> ',' <designational expression>

and the general GO TO statement is simply

<go-to statement> ::= 'go to' <designational expression>

In FORTRAN, however, the label list is part of the GO TO statement itself. Here there are three kinds of GO TO statement -- the simple form discussed above and the assigned and computed GO TO statements, definable syntactically by

<assigned GO TO statement> ::= 'GO TO ' <variable> ',(' <label list> ')'

<computed GO TO statement> ::= 'GO TO (' <label list> '),' <variable>

(The comma is sometimes omitted from the computed GO TO statement.)

Conditional statements are also of various kinds. In FORTRAN, there is an arithmetic IF statement defined by

<arithmetic IF statement> ::= 'IF(' <arithmetic expression> ')' <statement number> ',' <statement number> ',' <statement number>

and a logical IF statement defined by

<logical IF statement> ::= 'IF(' <Boolean expression> ')'
<statement>

Strictly speaking, Boolean expressions are called logical expressions in FORTRAN; also, the statement which appears as part of a logical IF statement is somewhat restricted, and, in particular, cannot be a DO statement or another IF statement. Analogous, although somewhat weaker, restrictions exist in ALGOL. After the if clause (which, we recall from section 2-5, consists of the word if, followed by a Boolean expression, followed by then) there may come any statement, except that if it is another if statement it must be enclosed between the words begin and end (thus making it a compound statement of a rather degenerate kind), while if it is a for statement it cannot be followed by else unless it is so enclosed. After the word else, if it appears, may come any statement, including, of course, a compound statement or block enclosed between the words begin and end. The ALGOL rule for if statements is simply

<if statement> ::= <if clause> <unconditional statement>

where an unconditional statement may be a compound statement or block, enclosed in begin and end, or a "basic statement" which may be an assignment, go-to, or procedure statement -- in other words, just about anything except another if statement or a for statement. A conditional statement may then be either an if statement, an if statement followed by else and an arbitrary statement, or an if clause followed by a for statement. (All of the above discussion omits the treatment of labels, which is taken up in section 4-1.)

Iteration statements differ depending on whether the iteration is restricted to a regularly increasing sequence of values, or whether

it is allowed to be more general. In FORTRAN, we may define DO statements by the syntactic rule

<DO statement> ::= 'DO' <statement number> <variable> '=' <two or three expressions>

where "two or three expressions" is defined by

<two or three expressions> ::= <expression> ',' <expression> | <expression> ',' <expression> ',' <expression>

In ALGOL, a for statement is a for clause followed by an arbitrary statement, where for clauses are defined by

<for clause> ::= 'for' <variable> ':=' <for list> 'do'

To restrict ALGOL iterations to approximately the scope of FORTRAN iterations, one would define for lists by the rule

<for list> ::= <arithmetic expression> 'step' <arithmetic expression> 'until' <arithmetic expression>

thus making them correspond roughly to the "two or three expressions" of FORTRAN, as defined above. Actually, however, for lists are sequences of for list elements, each of which may be of the above form, of the "while" form (that is, an arithmetic expression, followed by while, followed by a Boolean expression), or simply an expression. (For special reasons, the semantics of iteration is postponed until the chapter on programs and their effects.)

Subroutine calling statements are very easy to define syntactically. In FORTRAN, a subroutine is called by using the word CALL followed by a subroutine reference; in ALGOL, even the word CALL is omitted. The structure of subroutine references themselves will be taken up at the end of section 5-4.

3-2 Semantics of Assignment

The effect of an assignment statement is to change the current state of the computation to a new state. In the case of a simple assignment without side effects, the new state is the same as the old, except for the value of one variable. In general, however, this need not be the case.

We will treat the word effect, as it is used above, as a technical term meaning a function of the current state of some computation whose value is the next state. Thus if $\mathcal{S}$ is the set of all possible states of a computation, then an effect e is a function from $\mathcal{S}$ to $\mathcal{S}$; if S is the current state, then we write e(S) = S', where S' is the next state. (Side effects are effects in this sense.)

How can we define an effect in a precise way? If e is an effect and e(S) = S', then we must describe how S' is obtained from S. Regarding S' as a vector, we must be able to determine every component of that vector in terms of the components of the vector S. Or, to put it another way, regarding S' as a (content) function, we must be able to determine S'(x), for every x in the domain of S', in terms of the various values of S(x).

In the simplest case, where the assignment a assigns the current value of the expression e to the variable v, we may write

<assignment> a ::= <variable> v ':=' <expression> e

<effect> $a^e(S) = S'$, where $S'(v) = e^v(S)$, $S'(z) = S(z)$ for $z \neq v$

assuming that the assignment symbol := is used. The semantic rule here defines S'(v) first, and then S'(z) for all $z \neq v$; thus it gives a complete description of the new state S' in terms of the old

state S. It is assumed that the expression e has a value e^V, which is a function of S; this value becomes the new value of the variable v, while every other variable has the same value as it did before.

Let us now assume that the expression e has a side effect e^S in addition to its value. In this case the new state vector $a^e(S)$ will be the same as the new state vector $e^S(S)$ produced by the side effect, except that the new value of v will be $e^V(S)$, just as it was before. Thus in this case we may write

<assignment> a ::= <variable> v ':=' <expression> e
<effect> $a^e(S) = S'$, where $S'(v) = e^V(S)$, $S'(z) =$
$S''(z)$ for $z \neq v$, where $S'' = e^S(S)$

Another level of complication in assignments arises when the variable v may be subscripted. In this case the new state vector will have one of its variables set to the current value of the given expression, as before; but which variable is set to this value depends on the old state vector. Consider, for example, the assignment A(I-7) = A(I-6), where A is given in FORTRAN by DIMENSION A(3). If e is the effect of this assignment, and

$$S = \{I = 9,\ A(1) = 2.5,\ A(2) = 5.0,\ A(3) = -12.5\}$$

then $e(S) = S'$, where

$$S' = \{I = 9,\ A(1) = 2.5,\ A(2) = -12.5,\ A(3) = -12.5\}$$

whereas if

$$U = \{I = 8,\ A(1) = 2.5,\ A(2) = 5.0,\ A(3) = -12.5\}$$

then $e(U) = U'$, where

$$U' = \{I = 8,\ A(1) = 5.0,\ A(2) = 5.0,\ A(3) = -12.5\}$$

Thus a variable occurring on the left side of an assignment will have a new attribute giving the variable (that is, an element of the domain of the state vectors under consideration) as a function of the current state vector. This is called the L-value (L for left side) of the variable. In the above example, if v^l is the L-value of the variable v, then $v^l(S)$ is the variable A(2), while $v^l(U)$ is the variable A(1). The relation between the L-value of a variable and its ordinary value (sometimes called its R-value) is

$$v^v(S) = S(v^l(S))$$

where v^v is the value, and v^l the L-value, of the variable v. Unsubscripted variables, of course, have constant L-values; that is, their L-values do not depend on the current state vector.

If the variable v has an L-value v^l, the assignment rule above may be rewritten

<assignment> a ::= <variable> v ':=' <expression> e

<effect> $a^e(S) = S'$, where $S'(v^l(S)) = e^v(S)$, $S'(z)$
$= S''(z)$ for $z \neq v^l(S)$, where $S'' = e^s(S)$

Further complications ensue if the left side may have a side effect. In FORTRAN, as it was originally defined, this cannot occur, because subscripts are restricted to certain very simple forms. However, in most languages allowing subscripted variables, including ALGOL, these may include function references as part of a subscript, and such function references may have side effects. At this point there are two plausible interpretations, each of which leads to its own semantic rule. For example, consider the assignment

A[F(3)] := F(8)

where F(I) = K + I, for a constant K which is incremented by one each

time F is called. If K is initially set to 1, we may interpret this in either of the following ways:

(a) Calculate F(8), which is 9; then set A(5) = 9 (since K has been incremented by 1 in calculating F(8), and therefore F(3) is now 5); or

(b) Calculate F(3), which is 4, then set A(4) = 10 (since K has been incremented by 1 in calculating F(3), and therefore F(8) is now 10).

The first of these methods corresponds to our intuitive ideas of calculating the value of the right side first, and then worrying about the left side. It corresponds, in our terminology, to the assignment rule

<assignment> a ::= <variable> v ':=' <expression> e

<effect> $a^e(S) = S'$, where $S'(v^\ell(e^S(S))) = e^V(S)$, $S'(z) = S''(z)$ for $z \neq v^\ell(e^S(S))$, where $S'' = v^S(e^S(S))$

assuming that v^S is the side effect of v. The second method corresponds to our intuitive ideas, at a lower level, that "when in doubt, always do things from left to right"; it corresponds to the rule

<assignment> a ::= <variable> v ':=' <expression> e

<effect> $a^e(S) = S'$, where $S'(v^\ell(S)) = e^V(v^S(S))$, $S'(z) = S''(z)$ for $z \neq v^\ell(S)$, where $S'' = e^S(v^S(S))$

In fact, the informal semantic description of assignments given in the ALGOL report specifies that the left side is to be evaluated first, as we have done here.

Assignments, like expressions, are affected by type conversion. Even in languages (such as FORTRAN II) in which mixed mode in expressions is not permitted, it is almost always allowed to set the

value of a variable of one type to the value of an expression of a different type, after suitable conversion. The conversion function to be used, of course, depends on the two types involved; in FORTRAN, for example, we may define

ntcf(real, real) = ident ntcf(integer, real) = fix
ntcf(real, integer) = float ntcf(integer, integer) = ident

where ntcf(vt, et) is the name of the type conversion function needed to convert an expression-value of type et to the type vt of the variable on the left. (The fix function normally truncates its argument; thus, for example, fix(3.99) is 3, not 4.) In order to use such functions with arguments, we shall use the convention, suggested by the language LISP, according to which

$$\text{apply}(f, x_1, \ldots, x_n) = f(x_1, \ldots, x_n)$$

Here f, of course, may itself be calculated by the use of another function, such as ntcf above. Using this convention, we may modify the last rule above for assignment statements as follows, assuming that v^t and e^t give the type of v and the type of e respectively:

<assignment> a ::= <variable> v ':=' <expression> e
<effect> $a^e(S) = S'$, where $S'(v^\ell(S)) = \text{apply}(\text{ntcf}(v^t, e^t), e^v(v^s(S)))$, $S'(z) = S''(z)$ for $z \neq v^\ell(S)$, where $S'' = e^s(v^s(S))$

We may also wish to replace e^t by $e^t(S)$ in the above equation when the type of an expression is in fact a function of the current state vector. ALGOL, for example, does not outlaw expressions of the form if b then a else i where a is real and i is integer; the type of such an expression obviously depends on the current value of b.

Let us now consider how to give semantic rules for multiple

assignments. As before, we shall assume that all processing is carried out from left to right. This statement must be made in the case of multiple assignments even without side effects, as may be seen by considering

A[i] := i := i+1

in ALGOL. If the initial value of i is 5, then either A[5] or A[6] is set to 6, depending upon whether processing proceeds from left to right or right to left. (The same thing happens in PL/I with A(I),I=I+1 .) Nor is it enough to say that the assignments are made from left to right, because this would seem to imply, for example, that both A[5] and A[6] would be set to 6 in the assignment

A[i] := i := A[i] := i+1

and this is clearly not the case. What happens is that the subscripts are evaluated from left to right, to enable us to decide what cells have their values changed; then the expression on the right is evaluated, and its value placed in each of these cells. It follows that a left part list, in the ALGOL sense, has associated with it a set of L-values, rather than a single L-value. If this set is called p^z, for the left part list p, while p^s is the side effect of p and p^t the type of p, one may give a semantic rule to go with the ALGOL syntactic rule for assignment statements:

<assignment statement> x ::= <left part list> p <arithmetic expression> a; y ::= <left part list> q <Boolean expression> b

<effect> $x^e(S) = S'$, where $S'(z) = apply(ntcf(p^t, a^t), a^v(p^s(S)))$ for $z \in p^z(S)$, $S'(z) = S''(z)$ for $z \notin p^z(S)$, where $S'' = a^s(p^s(S))$; $y^e(S) = S'$, where $S'(z) = apply(ntcf(q^t, b^t), b^v(q^s(S)))$ for $z \in q^z(S)$, $S'(z) = S''(z)$ for $z \notin q^z(S)$, where $S'' = b^s(q^s(S))$

We note that in ALGOL a left part list must have a single type; that is, multiple assignments such as

$$a := i := j$$

where a is real and i is an integer, are not permitted. Let us derive the attributes of left part lists from the corresponding attributes for left parts. Remembering that processing takes place from left to right, the rules may be written as follows:

<left part list> x ::= <left part> p; y ::= <left part list> z <left part> q

<type> $x^t = p^t$; $y^t = z^t$

$(z^t = q^t)$

<set of L-values> $x^z(S) = p^{\ell}(S)$; $y^z(S) = z^z(S) \cup \{q^{\ell}(z^s(S))\}$

<side effect> $x^s(S) = p^s(S)$; $y^s(S) = q^s(z^s(S))$

Here it is assumed that left parts have L-values, with superscript ℓ, and side effects, with superscript s.

Our definition of ALGOL assignments is still not complete, because it fails to take into account the case in which the evaluation of the expression on the right is never completed. This case will be taken up in section 4-3.

3-3 Transfer Statements

We now pass to other kinds of statements besides assignment.

The logical IF statement whose syntax was given in section 3-1 has an effect, much like the effect of an assignment statement, which may be specified semantically as follows:

<logical IF statement> s ::= 'IF(' <Boolean expression> b ')' <statement> t

<effect> $s^e(S) = \underline{\text{if}}\ b^v(S)\ \underline{\text{then}}\ t^e(b^s(S))\ \underline{\text{else}}\ b^s(S)$

Here it is assumed that the Boolean expression b has a value b^v and a side effect b^s; we may replace $b^s(S)$ by S if no side effect is present. If the statement t is an assignment statement, then t^e is its effect, as specified in the preceding section. Even if t is not an assignment statement, however, it must, clearly, have an effect in this sense.

The arithmetic IF statement given in section 3-1 does not change the values of any of the program variables. We may say that it "does not have an effect"; but what we mean is that its effect is the identity function -- that is, if s is such a statement and s^e is its effect, then $s^e(S) = S$. The arithmetic IF statement, however, also determines the statement number of the next statement as a function of the current state vector. We could, if we wanted to, make this into a semantic attribute as follows:

<IF statement> i ::= 'IF(' <arithmetic expression> e ')' <statement number> a ',' <statement number> b ',' <statement number> c

<next statement number> $i^z(S) = \underline{\text{if}}\ e^v(S) < 0\ \underline{\text{then}}$ a $\underline{\text{else if}}\ e^v(S) = 0\ \underline{\text{then}}$ b $\underline{\text{else}}$ c

Unfortunately, this type of rule does not extend to the logical IF statement. Consider, for example, the statement IF (K=0) GO TO 25. If the current state vector S specifies K = 0 (that is, if S(K) = 0), and if z is the attribute giving the next statement number, we may write z(S) = 25. But what is z(S) if S(K) ≠ 0? If statement number 23, for example, immediately follows this IF statement, then we might write z(S) = 23. But it is not necessary, of course, for an arbitrary statement in FORTRAN to have a statement number at all.

Let the statements in a program be numbered in sequence from 1 through n. If a statement is numbered k in this way -- that is, if it is the kth statement in its program -- then we say that its statement index is k. What we would like to associate with an IF statement is an attribute giving the statement index of the next statement, as a function of the current state of the computation. It should be clear from the discussion of the preceding chapter that this cannot be done without introducing inherited attributes. A statement, by itself, contains no statement index information; this can only be inherited from the program of which that statement is a part.

There are two inherited attributes which we need. First, given a statement, we want to know its own statement index. Second, we need a function which gives, for each statement number in the program (or, in other languages, for each label), the corresponding statement index. Using these, we may give syntactic and semantic rules for a general FORTRAN IF statement as follows:

<IF statement> f ::= 'IF(' <arithmetic expression> e ')'
<statement number> x ',' <statement number> y ','
<statement number> z; g ::= 'IF(' <Boolean
expression> b ')' <statement> t
<statement index>^{i}
<label function>^{a}

$t^i = g^i$

<effect> $f^e(S) = e^s(S)$; $g^e(S) =$ *if* $b^v(S)$ *then* $t^e(b^s(S))$ *else* $b^s(S)$

<exit index> $f^x(S) =$ *if* $e^v(S) < 0$ *then* $f^a(x)$ *else if* $e^v(S) = 0$ *then* $f^a(y)$ *else* $f^a(z)$; $g^x(S) =$ *if* $b^v(S)$ *then* $t^x(b^s(S))$ *else* $g^i + 1$

The new attribute, "exit index," gives the statement index of the next statement to be executed; we have assumed that the (more or less arbitrary) statement t has an exit index t^x as well as an effect t^e. We have also assumed, as before, that the arithmetic expression e and the Boolean expression b have values e^v and b^v respectively and side effects e^s and b^s respectively. The inherited attribute equation $t^i = g^i$ sets the statement index of t to be the same as that of g; it reflects the fact that if a logical IF statement is the kth statement of its program, then the statement which it contains, being part of this kth statement, can also be said to have statement index k.

We must now give rules for the effect and the exit index of a more general statement. For simplicity, let us assume that there are only three kinds of statements: assignments, IF statements, and GO TO statements. Then the general rule for statements may be drawn up as follows:

<statement> x ::= <assignment statement> a; y ::= <GO TO statement> g; z ::= <IF statement> i

<statement index>^{i}

$a^i = x^i$; $g^i = y^i$, $i^i = z^i$

<label function>^{a}

$a^a = x^a$; $g^a = y^a$; $i^a = z^a$

<effect> $x^e(S) = a^e(S)$; $y^e(S) = g^e(S)$; $z^e(S) = i^e(S)$

<exit index> $x^x(S) = a^x(S)$; $y^x(S) = g^x(S)$; $z^x(S) = i^x(S)$

This rule is typical of those in which one nonterminal is defined syntactically as one of several other nonterminals, with no string concatenation and no terminals in the syntactic rule. The synthesized attributes are often simply "passed up the tree" and the inherited attributes "passed down the tree," as here. That is, the value of each synthesized attribute of the nonterminal on the left is given as the corresponding value of the same attribute of each nonterminal on the right; and the reverse is true for the inherited attributes. (The statement index and the label function of a statement, of course, must be further inherited from the program containing that statement. Rules for doing this are given in the following chapter.)

It remains to specify the effect of a GO TO statement and the exit index of an assignment or GO TO statement. Normally, an assignment statement exits to the following statement, that is, to the statement whose index is one greater than its own. Thus by adding the semantic rules

<statement index>^{i}

<exit index> $a^x(S) \equiv a^i + 1$

to any definition of an assignment statement a, we may supply the definition of exit index needed in this case. (Again, this is in the absence of the special type of side effect to be discussed in section 4-3.) A simple GO TO statement may be defined very easily as

<GO TO statement> g ::= 'GO TO ' <statement number> n

<statement index>^{i}

<label function>^{a}

<effect> $g^e(S) = S$

<exit index> $g^x(S) \equiv g^a(n)$

In fact, if GO TO statements are restricted to this simple form, we could eliminate the mention of their statement index and the equation $g^i = y^i$ in the inherited attribute equation list for statement indices of a statement; also, we could eliminate the effect of a GO TO statement and replace $g^e(S)$ by S in the rule for the effect of a statement. Many languages, however, have more complex forms of GO TO statements. For the assigned GO TO statement whose syntax was given in section 3-1, the label list gives a range of values for the given variable. This is now a label-valued variable; its set of values (see the end of section 2-2) is a set of labels. Such variables are set by ASSIGN statements, whose syntax and semantics is

<ASSIGN statement> s ::= 'ASSIGN' <statement number> n 'TO' <identifier> i

<effect> $s^e(S) = S'$, where $S'(i) = n$, $S'(z) = S(z)$ for $z \neq i$

The syntax and semantics of assigned GO TO statements is then

<assigned GO TO statement> s ::= 'GO TO' <variable> v ',(' <label list> ')'

<effect> $s^e(S) = S$

<label function>^{a}

<exit index> $s^x(S) = s^a(S(v))$

with the label function inherited as before.

For the computed GO TO statement of section 3-1, let us assume that the label list has as its attribute a label-valued function of integers. Thus the label list

23, 24, 25, 99

is associated with a function f such that f(1) = 23, f(2) = 24, f(3) = 25, and f(4) = 99. The label list also has a length, which is in this case 4. If we denote the length of the label list a by a^n, and the associated function as above by a^f, we may formulate the syntax and semantics of computed GO TO statements as

<computed GO TO statement> s ::= 'GO TO(' <label list> a '),' <variable> v

<effect> $s^e(S) = S$

<label function>^{a}

<exit index> $s^x(S) = \underline{\text{if}}\ 1 \leq S(v) \leq a^n\ \underline{\text{then}}\ a^f(S(v))$ $\underline{\text{else}}$ error

It is assumed here that if we attempt to execute such a statement at a time when its variable is not in bounds -- that is, between 1 and the length of the label list, inclusive -- a run-time error should result. Some FORTRAN systems, on the other hand, allow the program to proceed normally (without transferring) in this case, and if we wanted our semantics to reflect this fact we would introduce the statement index as an inherited attribute and proceed as we did with the exit index of an assignment statement.

A go to statement in ALGOL involves a designational expression, whose value is a function of the current state vector. A designational expression may also have a side effect; it is permitted, for example, to go to A[f(i)] where A is a switch and f is an integer-valued procedure which changes the value of some variable. Ignoring side effects for the moment, we may give the syntax and semantics of designational expressions as follows:

<simple designational expression> x ::= <label> a; y ::=
<switch designator> d; z ::= '(' <designational
expression> e
<label function>^{a}
$d^a = y^a$; $e^a = z^a$
<exit index> $x^X(S) \equiv x^a(a)$; $y^X(S) = d^X(S)$; $z^X(S) = e^X(S)$

<designational expression> x ::= <simple designational
expression> d; y ::= <if clause> i <simple
designational expression> e 'else' <designational
expression> z
<label function>^{a}
$d^a = x^a$; $i^a = y^a$; $e^a = y^a$; $z^a = y^a$
<exit index> $x^X(S) = d^X(S)$; $y^X(S) =$ if $i^V(S)$ then
$e^X(S)$ else $z^X(S)$

Go to statements may then be given simply by

<go-to statement> g ::= 'go to' <designational expression> e
<label function>^{a}
$e^a = g^a$
<effect> $g^e(S) = S$
<exit index> $g^X(S) = e^X(S)$

Note how the label function must be inherited through these definitions in order to be applied at the point where a designational expression is identified as a particular label.

3-4 Input-Output Statements

A simple input statement which reads a new value for the variable X has an effect, in the sense defined above, since after it is executed the new state vector is different from the old one; the value of X has changed. This effect, however, is not a function of the values of the internal variables of the program, but rather of the current state of the input tape, the input deck, or the input medium in general. A simple output statement which prints out the value of the variable X has no effect on the internal variables of the program; the new state of these variables is the same as the old one. However, such a statement changes the current state of the output medium.

If we are to include input-output statements in the analysis of the preceding section, we must give to each such statement an effect and and exit index, as functions of the current state vector. The exit index is obvious; such statements do not transfer, and we may therefore do the same thing we did for assignment statements. To define an effect, however, we shall have to make what may seem a radical assumption: mathematically, we shall make no distinction between external and internal variables. Because of this, when we speak of the "current state of the computation," or the "current state vector," as a content function, we shall mean a function of all variable quantities involved in the computation -- input and output media and their positioning, as well as what are ordinarily called variables. The current state of the internal variables, the current state of the input media, and the current state of the output media are therefore merely restrictions of the total current state of the computation to a particular subset of its domain as a content function.

The simplest input/output media are one-way. A one-way input medium, such as a deck of cards or a paper tape to be read, may be thought of as a variable whose values are sequences of data to be read. If t is such a variable, and $(x_1, x_2, \ldots, x_n)$ is its current value, then, after a read operation, its current value will be $(x_2, \ldots, x_n)$. The piece of data denoted by x_1, having been read, is no longer part of the value of t; or, to put it another way, the sequence which is the current value of t consists of those items which are still to be read, starting with the item which will be read next. A one-way output medium, such as an output sheet or a paper tape being punched, may be considered as a variable whose values are sequences of data which have thus far been output. If t is such a variable, and the item currently being output is x, then the current value of t is changed from $(x_1, \ldots, x_n)$ to $(x, x_1, \ldots, x_n)$.

A sequential, two-way input-output medium, such as a tape used to store intermediate results of a computation, cannot be represented simply as a variable whose values are the sequences of information which it currently contains. It is necessary also to know the position of the reading and writing heads with respect to the tape; this position varies, of course, as the tape moves back and forth. Perhaps the best way to model this situation is to consider the tape as an array of records, each of which is treated as a variable. If there are n records on the tape t, then t[1], ..., t[n] are n separate variables, whose values are the possible contents of a given record; there is then also another variable h_t, representing the tape head for this particular tape, and its values are the variables t[1], ..., t[n] themselves (or, alternatively, the integers from 1 through n). If the value of h_t is t[k], then t[k] is being read or written next. We may obtain an alternate model of a one-way input medium by considering it as a special case of a two-way medium modeled in this way.

As an example of input-output on one-way media, let us postulate two variables, reader and printer, such that S(reader) and S(printer), for the state vector S, are respectively the current sequence of records waiting to be read and the current sequence of records which have thus far been printed. For the moment, we shall also assume that each record consists of the value of a single variable. Simple READ and PRINT statements such as

READ X, Y, Q1

PRINT I1, I2, I3, I4, I5

(taken from the language BASIC) may now be described as follows:

<READ statement> x ::= 'READ' <variable> y; y ::= <READ statement> z ',' <variable> w

<effect> $x^e(S) = S'$, where if $S(\text{reader}) = (x_1, x_2, \ldots, x_n)$ then $S'(\text{reader}) = (x_2, \ldots, x_n)$, $S'(v) = x_1$, and $S'(z) = S(z)$ for $z \neq v$, reader; $y^e(S) = S'$, where if $z^e(S) = S''$ and $S''(\text{reader}) = (x_1, x_2, \ldots, x_n)$ then $S'(\text{reader}) = (x_2, \ldots, x_n)$, $S'(w) = x_1$, and $S'(z) = S''(z)$ for $z \neq w$, reader

<PRINT statement> x ::= 'PRINT' <variable> y; y ::= <PRINT statement> z ',' <variable> w

<effect> $x^e(S) = S'$, where if $S(\text{printer}) = (x_1, \ldots, x_n)$ then $S'(\text{printer}) = (S(v), x_1, \ldots, x_n)$, $S'(z) = S(z)$ for $z \neq$ printer; $y^e(S) = S'$, where if $z^e(S) = S''$ and $S''(\text{printer}) = (x_1, \ldots, x_n)$ then $S'(\text{printer}) = (S(w), x_1, \ldots, x_n)$, $S'(z) = S''(z)$ for $z \neq$ printer

In practice, of course, records do not correspond to values of integer and real variables; an input record, for example, may be considered as an input card, or a character on such a card. If we take

the individual characters as records, and set aside (say) fifteen characters for each integer or real variable, then we may replace the start of the semantic rule for the effect of a READ statement by

> <effect> $x^e(S) = S'$, where if $S(\underline{reader}) = (x_1, x_2, \ldots, x_{15}, x_{16}, \ldots, x_n)$ then $S'(\underline{reader}) = (x_{16}, \ldots, x_n)$, $S'(v) = cconv(x_1, x_2, \ldots, x_{15})$, and $S'(z) = S(z)$ for $z \neq v$;

and follow this by similar changes in the equation for $y^e(S)$. Here $cconv(x_1, \ldots, x_{15})$ is the integer, real number, etc., resulting from character-code conversion, where the values of the fifteen characters in the string representing that number are $x_1, \ldots, x_{15}$, respectively; thus, for example, in FORTRAN, cconv('3', '.', '0', 'E', '+', '0', '3', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ') = 3000. Further complications may ensue if the number of characters which represents a number is allowed to be variable, or if formatting is allowed; these subjects are taken up in section - .

The definition of the READ statement given above is deficient in one other important respect: it does not indicate the effect when there are no more cards to be read. If $S(\underline{reader}) = (x_1)$ and S' is the new state vector after reading, then $S'(\underline{reader})$ is presumably the null sequence of values; but nothing is said about what happens when $S(\underline{reader})$ is itself the null sequence. Different READ statements handle this case in different ways. Some set an end-of-file flag, which is itself a variable. This variable can then be tested by an "if end of file" statement. Some simply do nothing; thus if $S(\underline{reader})$ were the null sequence, we would have $S' = S$. Finally, some are equipped with a point to which to transfer on end of file. In this case, of course, the exit index of the READ statement will depend upon whether $S(\underline{reader})$ is the null sequence, for the current state vector S.

Let us now consider some statements which act upon a sequential file called ALPHA, which may be implemented as a tape, drum, or disk file, or the like. Let the current length of the file ALPHA be n; we therefore postulate n variables ALPHA[1], ALPHA[2], ..., ALPHA[n], each of which holds a record of the file. As before, we assume that the records correspond to values of variables. There is also a variable pos.ALPHA, the position variable of the file ALPHA, whose values are 0 through n. For brevity, we shall now specify correspondences between typical input-output statements on such a file and assignment and conditional statements which explicitly involve the variables ALPHA[i], $1 \leq i \leq n$, and pos.ALPHA:

OPEN FILE ALPHA	pos.ALPHA = 0
READ ALPHA, X	pos.ALPHA = pos.ALPHA + 1, X = ALPHA[pos.ALPHA]
IF (EOF, ALPHA) THEN ...	IF pos.ALPHA $\geq$ n THEN ...
WRITE ALPHA, X	pos.ALPHA = pos.ALPHA + 1, ALPHA[pos.ALPHA] = X
REWIND ALPHA	pos.ALPHA = 0
BACKSPACE ALPHA	pos.ALPHA = pos.ALPHA - 1
POSITION ALPHA, N	pos.ALPHA = N
POSITION FORWARD ALPHA, N	pos.ALPHA = pos.ALPHA + N
POSITION BACKWARD ALPHA, N	pos.ALPHA = pos.ALPHA - N

It should be easy to see from these correspondences how syntactic and semantic definitions of input-output statements could be constructed, since we already know how to make such definitions of assignment and conditional statements. We shall give two such constructions as examples:

<READ statement> x ::= 'READ' <file name> n ',' <variable> x

<effect> $x^e(S) = S'$, where $S'(n^p) = S(n^p) + 1$, $S'(x) = S(n^m(S(n^p) + 1))$, $S'(z) = S(z)$ for $z \neq n^p$, x

<WRITE statement> x ::= 'WRITE' <file name> n ',' <variable> x

<effect> $x^e(S) = S'$, where $S'(n^p) = S(n^p) + 1$, $S'(n^m(S(n^p) + 1)) = S(x)$, $S'(z) = S(z)$ for $z \neq n^m(S(n^p) + 1)$, x

Here, if n is ALPHA, then n^p is pos.ALPHA and $n^m(k)$ = ALPHA[k], for each $k > 0$. Note that we must write $n^m(S(n^p) + 1)$ (or $n^m(S'(n^p))$), rather than $n^m(S(n^p))$, in each of the above rules, to express the fact that pos.ALPHA is considered as being incremented before it is used.

3-5 Machine Instructions

All of the analysis which we have carried out for commands in algebraic languages has its analogue for instructions on digital computers.

The components of a state vector for a computer are the registers, the memory cells, and the input-output units as suggested in the previous section. The memory cells, of course, include both program and data words. If, for a given program, there exists a constant association between variable names and addresses, there is no difficulty in viewing the α-component of a state vector, for example, where α is an address, as its v-component where v is a variable whose address is α. Program words or instruction words are also state vector components; if their values change during the execution of a program, we refer to that program as self-modifying.

An instruction word in a computer normally has an effect, much like the effect of a statement in an algebraic language. This effect specifies the new state S' of all variables in the computer except for the program counter, given the old state S. It is usually independent of where the instruction word is stored in memory, although there are exceptions, as, for example, in the case of a subroutine-calling instruction (since the return address depends on the instruction location). An instruction word also has an exit address, which is the analogue of the exit index of an algebraic language statement. This gives the value of the program counter after the instruction is executed, as a function of the current state vector, and, except in the case of an unconditional transfer, is dependent also on the address of the instruction word.

Simple instructions on computers often have algebraic equivalents, just as do simple input-output commands. As an example, let us

consider a highly simplified computer with a 16-bit word, and one instruction per word. The right-hand 12 bits in a word are the address field, and the left-hand 4 bits give the operation code, as follows:

Operation code (octal)	Operation	Mnemonic	Algebraic equivalent
00	Halt	H	none
01	Load Y	LD Y	ac = Y
02	Load N	LDI N	ac = N
03	Add Y	AD Y	ac = ac + Y
04	Add N	ADI N	ac = ac + N
05	Subtract Y	SU Y	ac = ac - Y
06	Subtract N	SUI N	ac = ac - N
07	Store Y	ST Y	Y = ac
10	Left shift by N	LS N	$ac = ac*2^N$
11	Right shift by N	RS N	$ac = ac/2^N$
12	Transfer	TR L	GO TO L
13	Transfer on +	TP L	IF $ac \geq 0$ GO TO L
14	Transfer on -	TM L	IF $ac < 0$ GO TO L
15	Increment	IN Y	Y = Y + 1
16	Decrement	DE Y	Y = Y - 1
17	Subroutine call	CA L	none

where ac is the 16-bit accumulator, N is the signed integer in the address field, the letter I in the mnemonics specifies immediate addressing, Y is the contents of the cell whose address is contained in the address field, all shifts are arithmetic (that is, carry out the indicated arithmetic operations properly in all cases), and L is the contents of the address field, taken as representing the address

of the instruction word to which transfer is made. Let us first give an analogue of the syntactic and semantic rules for statements of section 3-3:

<instruction> $\underline{w}$ ::= <assignment instruction> $\underline{a}$; $\underline{x}$ ::= <transfer instruction> $\underline{t}$; $\underline{y}$ ::= <subroutine call instruction> $\underline{c}$; $\underline{z}$ ::= <halt instruction> $\underline{h}$

<statement address>^{d}

$a^d = w^d$; $t^d = x^d$; $c^d = y^d$; $h^d = z^d$

<label function>^{a}

$a^a = w^a$; $t^a = x^a$; $c^a = y^a$; $h^a = z^a$

<effect> $w^e(S) = a^e(S)$; $x^e(S) = t^e(S)$; $y^e(S) = c^e(S)$; $z^e(S) = h^e(S)$

<exit address> $w^x(S) = a^x(S)$; $x^x(S) = t^x(S)$; $y^x(S) = c^x(S)$; $z^x(S) = h^x(S)$

The statement address and the exit address are analogous here to the statement index and the exit index, respectively, in an algebraic language description. This rule may be used regardless of whether assembly language (mnemonics) or machine language (operation codes in octal, binary, etc.) is being described. In assembly language, we might describe the syntax of transfer instructions as

<transfer instruction> ::= <blanks> 'TR' <blanks> <label> | <blanks> 'TP' <blanks> <label> | <blanks> 'TM' <blanks> <label>

<blanks> ::= ' ' | <blanks> ' '

<label> ::= <letter> | <label> <letter> | <label> <digit>

whereas the syntax of transfer instructions in machine language might be given as

<transfer instruction> ::= '1010' <address> | '1011' <address> | '1100' <address>

<address> ::= <bit> <bit> <bit> <bit> <bit> <bit> <bit> <bit> <bit> <bit> <bit> <bit>

<bit> ::= '0' | '1'

A syntactic and semantic rule for transfer instructions in assembly language might be given as

<transfer instruction> x ::= <blanks> 'TR' <blanks> <label> a;
y ::= <blanks> 'TP' <blanks> <label> b;
z ::= <blanks> 'TN' <blanks> <label> c

<statement address>^{d}

<label function>^{a}

<effect> $x^{e}(S) = S$; $y^{e}(S) = S$; $z^{e}(S) = S$

<exit address> $x^{x}(S) \equiv x^{a}(a)$; $y^{x}(S) =$ if $S(ac) \geq 0$ then $y^{a}(b)$ else $y^{d} + 1$; $z^{x}(S) =$ if $S(ac) < 0$ then $z^{a}(c)$ else $z^{d} + 1$

Similar rules may be given for machine language, except that here there would be no label function, and we would write simply a, b, and c instead of $x^{a}(a)$, $y^{a}(b)$, and $z^{a}(c)$. The values of the label function, in either case, are addresses here, of course, rather than statement indices. The constant 1 appearing in the semantic rule for the exit address will be replaced, in a more general case, by the number of words per instruction, which may be fractional; thus, if there are two instructions per word, one of these (presumably the one on the left) has some integer address α, while the other has address $\alpha + \frac{1}{1}$.

It is easy to see the we could save space in our semantic description by changing $x^{e}(S) = t^{e}(S)$, in the rule for the effect of

an instruction, by $x^e(S) = S$ in this case, and eliminating entirely the rule for the effect of a transfer instruction. The possibility of this sort of abbreviation is one reason why we have divided up the sixteen instruction types here into four classes, rather than giving sixteen alternatives in the syntactic rule for an instruction. Assignment instructions may profitably be further classified; we define them as follows:

<assignment instruction> x ::= <direct addressing instruction> d;
y ::= <immediate addressing instruction> i
<statement address>^{d}
<label function>^{a}
$d^a = x^a$; $i^a = y^a$
<effect> $x^e(S) = d^e(S)$; $y^e(S) = i^e(S)$
<exit address> $x^x(S) \equiv x^d + 1$; $y^x(S) \equiv y^d + 1$

Just as we could eliminate the effect in the previous rule, here we could eliminate the exit address (and the statement address) by replacing $a^x(S)$ by $a^d + 1$ in the rule for the exit address of an instruction. Whether we do this or not, however, there is no need for the statement address to be inherited further than this, nor for the exit address to be synthesized at any lower level. Thus the direct and the immediate addressing instructions have only effects and label functions. For the immediate addressing instructions, for example, we might write

<immediate addressing operation code> v ::= 'LDI'; w ::=
'ADI'; x ::= 'SUI'; y ::= 'LS'; z ::= 'RS'
<operation> $v^o(S, n) = n$; $w^o(S, n) = S(\underline{ac}) + n$;
$x^o(S, n) = S(\underline{ac}) - n$; $y^o(S, n) = S(\underline{ac}) * 2^n$;
$z^o(S, n) = S(\underline{ac})/2^n$

<immediate addressing instruction> $\underline{i}$ ::= <blanks> <immediate addressing operation code> $\underline{c}$ <blanks> <address> $\underline{a}$ <label function>^{a}

<effect> $i^e(S) = S'$, where $S'(\underline{ac}) = c^o(S, a^v)$, $S'(z) = S(z)$ for $z \neq \underline{ac}$

where the address a is assumed to have the value a^v. For the direct addressing instructions, assuming that each state vector S has a component $S(a^v)$ for each value a^v of each address $\underline{a}$, we then write

<direct addressing operation code> $\underline{u}$ ::= 'LD'; $\underline{v}$ ::= 'AD'; $\underline{w}$ ::= 'SU'; $\underline{x}$ ::= 'ST'; $\underline{y}$ ::= 'IN'; $\underline{z}$ ::= 'DE'

<operation> $u^o(S, v) = S(v)$; $v^o(S, v) = S(\underline{ac}) + S(v)$; $w^o(S, v) = S(\underline{ac}) - S(v)$; $x^o(S, v) = S(\underline{ac})$; $y^o(S, v) = S(v) + 1$; $z^o(S, v) = S(v) - 1$

<register flag> u^r = true; v^r = true; w^r = true; x^r = false; y^r = false; z^r = false

<direct addressing instruction> $\underline{d}$ ::= <blanks> <direct addressing operation code> $\underline{c}$ <blanks> <address> $\underline{a}$ <label function>^{a}

<destination> d^d = if c^r then ac else a^v

<effect> $d^e(S) = S'$, where $S'(d^d) = c^o(S, a^v)$, $S'(z) = S(z)$ for $z \neq d^d$

The instruction words of most computers, of course, involve more than two fields. Instead of a simple address, as above, we have an address which is subject to index register modification, indirect addressing, relative or paged addressing, and the like. The "destination" given above as a semantic attribute is then replaced by the effective address, which is also a semantic attribute. We might write

<effective address> $i^s(S) = a^v + S(x^r)$

for an instruction i whose address field a has value a^v and whose index register field specification x corresponds to the index register x^r, or

<statement address>^{d}

<effective address> $i^s(S)$ = **if** $m^v = 0$ **then** a^v **else** $(i^d/p)\times p + a^v$

for a scheme resembling that of the PDP-8 (omitting indirect addressing), with page size p, such that page-zero addressing or current-page addressing is used depending on whether the value m^v of the address mode field m of the instruction is zero or one. A single-level indirect addressing scheme with indexing may be described semantically by

<effective address> $i^s(S)$ = **if** $n^v = 0$ **then** y **else** $S(y-(y/2^z)\times 2^z)$, where $y = a^v + S(x^r)$

where z is the number of bits in the address field and n^v is the value of the indirect address field n. Multi-level indirect addressing requires a recursive semantic rule, since it is possible for the calculation of the effective address in this case to continue indefinitely.

NOTES

The notion which we have called the effect of a command -- that is, a function from the old state of the computation into the new state -- is one of the most frequently rediscovered notions in all of programming science. The name "effect" goes back to [McCarthy 63], but the concept is found in [Elgot and Robinson 64] and [Maurer 66] for machine instructions, [Park 68] in connection with data structures, [Cooper 69] for program schemes, and [Scott 70] in a general framework.

A certain amount of our work on the semantics of assignment statements is hinted at in [Strachey 66], [de Bakker 69], [Burstall 70], and [Igarashi 71]. In particular, the concept of the L-value is due to Strachey. In its original form, as noted explicitly in [Park 68], Strachey's model involves a set of addresses, called L-values, and a set of their possible contents, called R-values; the "content of the store" (what we have been calling the current state) is a function from L-values to R-values. Variable names, however, are elements of still another set, and there is a mapping (sometimes called the "environment") from this set to the L-values, and hence, through the content of the store, to the R-values. This model allows storage allocation algorithms, such as those of ALGOL, to be described as changes in the environment. Side effects of both left and right sides of assignments are also treated in [Strachey 66].

The term "statement index" is due to the author. "Statement numbers," which are statement indices in this sense, are treated in [McCarthy 66]; but the term "statement number" has a different meaning in FORTRAN from its meaning in this paper. The use of the exit index as an attribute of certain statements is foreshadowed in [Burstall 70].

The treatment of input-output given here is introduced in essential form in [Maurer 66]. The idea that internal and external variables ought not to be distinguished may appear somewhat startling, since they seem to be so different from each other. The answer to this argument is that an integer variable and a card reader are really no more different from each other than an integer variable and a real variable -- in each case, we have two quantities, each of which varies (and thus has a state vector component), but with quite different sets of values.

It was noted without comment in [McCarthy 63] that state vectors are just as applicable to the description of programs on digital computers as they are for programs in algebraic languages. McCarthy was concerned mainly here with the latter case, whereas [Ershov 60], [Elgot and Robinson 64], [Maurer 66], [Elgot 68], and [Wagner 68], among others, are concerned mainly with digital computers, and construct for them (semi-formally by Ershov and formally by the others) state vectors and functions on them corresponding to computer instructions.

EXERCISES

1. Give formal syntactic rules to correspond to the following informal specifications (suggested by the language BASIC):

(a) An assignment statement has the form LET v=e , where v is a variable and e is an expression.

(b) A conditional transfer statement has the form IF (e) THEN n , where e is an arithmetic expression and n is a line number. (Note: Line numbers in BASIC are much like statement numbers in FORTRAN.)

(c) A subroutine call statement has the form GOSUB P , where P is the name of a subroutine.

(d) A NEXT statement consists of the word NEXT followed by the name of a variable. (This is used to terminate iterations; the variable is the loop index of the iteration.)

2. Rewrite the three syntactic rules given in section 2-1 for assignment statements, left part lists, and left parts in ALGOL in such a way that the rule for left part lists disappears (in other words, there are only assignment statements and left parts) and the new rule for assignment statements is equivalent to the old rule.

3. Suppose that assignments in a simple language are of the form v = p op p , where v is a variable, p is a variable or a number, and op is +, -, *, or /. Give syntactic and semantic rules for such assignments in terms of variables and numbers alone, without introducing "expression" as a nonterminal.

4. How should the first syntactic and semantic rule in section 3-1 be extended to signify that no type conversion is to be permitted (that is, v:=e is not allowed when v and e are of different types)?

5. Consider the following alternative method of specifying the semantics of an IF statement. We employ a special word, next, for semantic purposes. This word is defined to be the exit label of any statement whenever it exits normally (that is, whenever it does not transfer anywhere, but is followed by the next instruction in sequence). The exit label now takes the place of the exit index as a semantic attribute of statements; its value is always either a label (in FORTRAN, a statement number) or the word next.

(a) Would this convention allow us to eliminate any of the other attributes of statements?

(b) Rewrite the semantic rules for FORTRAN IF statements given in section 3-3, using this convention.

(c) Two semantic rules are given in section 3-3 which specify the exit index of an arbitrary assignment statement which always exits normally. What rule or rules would replace these if the above convention were used?

6. Rewrite the semantic rules for the last three nonterminals defined in section 3-3 (simple designational expression, designational expression, and go-to statement) under the assumption that all of these have side effects, as do if clauses and switch designators (but not labels).

7. Modify the definition of the file READ statement at the end of section 3-4:

(a) To test for end-of-file, and do nothing if the file is actually positioned at the end. Assume that the length of the file n is a semantic attribute of it, called n^n.

(b) To set an end-flag on end of file (and do nothing else in this case). Assume that the end-flag of the file n is called n^g.

Also give a syntactic and semantic definition of the OPEN FILE statement in this case, assuming that this statement resets the flag. (The values of the flag should be taken to be true and false.)

(c) To transfer to a given point on end of file. In this case the statement has the form READ (α, END=n), x where α is a file name, n is a label, and x is a variable name. If the file has reached the end, transfer to n; otherwise, exit normally.

8. Give syntactic and semantic definitions of the REWIND, the BACKSPACE, and the POSITION FORWARD statements defined in section 3-4, using the same conventions as are used there for the definitions of the READ and WRITE statements.

9. Give the changes that would have to be made in the syntactic descriptions of

(a) immediate addressing operation codes,

(b) immediate addressing instructions,

(c) direct addressing operation codes,

(d) direct addressing instructions,

defined in section 3-5, so that machine language, rather than assembly language, would be described.

10. Give a syntactic and semantic description of the subroutine call instruction described in section 3-5, assuming that it stores the return address (that is, its own address plus 1) in the address field of L and the operation code for a transfer in the operation code field of L, and then transfers to L+1.

C H A P T E R F O U R

P R O G R A M S

4-1 Program Sections and Labeled Statements

A collection of statements separated by semicolons (or other similar separators) has various common names, such as a compound statement, a block, or simply a program. In order not to conflict with any of the terminology of ALGOL, we shall call this a program section, or simply a section. We have the immediate syntactic rule

<section> ::= <statement> | <section> ';' <statement>

By a statement we here mean a possibly labeled statement; thus we cannot simply use the syntax and semantics of statements from the previous chapter.

In developing the semantics of program sections, let us first consider how label functions are to be calculated. Consider the following ALGOL program with statement indices given:

Statement Index	Statement
1	p := 1;
2	i := 0;
3	u: if i=n then go to v;
4	p := pxa;
5	i := i+1;
6	go to u;
7	v:

This section calculates a^n by successive multiplication, where a is real and i is a positive integer. Syntactically, it consists of a section (namely, the first six statements) followed by the semicolon on the sixth statement, followed by the dummy statement v: .

The label function f which is to be associated with this section satisfies f(u) = 3 and f(v) = 7. This label function must then be inherited by each statement of the section; for example, in order to calculate the exit index of the sixth statement, we must know that the statement index of the statement labeled u is 3. We now make use of another construction from elementary set theory: the definition of a function as a set of ordered pairs. Let f: A → B be any function, and consider the set of all constructions of the form (x, f(x)), for all $x \in A$. Each (x, f(x)) is referred to as an ordered pair, and the set of all these ordered pairs clearly specifies f completely and may be taken as a definition of f. In this way, we may build the theory of sets on the primitive notions of set and ordered pair, rather than set and function. (The alternative would be to define an ordered pair as a function whose domain is {1, 2}; for the ordered pair f = (u, v), we would have f(1) = u and f(2) = v.)

The advantage which we derive from considering a function as a set of something arises from the operation of taking the union of two sets. In our example, the label function f becomes the set

$$\{(u, 3), (v, 7)\}$$

Let us now define the local label function of a statement, or of a section, as consisting of those ordered pairs (x, f(x)) for which x is a label of the given statement, or of some statement in the given section. Thus the local label function of the section consisting of the first six statements in our example is the set containing only the ordered pair (u, 3); the local label function of the seventh

statement contains only the pair (v, 7). The local label function of the entire section, as a set of ordered pairs, is the union of these two sets; thus we may write

<section> x ::= <statement> s; y ::= <section> z ';'
<statement> t
<local label function> $x^b = s^b$; $y^b = z^b \cup t^b$

Both z^b and t^b are sets of ordered pairs; in particular, t^b may contain more than one ordered pair, since a single ALGOL statement may have more than one label. Even in FORTRAN, where this is not allowed, t^b is still a set of either one ordered pair or none. The local label function of a statement is then defined by

<statement> x ::= <unlabelled statement> u; y ::= <label> a ':'
<statement> z
<statement index>^{i}
$u^i = x^i$; $z^i = y^i$
<local label function> $u^b = \phi$; $y^b = \{(a, y^i)\} \cup z^b$

where the statement index must be inherited from the section in which the statement is contained. This process is particularly simple: we define the number of statements in a section as a semantic attribute of it, and then the statement index of the last statement in any section is equal to the number of statements in that section. The semantic rules

<number of statements> $x^n = 1$; $y^n = z^n + 1$
$s^i = 1$; $t^i = y^n$

may be added to the definition of a section given above to perform this calculation.

The ordinary label function is obtained from the local label

function of any section which appears as part of a higher-level rule such as

<block> ::= 'begin' <declarations> <section> 'end'

or

<compound statement> ::= 'begin' <section> 'end'

Taking the second of these as an example, we may define its label function as the local label function of the section which it contains; this section, then inherits that label function, as follows:

<compound statement> c ::= 'begin' <section> s 'end'
<label function> $c^a = s^b$
$s^a = c^a$

By slightly extending the notion of an inherited attribute, we may shorten this definition by writing

<compound statement> c ::= 'begin' <section> s 'end'
$s^a = s^b$

Here the family-tree analogy breaks down; the section s "inherits" its label function from itself! We will continue to use such rules, however, and to refer to them as inherited attribute equations whenever the quantity on the left is an attribute of a quantity on the right of the syntactic rule. Note that only sections which are directly part of compound statements and the like have their label functions determined in this way. We certainly do not, for example, want the label function of the section consisting of the first six statements illustrated at the start of this section to contain only the pair (u, 3); it must contain the pair (v, 7) as well.

There is another sort of semantic rule which it is important to associate with compound statements and the like. Sets of ordered

pairs can represent multi-valued, as well as single-valued, functions. We would like the label function of a section to be single-valued; in fact, it will be single-valued if and only if there are no duplicate labels. A set S of ordered pairs represents a single-valued function if and only if the statement

$$(x, y) \in S \text{ and } (x, z) \in S \text{ implies } y = z$$

is true. We may add the semantic rule

(s^b is single-valued)

to the definition of compound statements above. We might also add the semantic rule

(y^b is single-valued)

to the definition of a statement; this would check for multiple labels on a single statement, which is perhaps unnecessary since there is no real ambiguity here. We do not, however, need to add any further test of this kind to the definition of a section, since such a test will always succeed if the corresponding test for compound statements succeeds.

The device of treating functions as sets of ordered pairs has many further uses in semantic rules. In FORTRAN, for example, variables need not be declared unless they are logical, complex, or double precision, or unless their names violate the "I, J, K, L, M, or N rule." If we wish all state vectors applicable to a given FORTRAN program to involve the set of all variables of that program, we must therefore associate with each statement a set of ordered pairs (vn, type), or its equivalent, where vn is a variable name used in the given statement and type is its type according to the "I, J,

K, L, M, or N rule." This may then be treated as a function from variable names to types, and the sets of ordered pairs may be combined just as the pairs for the label function are. We shall see in Chapter 5 that most of the attributes of declarations are of this kind; that is, they are functions of variable names, given by sets of ordered pairs, which give their types, array dimensions, and other such information.

4-2 The Effect of a Section

What other semantic attributes do we want a program section to have? For a section which is a straight-line program -- that is, one in which control always passes from each statement to the next, without branching -- each individual statement has an effect which transforms the current state vector, and by stringing these all together we obtain an effect for the entire section. This may be denoted by the rule

<section> x ::= <statement> s; y ::= <section> z ';'
<statement> t
<effect> $x^e(S) = s^e(S)$; $y^e(S) = t^e(z^e(S))$

assuming that s^e and t^e are the respective effects of the sections s and t. In terms of still another elementary function-theoretical notion, y is the composition of t and z, and we may write $y = t \circ z$. This notation is somewhat confusing in a programming context because the effect of y is the effect of z followed by the effect of t, not the other way around; thus we would like to write $y = z \circ t$, rather than $t \circ z$. This can be done if we define $f \circ g$, in general, as the function h satisfying $h(x) = g(f(x))$ (rather than $f(g(x))$). In what follows, we shall not use either convention, but shall continue to write semantic rules without using the composition operator.

Does the concept of effect make sense for a section which is not a straight-line program? Suppose that y is a section and S is a state vector. Suppose further that y is "self-contained"; that is, every label which is referenced in the section is also defined in the section. Let us consider the section as a program and run it; when it reaches the end, there will be some current state vector S', and we set $y^e(S) = S'$ where y^e is the effect which we are defining for y.

If the program never reaches the end -- that is, if it goes into an endless loop -- we leave $y^e(S)$ undefined. That is, the effect of y is, in general, a partial function -- a function which is undefined for certain elements of its domain. We recall from section 2-2 that we may write $f: X \to Y$ even when the range of f is not Y, but merely a subset of Y. If the domain of f is not X, but merely a subset of X, then f is called a partial function; if the domain of f is X, then f is a total function.

How can we determine the effect of a section from the effects of its individual statements? We start by defining the single-step function of a program section. Most computers have a "single-step button" or its equivalent; if the computer is stopped and the single-step button is pushed, a single instruction will be executed, namely the instruction whose code is in the memory word (or starts at the memory byte) whose address is currently in the program counter. At the same time, the program counter is modified in the way indicated by the instruction which is executed, so that, by pushing the button over and over, the operator can "step through" a program. The single-step function of a program section is set up in an analogous fashion. It is a function of one argument, and that argument is a pair of the form (S, k), where S is a state vector and k is a statement index; and its value is the next such pair, in execution order.

Any pair of the form (S, k) may be identified with a single state vector T having all the components that S does, plus a new component corresponding to a program-counter variable λ whose values are statement indices. Recalling the discussion at the end of section 2-2, let M be any set of variables and let $M' = M \cup \{\lambda\}$, where $\lambda \notin M$. If $\mathcal{S} = \prod_{x \in M} V_x$ and $\mathcal{S}' = \prod_{x \in M'} V_x$, where V_x is the set of all legal values of the variable x, then there is a natural correspondence between $\mathcal{S}'$

and $\mathcal{L} \times V_\lambda$. Specifically, if $S \in \mathcal{L}$ and $k \in V_\lambda$, then the pair (S, k) defines an element $S' \in \mathcal{L}'$ whose λ-component is k and whose other components are the same as the corresponding ones of S. If λ is a program counter, then V_λ is a set of indices (or addresses) of statements of the given program.

A semantic rule for single-step functions may be given as follows:

<section> $\underline{x}$ ::= <statement> $\underline{s}$; $\underline{y}$ ::= <section> $\underline{z}$ ';'
<statement> $\underline{t}$

<number of statements> $x^n = 1$; $y^n = z^n + 1$

$s^i = 1$; $t^i = y^n$

<single-step function> $x^p((S, 1)) = (s^e(S), s^x(S))$;

$y^p((S, k)) = z^p((S, k))$ for $1 \leq k \leq z^n$,

$y^p((S, y^n)) = (t^e(S), t^x(S))$

Note that by writing $y^p((S, k))$ rather than $y^p(S, k)$, we define a function of one argument (which is a pair), rather than two.

Now we are ready to define the generalized effect of a program section, from which we will easily obtain the effect. The generalized effect has the same form as the single-step function: it has a single argument, which is a pair (S, k), and its value is also a pair (S', k'). If the section is started at the kth statement, and the starting state vector is S, then, when the program terminates, the final state vector will be S' and the final statement index will be k'. For programs using the statements we have defined thus far, k' will always be one greater than the number of statements in the program section (but see also the discussion in section 4-3). The generalized effect g may be defined in terms of the single-step function p, and the total number of statements n, by the equation

$$g((S, k)) = \underline{\text{if}}\ 1 \leq k \leq n\ \underline{\text{then}}\ g(p((S, k)))\ \underline{\text{else}}\ (S, k)$$

This is our first use of a recursive conditional expression; g is defined by expressing g(z), for a given z, in terms of g(z'), where z' depends on z. Intuitively, the result g((S, k)) when the program is started at the k-th statement, with current state vector S, is the same as the result g(p((S, k))) when the program is started at the statement with index k_1, with current state vector S_1, where $(S_1, k_1) = p((S, k))$ is the result of the first step of the program when it is started at statement k with state vector S.

Recursive conditional expressions are the only expressions encountered so far which give a partial function as their result. If the given program section is an endless loop, or if, more generally, it runs endlessly when it is started at certain pairs (S, k) as above, then the recursive conditional expression becomes a circular expression -- it cannot itself be evaluated in a finite number of steps. More importantly, a recursive conditional expression does not, in itself, lead to any method of determining when it is properly defined and when it is not. Thus, when we use a semantic definition of a programming language which involves such expressions, we can always determine the effect of a program, so long as we know that it actually has an effect, but otherwise we cannot. Intuitively, this is quite reasonable. It is easy, for example, to write a simple program to search for counterexamples to Fermat's last theorem, or any number of other unsolved problems in mathematics; we should not expect to be able to determine whether such a program always loops endlessly by means of simple algorithmic arguments. Moreover, by the elementary theory of recursive functions, there are certain problems that are not only unsolved but unsolvable; and one of these is the problem of determining, for an arbitrary recursive conditional expression, whether it can be evaluated in a finite number of steps.

Are there any programming languages in which the effect of a

program may always be calculated without encountering any unsolvable problems? For straight-line programs, this can always be done; but straight-line programs are not very general. A much more important class of programs of this type will be introduced in Chapter 6: those programs for which it is possible to construct a proof of termination. The importance of this class of programs arises from the fact that, while in theory a programming language may be used to write programs for which it is not known whether they terminate, in practice this is almost never done intentionally. Even when it is done intentionally, the resulting program is often run under the control of an operating system which stops it automatically after a certain maximum time, so that, whatever happens, we are only interested in the result of a finite sequence of operations. By restricting our analysis to programs which are known to terminate, we are able to avoid the general problem of the existence of unsolvable questions.

Let us give a semantic rule for the generalized effect of a compound statement:

<compound statement> c ::= 'begin' <section> s 'end'

<generalized effect> $c^g((S, k)) = \underline{\text{if}}\ 1 \leq k \leq s^n\ \underline{\text{then}}\ c^g(s^p((S, k)))\ \underline{\text{else}}\ (S, k)$

<effect> $c^e(S) = S'$, where $(S', k') = c^g((S, 1))$ for some k'

Here s^p is the single-step function of s, while s^n is the number of statements of s; although in this case we will always have $k' = s^n + 1$, that fact is not made use of. Notice that we have also assumed that the program always starts at its first statement. This assumption is tenable in ALGOL, but not in certain other languages, which have ENTRY statements or the equivalent; an ENTRY statement (actually a declaration, rather than an executable statement) defines a label of a program to be an allowable starting point. In such a case, we

may let such a declaration inherit the generalized effect of the program in which it is contained; if the label function is also inherited, we may write (for example)

<entry declaration> x ::= 'ENTRY' <label> a; y ::= <entry declaration> z ',' <label> b

<generalized effect>^{g}

$z^g = y^g$

<label function>^{a}

<name function> $x^m = \{(a, f)\}$ where $f(S) = S'$ for $(S', k') = x^g((S, x^a(a)))$; $y^m = z^m \cup \{(b, f)\}$ where $f(S) = S'$ for $(S', k') = y^g((S, y^a(b)))$

The name function then associates names of programs with their corresponding effects; that is, if a is a label of a function appearing in an entry declaration, and m is the name function, then m(a) is the effect of the given program when started at the label a. Such functions and their generalizations are further studied in section 5-4.

4-3 Multiple Exits and Escape

It is very common for a program section to have more than one possible exit. In this case, we may determine an exit index for the section, in much the same way as we have determined the effect. Given the starting state vector S, the value of the exit index $c^x(S)$ of the section c is the index of the statement to which transfer is ultimately made after the given section is finished. Thus, in particular, a section has an effect and an exit index — just as an individual statement does. In fact, the appearance of a compound statement or a block, or the calling of a subroutine, may be regarded exactly as if it were the execution of a single statement whose effect and exit index are those of that compound statement, block, or subroutine.

The semantic rule for the effect of a compound statement as given in section 4-2, namely

<effect> $c^e(S) = S'$, where $(S', k') = c^g((S, 1))$
for some k'

needs only a very slight modification to produce a semantic rule for the exit index of a compound statement:

<exit index> $c^x(S) = k'$, where $(S', k') = c^g((S, 1))$
for some S'

Similar constructions may be used where entry declarations are allowed.

Let us now see how the phenomenon of multiple exits affects the semantics of a block structure language. We shall assume that every statement in a program, including those in inner blocks, has a distinct statement index, and that these are numbered from the

beginning of the program. In an if statement of the form if B then S, the if clause (if B then) will be considered as a statement, and S will be considered as another statement, or as several statements if it is a compound statement or a block. The meaning of "if B then" is taken to be "if not B, then skip around S." Here is an example of a program with statement indices specified:

```
STATEMENT INDEX   STATEMENT
                  integer i, j, m, n; Boolean match; real array A[1:n], B[1:m];
      1           match:=true;
      2           i:=1;
      3           outer: if A[i] ≠ 0 then
      4           begin j:=1;
      5           inner: if A[i]=B[j] then
      6           go to found;
      7           j:=j+1;
      8           if j ≤ m then
      9           go to inner end;
     10           i:=i+1;
     11           if i ≤ n then
     12           go to outer;
     13           match:=false;
     14           found:
```

This routine exits with match = true if A[i]=B[j]≠0 for any i and j, $1 \leq i \leq n$, $1 \leq j \leq m$. Statements 4 through 9 constitute an ALGOL compound statement with two exits; one to statement 10 (the normal exit) and one to statement 14.

Recalling the syntactic and semantic definition of logical IF statements in FORTRAN which was given at the beginning of section 3-3, we may set up a similar definition for if statements in ALGOL:

<if statement> i ::= <if clause> c <unconditional statement> u

<effect> $i^e(S) = \underline{\text{if}}\ c^v(S)\ \underline{\text{then}}\ u^e(c^s(S))\ \underline{\text{else}}\ c^s(S)$

<exit index> $i^x(S) = \underline{\text{if}}\ c^v(S)\ \underline{\text{then}}\ u^x(c^s(S))\ \underline{\text{else}}\ c^x(S)$

Here c^v and c^s are respectively the value and the side effect of the if clause c, that is, of the Boolean expression which it contains; u^e and u^x are respectively the effect and the exit function of u. We can now see what happens when u is a compound statement or a block. For example, in statement 3 above, u is the entire compound statement including statements 4 through 9, and its effect and exit function are taken in the sense of the preceding discussion. If S is any state vector, then $u^x(S) = 14$ -- that is, the compound statement u exits to "found" -- if there exists j, $1 \leq j \leq m$, such that S indicates A[i] = B[j], that is, S(A[S(i)]) = S(B[S(j)]); and $u^x(S) = 10$ otherwise. Since the if clause $c = \{A[i] \neq 0\}$ has no side effect, that is, $c^s(S) = S$, the exit index i^x of the entire if statement is therefore such that if $S(A[S(i)]) \neq 0$ then $i^x(S) = u^x(S)$ as above, while otherwise $i^x(S) = 10$, that is, the exit index of the if clause, which is the index of the statement immediately following the if statement.

In order to bring this about, we shall have to associate with each compound statement or block in an ALGOL program, including the outermost block, a <u>set</u> of statement indices of statements in that block. Thus, in our example, the outermost block involves statements 1, 2, 3, 10, 11, 12, 13, and 14, plus the compound statement whose index is 4 and which contains statements 4, 5, 6, 7, 8, and 9. The computation performed by a block is taken as ending when any transfer is made to a statement outside that block. Thus the rule for compound statements at the end of section 4-2 is replaced by

<compound statement> c ::= 'begin' <section> s 'end'

<statement index>^{i}

$s^i = c^i$

<number of statements> $c^n = s^n$

<generalized effect> $c^g((S, k)) =$ if $c^i \leq k \leq c^i + s^n$ then $c^g(s^p((S, k)))$ else (S, k)

<effect> $c^e(S) = S'$, where $(S', k') = c^g((S, c^i))$ for some k'

<exit index> $c^x(S) = k'$, where $(S', k') = c^g((S, c^i))$ for some S'

where a section is now defined by

<section> x ::= <statement> s; y ::= <section> z ';' <statement> t

<number of statements> $x^n = s^n$; $y^n = z^n + t^n$

<statement index>^{i}

$s^i = x^i$; $z^i = y^i$; $t^i = y^i + z^n$

<single-step function> $x^p((S, 1)) = (s^e(S), s^x(S))$;

$y^p((S, k)) = z^p((S, k))$ for $1 \leq k \leq z^n$;

$y^p((S, y^n)) = (t^e(S), t^x(S))$

Since, in this rule, a "statement" may itself be compound, it has, as an attribute, a "number of statements," with superscript n. This is the number of statements which it contains, if it is a compound statement or block; it is 1 for an assignment statement; and it is one more than the number of statements in S, for a statement of the type if B then S.

When an ALGOL program contains a procedure used as a function, the procedure may or may not exit normally. Consider the following program with statement indices specified:

```
STATEMENT INDEX     STATEMENT
                    integer j, k; integer procedure g(i); integer i;
        1           begin j:=j+10;
        2           g:=7;
        3           if i=2 then
        4           go to m
        5           end g;
        6           j:=3;
        7           k:=g(1)+g(2)+g(3);
        8           go to done;
        9           m: outreal(float(j));
       10           done:
```

This program section starts at statement number 6. Since the value of g(i) is always 7, g(1) + g(2) + g(3) would seem to be equal to 21; but in fact the evaluation of this expression is never completed because we transfer to m when i=2, that is, when evaluating g(2). Thus the output function outreal(float(j)) is performed, and the number 23.0 is printed out, because j is initialized to 3 and is incremented twice, once in evaluating g(1) and once in evaluating g(2).

The procedure g has two exits; one to statement 5 (the normal exit) and one to statement 9. We say that g escapes if it jumps to statement 9. Thus g will have three associated state vector functions: a value, which in this case is always 7 (and is independent of the state vector); an effect, which here is always to increase j by 10 (that is, it is the effect of the statement j:=j+10); and an exit index. If S(i) = 2, for the state vector S, then x(S) = 9, where x is the exit index; if S(i) ≠ 2, so that g exits normally, we will set x(S) equal to a special keyword, normal, to denote this fact. The effect and the exit index of g, of course, are determined in the man-

ner described at the beginning of this section.

The possibility of escape affects all of the semantic rules for assignments, expressions, terms, factors, and primary expressions. Consider, for example, the simplified rule for expressions with side effects given in section 2-3:

<expression> e ::= <term> y '+' <term> w
<side effect> $e^S(S) = w^S(v^S(S))$
<value> $e^V(S) = v^V(S) + w^V(v^S(S))$

If escape is allowed, this rule becomes

<expression> e ::= <term> y '+' <term> w
<exit index> $e^X(S) =$ if $v^X(S) \neq$ normal then $v^X(S)$ else $w^X(v^S(S))$
<side effect> $e^S(S) =$ if $v^X(S) \neq$ normal then $v^S(S)$ else $w^S(v^S(S))$
<value> $e^V(S) =$ if $e^X(S) \neq$ normal then none else $v^V(S) + w^V(v^S(S))$

Here the superscripts x, s, and v are presumed to have the same meanings for terms as for expressions. If y escapes, then the side effect of y+w is simply the side effect of y, assuming that evaluation takes place from left to right. Otherwise, whether w escapes or not, it is the result of combining the side effects of y and w. Of course, if y escapes, then y+w is said to escape to the same place; otherwise y+w escapes only if w does. Here by "y escapes" we mean "the exit index of y is not normal." It should also be clear that y+w has a value if and only if it does not escape.

Similar changes may be made in other rules for expressions, and in rules for terms and factors. For a primary expression, we must determine its value, side effect, and exit index. Just as the side ef-

fect of most kinds of primary expression is the identity -- that is, e(S) = S, where e is the side effect -- so the exit index of most kinds of primary expression is always normal. In ALGOL, the only cases where it can be anything other than normal are function references, as g(2) in the example above; expressions enclosed in parentheses; and parameters called by name, to be discussed in section 5-4.

Again, consider the rule for assignments with side effects given in section 3-2:

<assignment> a ::= <variable> v ':=' <expression> e
    <effect> $a^e(S) = S'$, where $S'(v) = e^v(S)$, $S'(z) =$
        $S''(z)$ for $z \neq v$, where $S'' = e^s(S)$

In the presence of expressions which may escape, this becomes

<assignment> a ::= <variable> v ':=' <expression> e
    <effect> $a^e(S) =$ if $e^x(S) \neq$ normal then $e^s(S)$ else
        S', where $S'(v) = e^v(S)$, $S'(z) = S''(z)$ for
        $z \neq v$, where $S'' = e^s(S)$

If e escapes, the effect of the assignment is precisely the side effect of e; the variable v is not changed (unless it is changed by that side effect). Later in section 3-2, a more general rule is given, assuming that the variable v has an L-value v^l and a side effect v^s of its own, and that evaluation is as in ALGOL:

<assignment> a ::= <variable> v ':=' <expression> e
    <effect> $a^e(S) = S'$, where $S'(v^l(S)) = e^v(v^s(S))$, $S'(z)$
        $= S''(z)$ for $z \neq v^l(S)$, where $S'' = e^s(v^s(S))$

Assuming that the left side or the right side, or both, might possibly escape, this rule becomes

<assignment> a ::= <variable> v ':=' <expression> e

<effect> $a^e(S)$ = **if** $v^x(S) \neq$ normal **then** $v^s(S)$ **else if** $e^x(v^s(S)) \neq$ normal **then** $e^s(v^s(S))$ **else** S', where $S'(v^l(S)) = e^v(v^s(S))$, $S'(z) = S''(z)$ for $z \neq v^l(S)$, where $S'' = e^s(v^s(S))$

<exit index> $a^x(S)$ = **if** $v^x(S) \neq$ normal **then** $v^x(S)$ **else if** $e^x(v^s(S)) \neq$ normal **then** $e^x(v^s(S))$ **else** $a^n + 1$

<statement index>^{n}

Allowance for the possibility of escape must also be made, for example, in evaluation of subscripts. If a[e, e'] is an array reference, where e and e' are expressions, then e is normally evaluated first, followed by e'. If e escapes, then so does a[e, e']; if e does not escape, then e' may escape.

4-4 Iteration Statements

The original purpose of iteration statements (DO in FORTRAN and PL/I, for in ALGOL, PERFORM in COBOL, etc.) was to repeat a statement, or a group of statements, a certain number of times. If the number of iterations is constant, and the statement always exits normally, the effect of such an iteration statement is easy to define. For example, if A is a simple assignment statement with effect e, then the effect f of (for i:=1 step 1 until 3 do A) satisfies f(S) = e(e(e(S))) for each state vector S, provided that i is not referenced in the assignment A.

If A is a general command, with an effect e and an exit index x, we must distinguish normal exit of A from abnormal exit. Consider the following sequence:

```
a: k:=g(i);
m: n:=j;
```

where the function g may escape to m, as in the preceding section. It is clear that the exit index of statement a is constant, since it always exits to statement m. However, if we precede these two statements by

```
for i:=1 step 1 until 3 do
```

then we must distinguish normal exit to m and escape to m through g(i), because in the former case iteration proceeds, while in the latter case it does not. Let us do this by defining x(S) = normal whenever a exits normally, where x is the exit index of the statement a. Then the effect e' and the exit index x' of (for i:=1 step 1 until 3 do A) may be defined by

$$e'(S) = \underline{\text{if}}\ x(S) \neq \text{normal}\ \underline{\text{then}}\ e(S)\ \underline{\text{else if}}\ x(e(S)) \neq \text{normal}\ \underline{\text{then}}\ e(e(S))\ \underline{\text{else}}\ e(e(e(S)))$$

$$x^e(S) = \underline{\text{if}}\ x(S) \neq \underline{\text{normal}}\ \underline{\text{then}}\ x(S)\ \underline{\text{else if}}\ x(e(S)) \neq \underline{\text{normal}}\ \underline{\text{then}}\ x(e(S))\ \underline{\text{else}}\ x(e(e(S)))$$

where e and x, respectively, are the effect and the exit index of A. As before, it is assumed that A makes no reference to i.

Under this last condition, we may define the effect p^e and the exit index p^x of the more general iteration

$$p = (\underline{\text{for}}\ i := \alpha\ \underline{\text{step}}\ \beta\ \underline{\text{until}}\ \gamma\ \underline{\text{do}}\ A)$$

where α, β, and γ are integer constants and $\beta > 0$, by a recursive equation as follows. Let e and x be as before, and assume that x(S) = normal when A exits normally. Let

$$q = (\underline{\text{for}}\ i := \alpha+\beta\ \underline{\text{step}}\ \beta\ \underline{\text{until}}\ \gamma\ \underline{\text{do}}\ A)$$

and let the effect and the exit index of q be q^e and q^x respectively. We assume the same conditions on p^x and q^x as we do on x; that is, for example, $q^x(S)$ = normal whenever q exits normally. The semantics of p may now be defined in terms of the semantics of q (and A) as follows:

$$p^e(S) = \underline{\text{if}}\ \alpha > \gamma\ \underline{\text{then}}\ S\ \underline{\text{else if}}\ x(S) \neq \underline{\text{normal}}\ \underline{\text{then}}\ e(S)\ \underline{\text{else}}\ q^e(e(S))$$

$$p^x(S) = \underline{\text{if}}\ \alpha > \gamma\ \underline{\text{then}}\ \underline{\text{normal}}\ \underline{\text{else if}}\ x(S) \neq \underline{\text{normal}}\ \underline{\text{then}}\ x(S)\ \underline{\text{else}}\ q^x(e(S))$$

These rules do not work if $\beta < 0$; we must replace $\alpha > \gamma$ by $\alpha < \gamma$. In fact, for general β, we may write

$$p^e(S) = \underline{\text{if}}\ \beta(\gamma - \alpha) < 0\ \underline{\text{then}}\ S\ \underline{\text{else if}}\ x(S) \neq \underline{\text{normal}}\ \underline{\text{then}}\ e(S)\ \underline{\text{else}}\ q^e(e(S))$$

$$p^x(S) = \text{if}\ \beta(\gamma - \alpha) < 0\ \text{then}\ \underline{\text{normal}}\ \text{else if}\ x(S) \neq \underline{\text{normal}}\ \underline{\text{then}}\ x(S)\ \underline{\text{else}}\ q^x(e(S))$$

since the general condition for terminating the iteration is that $\gamma - \alpha$ and β have opposite signs (and therefore their product is negative). Clearly, if $\beta(\gamma - \gamma) = 0$ -- that is, $\beta = 0$ or $\alpha = \gamma$ -- iteration should continue. The slightly cumbersome property of ALGOL-style iteration illustrated above has given rise to separate statements, in the language PASCAL, for iterating with $\beta > 0$ and with $\beta < 0$. (FORTRAN simply requires $\beta > 0$ at all times.)

What happens if the statement A does, in fact, make reference to i? This will happen, in particular, in any loop which traverses a subscripted array. One solution is to replace S, throughout the right sides of the rules given above, by e'(S), where e' is the effect of the assignment i:=α (which always exits normally, since α is still being regarded as an integer constant). Difficulties arise, however, if i may be changed by A. What is the effect of the iteration

for i:=1 step 1 until 3 do i:=i-1

Does it iterate three times, or does it keep going indefinitely? According to the above definition, it iterates three times; but most compilers for existing languages would not implement this specification. It is possible, of course, to make such behavior illegal, and, in fact, although almost all iterated statements in actual programs make reference to the loop index, or controlled variable (i in this case), very few of them change the loop index.

Let us now consider what would be involved in casting the above equations in the form of semantic rules. If we wished to define the effect and the exit index of p, as above, we would first have to define the source string q as an attribute of p. This introduces a new level of complexity into our definitions; and we still have not relaxed the condition that α, β, and γ are constants. In the ALGOL re-

port, it is suggested that the best way to define an iteration statement is to regard it as a collection of statements which explicitly initialize, increment, and test the loop index. In our terminology, we should set up a program section for each element of a for-list, and then define the effect and the exit index of an iteration as the effect and the exit index, respectively, of the corresponding section. We shall now indicate how this may be done for ALGOL.

Our first task is to make the effect and the exit index of the statement to be iterated, and also the controlled variable, into inherited attributes of each element of the for-list. By doing this we may calculate an effect and an exit index for each such element, which is effectively the result of iterating only as far as that element specifies. The effect and the exit index of the entire for statement are then determined from those of the for-list elements by effectively taking these in sequence. Our syntactic and semantic rules for for-lists are thus

<for list> X ::= <for list element> U; Y ::= <for list> Z ',' <for list element> Y

<controlled variable>^{c}

$u^{c} = x^{c}$; $z^{c} = y^{c}$; $v^{c} = y^{c}$

<iterated effect>^{f}

$u^{f} = x^{f}$; $z^{f} = y^{f}$; $v^{f} = y^{f}$

<iterated exit index>^{n}

$u^{n} = x^{n}$; $z^{n} = y^{n}$; $v^{n} = y^{n}$

<effect> $x^{e}(S) = u^{e}(S)$; $y^{e}(S) =$ if $z^{x}(S) \neq$ normal then $z^{e}(S)$ else $v^{e}(z^{e}(S))$

<exit index> $x^{x}(S) = u^{x}(S)$; $y^{x}(S) =$ if $z^{x}(S) \neq$ normal then $z^{x}(S)$ else $v^{x}(z^{e}(S))$

We are here assuming that the controlled variable is a simple variable, and is not a parameter called by name; otherwise we would have to inherit its effect and exit index as well. The semantics of a for-statement may then be given by

<for statement> f ::= 'for' <variable> v ':=' <for list> i 'do' <statement> s

$i^c = v$

$i^f = s^e$

$i^n = s^x$

<effect> $f^e(S) = i^e(S)$

<exit index> $f^x(S) = i^x(S)$

The syntactic rule here ignores labelled for-statements, and defines a for-statement directly in terms of a for-list, without the intermediate nonterminal <for clause> as given in the ALGOL report.

It remains to specify the effect and exit index of each type of for-list element. For a simple arithmetic expression, no program section needs to be defined; the effect of this kind of element is simply the effect of assigning this expression to the controlled variable, followed by the effect of the iterated statement, where this intuitive description must be modified in the usual way to account for possible escape. If this were the only type of for-list element, we could write

<for list element> x ::= <arithmetic expression> c

<controlled variable>^{c}

<iterated effect>^{f}

<iterated exit index>^{n}

<effect> $x^e(S) =$ *if* $c^x(S) \neq$ normal *then* $c^s(S)$ *else* $x^f(S')$,

where $S'(x^c) = c^v(S)$, $S'(z) = S''(z)$ for $z \neq x^c$,

with $S'' = c^s(S)$

<exit index> $x^X(S) = c^X(S)$

Here we are assuming that the expression c has the value c^V, the side effect c^S, and the exit index c^X, and that no type conversion takes place (the type conversion rules here would be exactly the same as those for the assignment statements).

For each of the other two kinds of for-list element, we construct a program section as directed in the ALGOL report itself. For an element of the form A step B until C, where S is the statement to be iterated and V is the controlled variable, the ALGOL report gives this section as

```
    V := A;
L1: if (V-C)×sign(B)>0 then go to element exhausted;
    statement S;
    V := V + B;
    go to L1;
```

whereas for an element of the form E while F, the section is given as

```
L3: V := E;
    if ¬F then go to element exhausted;
    statement S;
    go to L3;
```

Here ¬F, of course, means "not F."

Our semantic rules for elements of a for-list are now constructed as follows. First we define the effect and the exit index of each of the statements above, in accordance with the rules for assignments, go-to statements, and if statements. We then define a single-step function (see section 4-2) applicable to pairs (S, i), where $1 \leq i \leq 4$. (We may regard V := V + B and go to L1 in the first

section above as a single statement, so that each of these sections consists of four statements.) The effect and the exit index of the for-list element are then defined by a recursive rule which makes reference to this single-step function, as is done at the end of section 4-2.

4-5 Multiple Use of Names

The concept of block structure in programming languages, in which each block has its own declarations, has given rise to the concept of several variables with the same name, each associated with a different block level. In modeling this situation by a collection of state vector functions, a problem arises as to the proper specification of the domain of a state vector. Must our state vectors have components for different variables with the same name? If so, we must find a way to identify such variables uniquely.

There are several different solutions to this problem. In the first place, there are certain situations in which it is not, in fact, necessary ever to consider state vectors with more than one α-component, for a given variable name α. In particular, this will be true in the absence of procedures. Consider a block structure language in which variable names may be re-used inside each block, but there are no procedure calls. In this case, there is within each block a unique variable with a given name which may be referenced or changed. Thus the state vectors in an innermost block have only one α-component for each α, and an innermost block has an effect and an exit index which involve those state vectors. We may now construct from these an effect and an exit index which may be used at the next lower block level. This new effect and exit index involve state vectors which are valid at that block level; that is, the α-component, for each α, references that variable named α whose scope includes that block level, even though α may have been redefined at the inner block level. The inner block is now treated as a single statement with the new effect and exit index, and, continuing inductively, we see that the state vectors in <u>any</u> block of the program may be taken to have only one α-component for each α.

In order to derive the effect e' and the exit index x' at the outer block level from the effect e and the exit index x at the inner block level, we go through a two-stage process. Let M' be the set of all variables whose scope includes the outer block level (defined at that level or at further outer levels). Let M^* be the set of all variables defined at the inner level; then $M = M' \cup M^*$ is the set of all variables whose scope includes the inner level. Let $\mathcal{S} = \prod_{x \in M} V_x$ and let $\mathcal{S}' = \prod_{x \in M'} V_x$; then $\mathcal{S}$ is the domain of e and of x, while $\mathcal{S}'$ is the domain of e' and of x'. The first step is to eliminate the variables defined at the inner level. Let $M'' = M' - M^*$; that is, M'' consists of all those variables whose scope includes the outer level and which were not redefined at the inner level. We then wish to construct e'' and x'' with domain $\mathcal{S}'' = \prod_{x \in M''} V_x$. For each $S \in \mathcal{S}$, we may form its restriction to M'', or $S|M''$; this is an element of $\mathcal{S}''$. Our definitions of e'' and x'' are then

$$e''(S|M'') = (e(S))|M'' \qquad x''(S|M'') = x(S)$$

In order for e'' and x'' to be well-defined in this way, it must be true, for each $S_1, S_2 \in \mathcal{S}$, that

$$\text{if } S_1|M'' = S_2|M'' \quad \text{then } (e(S_1))|M'' = (e(S_2))|M'' \text{ and } x(S_1) = x(S_2)$$

That is, the final values of any variables not defined in the block, and the exit, must not depend on initial values of any variables defined in the block. This, of course, is due to the fact that entering a block in which variables are defined does not, in itself, set these variables to any values (even default values, such as zero, although conceivably a block structure language could be described in which default values were assigned in this way). These facts about a block must be proved correct as part of the proof of correctness of the block.

To derive e' and x' from e" and x", we simply specify that the values of any variables whose names are re-used in the inner block cannot change during the execution of that block. Thus the rules are

$$(e'(S'))\big|M' = e''(S'|M'),\ e'(z) = z \text{ for } z \notin M';\ x'(S') = x''(S'|M')$$

In general, this, again, is true only in the absence of procedures. To see this, consider the following ALGOL program:

```
begin integer k;
procedure loop(loopflag); Boolean loopflag;
begin k:=k+1; loopflag:=false;
if k ≤ 10 then loopflag:=true end loop;
k:=1;
begin integer k; Boolean x; k:=1;
a: k:=k + k;
loop(x); if x then go to a;
outinteger(k) end;
outinteger(k) end
```

The output here consists of the numbers 1024 (that is, 2^{10}) and 11. The program manipulates two variables named k at the same time; denoting the outer k by α and the inner k by β, the statement k:=k + k always sets $\beta = 2^{\alpha}$, and the statement loop(x) sets $\alpha = \alpha + 1$. The effect of loop(x) involves state vectors without α-components, and it is possible to calculate an effect for the inner loop which likewise involves state vectors without α-components. However, in the calculation of the effect of the inner loop by considering the effects of its individual statements, the state vectors involved must have both α-components and β-components.

In the absence of <u>recursive</u> procedures, this problem may be

solved with only a slight increase in complexity. If each block has a unique name, the domain of each state vector, instead of being simply a set of variable names, is now a set of pairs (bn, vn), where bn is a block name and vn is a variable name defined in the corresponding block. If the names of blocks are not unique, or (more commonly) if some of the blocks do not have names, we assign to each block in the program a block index bi, analogous to the statement index defined earlier; that is, if a block b has index k, then b is the k-th block in its program. Pairs (bi, vn) are now used in place of the pairs (bn, vn). With each block we may associate as a semantic attribute, not only its own block index, but the sequence of indices of all blocks in which it is contained, working outward and ending with the index of the outermost block (normally 1). Each statement now inherits the sequence associated with the block in which it is contained, and this sequence is further inherited by the expression or expressions in this statement and ultimately by each variable in the statement. It is now easy to determine, for the variable name vn, which pair (bi, vn) represents it: we simply look at the sequence of indices bi which we have inherited, and pick the first one for which there exists a pair (bi, vn) for this particular vn. This corresponds to taking the innermost possible definition of a variable name.

Somewhat the same solution may be applied if the concepts of local and global variables are used, as in FORTRAN. Here we have no blocks, but we have main programs, functions, and subroutines, each of which may use variable names that need be unique only within itself. This time, since each function and each subroutine has a name, the pairs may be of the form (pn, vn), where pn is a program name (a special name, such as $MAIN, in the case of the presumably unique main program) and vn is a variable name which is local to that program. In the absence of global variables, the name of each program is in-

herited as an attribute of each variable in that program, and used as the first element of the pair (pn, vn).

Global variables in FORTRAN arise in three ways. The mathematically simplest system, available only in extended versions of FORTRAN, uses ENTRY and EXTERNAL declarations. We have seen in section 4-2 how an ENTRY declaration may be used to specify a FORTRAN program with multiple entry points, but, in some FORTRAN systems, it may be used to specify global variables as well. An EXTERNAL declaration specifies that reference to a certain variable name in the program containing that declaration refers to its definition in some other program as a global variable (with an ENTRY declaration). From the viewpoint of syntactic and semantic description, there is no difference between the two declarations, although for implementation purposes the variables declared in ENTRY declarations in a program normally occupy memory addresses next to the instruction words of that program. We set up a special name ($GLOBAL, for example) as the "program name" of all global variables given either by ENTRY or by EXTERNAL declarations.

When global variables are defined using blank common, each one will be assigned a common variable index. If α is the k-th common variable in a program, then α has common variable index k. All state vectors involving such a variable have a k-component, rather than an α-component, because, by the definition of blank common, the declaration of common variables defines their position (that is, the common variable index) only. If β is the k-th common variable in a subroutine of the above program, then α and β are considered to be the same variable. (This would happen, for example, if the program contained the statement COMMON X, ALPHA, Y and the subroutine the statement COMMON X, BETA, Y with no other COMMON statements in either program.) A variable α defined in labeled common, in a common block called b,

is similarly represented in state vectors by a (b, k)-component, where k is its common variable index relative to the block b; that is, α is the k-th variable to be defined in the block b.

Again, all of this analysis is in the absence of recursive subroutines; these, of course, are not allowed in FORTRAN. If subroutines are allowed to be recursive, there may be different variables with the same name at various recursion levels. There are several possible treatments of state vectors in this case, all of which seem either more cumbersome or more indirect than the treatments we have analyzed thus far. We may give our state vectors (cs, vn)-components, where vn is a variable name and cs is the current contents of the stack (that is, the general-purpose stack used in implementing the language). We may give our vectors vn-components in which the value of each variable is a sequence of values of all currently active variables with the name vn. A third solution is to impose an intermediate stage in the form of an infinite sequence of "generalized addresses" a_1, a_2, ...; a state vector in our sense is then the composition of a mapping L of variables into generalized addresses and a mapping C of generalized addresses into their current contents or values. Entry to a block or procedure, whether recursive or not, corresponds to a change in the current mapping L. This construction models the usual implementation of block-structure languages; its apparent disadvantage is that, if it is used, such statements as the fact that a block or procedure always does the same thing, regardless of the current contents of the stack, must be proved instead of being "built in" to the description of the language. Further discussion is given in section 5-5.

NOTES

Recursive conditional expressions play a central rôle in McCarthy's theory of computation [McCarthy 63]. In fact, McCarthy shows that a computation theory may be built up entirely in terms of recursive functions, including recursive conditional expressions, without any of the usual mechanisms of assignment and transfer of control. Roughly speaking, a conventional program P with n statements may be replaced by n functions of all the variables of P, each of which is defined as its successor (or as various of its successors, depending upon a conditional expression) with the same arguments, except that the argument corresponding to the assigned variable, if any, is replaced by the expression assigned. Using Ω to stand for "undefined" (which is taken as a third truth-value, in addition to true and false), a typical FORTRAN program is expressed in this theory roughly as follows:

```
      INTEGER FUNCTION GCD(M, N)
7     I = M
8     J = N
1     IF (I-J) 2, 4, 3
2     J = J - I
      GO TO 1
3     I = I - J
      GO TO 1
4     GCD = I
      RETURN
      END
```

```
gcd(m, n) = p7(Ω, Ω, m, n)
p7(i, j, m, n) = p8(m, j, m, n)
p8(i, j, m, n) = p1(i, n, m, n)
p1(i, j, m, n) = if i-j<0 then
    p2(i, j, m, n) else if i-j=0 then
    p4(i, j, m, n) else p3(i, j, m, n)
p2(i, j, m, n) = p1(i, j-i, m, n)
p3(i, j, m, n) = p1(i-j, j, m, n)
p4(i, j, m, n) = i
```

The definition of gcd(m, n) given above is equivalent to the usual definition of the greatest common divisor if m>0 and n>0, and cor-

responds quite closely to the given FORTRAN program. Of course, although none of the functions p1, p2, p3, p4, p7, or p8 calls itself, the entire collection of functions is ultimately recursive; for example, p1 calls p2 and p3, both of which call p1.

Our derivation of the generalized effect from the single-step function is similar to the construction of the so-called "tail function" [Mazurkiewicz 71]. The tail function corresponds to the generalized effect, and the transition function from which it arises corresponds to the single-step function.

The material on multiple exits and escape is new, so far as can be determined. The observation that the multiple use of names does not make it necessary to assign unique names to variables in a block structure language unless procedure calls are allowed is likewise new (see, for example, [Burstall 70] for the "unique name" approach). The subject of iteration, at least in this author's opinion, seems to have been avoided, up to now, more because of its intrinsic cumbersomeness than due to any logical difficulty. Several people have experimented with formalizing the "while" type of for-list element; notice that the "step-until" type reduces to the other two types, for we can always replace A step B until C by A, A+B while (C-A)×sign(B) < 0. For an alternate approach to iteration, using the FORTRAN DO statement, see [Maurer 72].

Generalized addresses, as suggested at the end of section 4-5, are introduced in [Strachey 66]. The concept of the state vector as a composition of two mappings, the set of generalized addresses forming the range of the first and the domain of the second, appears explicitly in [Park 68] and also in [Kaplan 68].

EXERCISES

1. How might the syntactic rule

<section> ::= <statement> | <section> ';' <statement>

given at the beginning of section 4-1, be altered (using semantic rules and semantic restrictions) in order to describe a language which does not use semicolons, but rather uses one statement per card as in FORTRAN (assume no continuation cards), and in which a card consists of

(a) n (≤ 72) characters followed by a carriage return character?

(b) exactly 72 characters, with possible trailing blanks?

2. The syntactic rule

<blanks> ::= ' ' | <blanks> ' '

in a language in which blanks are to be ignored, such as FORTRAN, may be altered to

<blanks> ::= ' ' | <carriage return character> '.' | <blanks> ' ' | <blanks> <carriage return character> '.'

to denote the fact that a carriage return character followed by a period is likewise to be ignored. Thus a period in column 1 effectively serves as a continuation character. Give analogous rules for the cases in which

(a) the continuation character, a period, is in the 72nd column of the (72-column) card before the continuation card;

(b) continuation is as in FORTRAN, with any non-blank character in column 6 signifying a continuation card.

3. Let $f: \mathfrak{I} \to \mathfrak{I}$ be an arbitrary single-step function, where $\mathfrak{I} = \prod_{x \in M'} V_x$ with $M' = M \cup \{\lambda\}$, and let P be an element of V. Treating λ as a program counter whose values are statement indices, P is thus such a statement index. Give explicit rules, in terms of f, for the effect and exit index of the statement with statement index P. The effect should be a function $e: \mathfrak{S} \to \mathfrak{S}$, where $\mathfrak{S} = \prod_{x \in M} V_x$; the exit index should be a function $x: \mathfrak{S} \to V_\lambda$.

4. Give a semantic rule involving a recursive conditional expression for the WHILE statement, defined syntactically as follows:

<WHILE statement> ::= 'WHILE' <Boolean expression> 'DO'
<statement>

Assume that the Boolean expression has a value, but no side effect, and that both the WHILE statement and the statement it contains have an effect but always exit normally. The intuitive meaning of WHILE b DO s is: if b is true, then stop; otherwise, do s and then loop back to test again if b is true.

5. (a) Modify the syntactic and semantic rules given for if statements in section 4-3 to take account of statement indices as in the example programs of that section; that is, the statement index of the if statement and of the statement it contains must be properly set up. How does this affect the semantic rule given in section 4-1 for the number of statements in a program section?

(b) Modify these rules further, to include statements of the form if B then U else V, where B is a Boolean expression and U and V are statements. Note that either U or V, or both, may be labeled, and that, if U is labeled and transfer is made to it, U is executed and transfer is then made around V. This transfer may in all cases be taken as the semantics of the word else (as a complete statement).

6. Modify the rules for

(a) factors,

(b) if clauses,

(c) arithmetic expressions,

(d) Boolean terms,

given in section 2-5, to allow for the possibility of escape.

7. The requirement, suggested in section 2-4, that the value of any exit index should be the special word normal whenever exit is made in the normal manner, necessitates changes in certain of the rules which we have studied. Discuss the changes, if any, which would be indicated in the rule for the exit index of

(a) a general FORTRAN IF statement,

(b) an assignment statement,

(c) a go to statement in ALGOL,

as given in section 3-3.

8. Iteration in assembly language is often aided by means of special instructions. Give syntactic and semantic rules, in the context of the assembly language of section 3-5, for the instruction

LOOP I

which decrements I by 1 and skips the next instruction (which is presumably a transfer back to the beginning of the loop) if $I \leq 0$ after the decrementing operation. (The PDP-10 instruction SOSLE -- Subtract One and Skip if Less or Equal to zero -- does this, among other things.)

9. Consider the statement made in section 4-5 that the final values of any variables not defined in a block must not depend on initial values of any variables defined in the block. Does this im-

ply that every variable defined in a block must be set to some value before it is used -- that is, must appear on the left side of an assignment (in execution order, and assuming no side effects) before it may appear on the right? Why or why not?

10. Consider the statement made in section 4-5 that it is not necessary, in the absence of procedures, to consider state vectors with more than one α-component, for a given variable name α. When ALGOL-style procedures are present, we showed here that this statement no longer holds; but our example involves an ALGOL convention not present in FORTRAN -- namely, the ability, within any block, to use quantities declared not only in that block but also in any outer block. Thus, in the example, we called loop(x) from inside the inner block, although the loop procedure was defined at the outer level. Suppose now that the FORTRAN style of subroutine calling is used. Since FORTRAN does not use recursive subroutines, it is possible to arrange a program and its subroutines in order, in such a way that any program calls only programs before it in order (or only programs after it in order). Under these conditions, is it possible, as we have done in the absence of procedures, to use state vectors with only one α-component for each variable name α?

CHAPTER FIVE

DECLARATIONS

5-1 Type Declarations

The purpose of declarations in a program is normally to state certain properties of identifiers in that program. In our terminology, the declarations determine the environment, which is considered as a set of ordered pairs. Each declaration contributes certain ordered pairs as its local environment. The union of all the local environments is the environment of the program, which is then inherited by each statement in the program. These relations are expressed by the following simplified description of programs in which declarations and executable statements are separated by semicolons:

<set of declarations> x ::= <declaration> d; y ::= <set of declarations> z ';' <declaration> e

<local environment> $x^z = d^z$; $y^z = z^z \cup e^z$

<program> p ::= <set of declarations> a <set of statements> b

$b^y = a^z$

<set of statements> x ::= <statement> s; y ::= <set of statements> z ';' <statement> t

<environment>^{y}

$s^y = x^y$; $z^y = y^y$; $t^y = y^y$

It is further assumed that the environment is inherited by the components of a statement -- the expressions, factors, terms, and pri-

mary expressions, and ultimately by the variable names. This is necessary in order to determine all properties of a variable name which will be used in the synthesis of values of expressions, effects of statements, and so on.

The semantic rule $b^y = a^z$ in the definition of a program given above is still another variation on the concept of an inherited attribute. The set of statements b is here inheriting its environment from its "brother," the set of declarations a. Like the rule for the label function of a compound statement given in section 4-1, this rule may be replaced by two conventional "father-to-son" rules, namely

<local environment> $p^z = a^z$

$b^y = p^z$

In a language in which the only declarations are type declarations, the environment may be cast in the form of a type function, as studied in section 2-4. If this function is called $\underline{t}$, then $\underline{t}(\underline{x})$ is the type of the variable $\underline{x}$. The type function for ALGOL type declarations (ignoring the use of own for the moment) may be constructed by using the following rules:

<type> $\underline{r}$ ::= 'real'; $\underline{i}$ ::= 'integer'; $\underline{b}$ ::= 'Boolean'

<type set> r^t = real; i^t = integer; $b^t = \{\text{true, false}\}$

<type declaration> $\underline{d}$ ::= <type> $\underline{t}$ <type list> $\underline{x}$

$x^t = t^t$

<type function> $d^z = x^z$

<type list> $\underline{x}$ ::= <simple variable> $\underline{y}$; $\underline{y}$ ::= <simple variable> $\underline{w}$ ',' <type list> $\underline{z}$

<type set>^{t}

$z^t = y^t$

<type function> $x^z = \{(v, x^t)\}$; $y^z = \{(w, y^t)\} \cup z^z$

Thus, in particular, the type declaration

integer x1, x5, zeta

would have as its type function the set of pairs

{(x1, integer), (x5, integer), (zeta, integer)}

We have made a distinction here between the keywords real and integer, which appear in ALGOL programs, and the sets real and integer, which are sets of all allowable real numbers and integers respectively. These sets may be defined in different ways, of course, for implementations on different actual computers. Note that type lists involve "simple variables" which are, in particular, not subscripted variables.

When a language contains other declarations, we might try to set up a separate function, like the type function, for each kind of information to be attached to a variable name. This, however, would mean that each of these functions would have to be a separate inherited attribute. It is somewhat simpler to have a single function of two arguments, of which the second is a special keyword describing the kind of infomation being passed. This is the function which we shall, from now on, call the environment. In the case of type information, we shall use the special keyword type. Thus for the example

integer x1, x5, zeta

of a type declaration given above, instead of a type function t with t(x1) = t(x5) = t(zeta) = integer, we wish to construct an environment y with y(x1, type) = y(x5, type) = y(zeta, type) = integer. This means that the set of ordered pairs we wish to construct becomes

{((x1,type),integer), ((x5,type),integer), ((zeta,type),integer)}

and the construction may be made by giving a semantic rule for a local environment, rather than a type function, as follows:

$$\langle\text{local environment}\rangle\ x^z = \{((v, \underset{\sim}{\text{type}}), x^t)\};\ y^z = \{((w, \underset{\sim}{\text{type}}), y^t)\} \cup z^z$$

Strictly speaking, if the local environment is **z**, we now have **z**((v, type)) = t, rather than **z**(v, type) = t, where v is a variable of type t.

The type function, or the type information in the local environment, determines the set of all state vectors which forms the domain of the functions giving the value, effect, exit index, etc., of various nonterminals in the language. If t is the type function and S is in the set of state vectors we wish to determine, then S has the same domain as t, and S(x), for each variable x, is a member of the set t(x). The set of all state vectors having this property will be called the state vector domain t^D of the type function t. Thus

$$t^D = \{S: \text{domain}(S) = \text{domain}(t) \text{ and } S(x) \in t(x) \text{ for all } x \in \text{domain}(t)\}$$

If the nonterminal x (say) has the associated type function x^t, we shall denote the state vector domain of x^t by x^{tD}. (Recall the discussion in section 1-3; if t^D were denoted by D_t, and x^t by t_x, then x^{tD} would have to be denoted by D_{t_x}, involving two subscript levels.) We shall use y^D for the state vector domain of a more general environment y, with the understanding that y^D depends only upon the type information in y; thus

$$y^D = \{S: \text{domain}(S) = \{x: y(x, \underset{\sim}{\text{type}}) \text{ is defined}\} \text{ and } S(x) = y(x, \underset{\sim}{\text{type}}) \text{ for all } x \in \text{domain}(S)\}$$

In all of the semantic rules which have been presented so far, the domain of the state vector functions which are semantic attributes

has been left understood. If we wish to specify this domain, we may, in the interest of brevity, include it within the parentheses which enclose the first usage of a state vector in that domain, the state vector symbol (normally S) then being followed by $\in$ (is a member of). Consider, for example, the first semantic rule involving state vectors given in section 2-3:

<expression> e ::= <term> v '+' <term> w
<value> $e^V(S) = v^V(S) + w^V(S)$

Suppose now that an expression also has an environment, which is, as usual, an inherited attribute. Then we might write

<expression> e ::= <term> v '+' <term> w
<environment>^{y}
$v^y = e^y$; $w^y = e^y$
<value> $e^V(S \in e^{yD}) = v^V(S) + w^V(S)$

By writing $e^V(S \in e^{yD})$, rather than simply $e^V(S)$, we are defining the domain of e^V as e^{yD}. The state vector symbol S may then be used, as usual, in the remainder of the semantic rule.

We have assumed throughout this section that our language is such that all declarations precede all executable statements. In some languages, this does not hold; declarations and executable statements may be mixed (and in fact the word "statement" often becomes a more general term, so that a declaration is also called a statement). In such a case, the following general rules may be used:

<program> $\underline{w}$ ::= <declaration> $\underline{d}$; $\underline{x}$ ::= <executable statement> $\underline{s}$; $\underline{y}$ ::= <program> $\underline{p}$ ';' <declaration> $\underline{e}$; $\underline{z}$ ::= <program> $\underline{q}$ ';' <executable statement> $\underline{t}$

<environment>^{y}

$s^{y} = \phi$; $p^{y} = y^{y}$; $q^{y} = z^{y}$; $t^{y} = z^{y}$

<local environment> $w^{z} = d^{z}$; $x^{z} = \phi$; $y^{z} = p^{z} \cup e^{z}$; $z^{z} = q^{z}$

<complete program> ::= <program> $\underline{p}$ 'end'

$p^{y} = p^{z}$

Here it is assumed, as before, that d^{z} is the set of pairs associated with d. The nonterminal <complete program> used here involves an end-marker, a standard technique often used with syntactic rules. The end-marker here is end, and it is assumed that no other rule in the description of this language involves end. If this is true, and the string S is a complete program, then no initial substring of S is a complete program. That is, if the characters of S, in order, are $c_1, \ldots, c_n$, then the string consisting of $c_1, \ldots, c_k$, for $k \leq n$, cannot be a complete program unless $k = n$. This property of languages is very convenient for the purposes of compilers and other translators which work on a program from left to right; if the property does not hold, such a translator might very easily "stop too soon" and translate only part of the source program.

5-2 Array Declarations

In a language in which array dimensions are always integer constants, each element of each array may be thought of as belonging to the domain of the type function, as defined above. Thus for an array in FORTRAN given by DIMENSION A(3), there are three real variables, A(1), A(2), and A(3); if t is the type function, then t(A(1)) = t(A(2)) = t(A(3)) = real. We may construct a simple rule for such array declarations as follows:

<array declaration> d ::= 'DIMENSION' <variable> v '('
<integer> i ')'
<type function> d^z = $\{$(concat(v, '(', k, ')', v^t):
$1 \leq k \leq i^v\}$

It is assumed here that the integer i has value i^v, and that the variable v has type v^t, which is in this case its default type (integer if v begins with I, J, K, L, M, or N, and real otherwise).

We have introduced here still another way of giving a semantic rule. Instead of specifying the set d^z by listing all its elements, or by an operation such as intersection, union, or cartesian product, we have specified it here as the image of a function. If X is any set and f is any function whose domain is X, then

$$\{f(x): x \in X\}$$

stands for the set of all elements f(x) for all $x \in X$. The definition of d^z above is effectively of this kind; the set X is the set of all integers between 1 and the value of i, inclusive, while f(i), for the integer i, is the ordered pair (A(i), t), for the array A of type t.

The function concat(s_1, ..., s_n) is defined as the string formed by concatenating the strings s_1, ..., s_n. Its appearance in the above

rule is a slight abuse of notation, since k is an integer, not a string. In fact, we do not need to use the concatenation function if we agree that the ordered pair (A, i), rather than the string 'A(i)', be in the domain of the type function for each i. Our rule for the type function of an array declaration would then read

$$\text{<type function>}\ d^Z = \{((v, k), v^t) : 1 \leq k \leq i^v\}$$

In the case of DIMENSION A(3), the type function would thus be the set of pairs

$$\{((A, 1), \underset{\sim}{\text{real}}), ((A, 2), \underset{\sim}{\text{real}}), ((A, 3), \underset{\sim}{\text{real}})\}$$

Generalization of these rules to multidimensional arrays, and to array declarations with specified types, is immediate. The second element of each pair (A, k), where A is the array name, is now itself an n-tuple of integers, for n-dimensional arrays. If b is the list of subscript bounds for a given array, we shall denote the set of all such n-tuples which are legal for that array by b^u. The rule for array declarations now becomes

$$\text{<array declaration>}\ d ::= \text{<type>}\ t\ \text{<variable>}\ y\ \text{'('}\ \text{<subscript bound list>}\ b\ \text{')'}$$
$$\text{<type function>}\ d^Z = \{((v, k), t^t) : k \in b^u\}$$

The method of definition of d^Z is essentially the same here as it was before; the second element of each pair ((A, k), t) is now the set of values of the given type t, rather than being determined by the variable name y.

To derive b^u from b, we may use a semantic rule involving cartesian products. If the subscript bound list is a list of integers separated by commas, giving the upper bounds, while the lower bounds are always 1 (as, for example, in FORTRAN), the rule becomes

<subscript bound list> x ::= <integer> i; y ::= <subscript bound list> z ',' <integer> j

<set of n-tuples> $x^u = \{k: 1 \leq k \leq i^v\}$; $y^u = z^u \times \{k: 1 \leq k \leq j^v\}$

Here, as before, i^v and j^v are the respective values of the integers i and j. This semantic rule involves still another method of specifying a set, namely the set of all quantities satisfying a certain property. If X is a set and P is a property of elements of X, then

$$\{x \in X: P(x)\}$$

stands for the set of all elements of X which have the property P. In the above rule, strictly speaking, we should write $\{k \in \underline{integer}: 1 \leq k \leq v\}$, rather than simply $\{k: 1 \leq k \leq v\}$, where v is either i^v or j^v and where integer, as before, is the set of all allowable integers.

We may generalize the above rule for subscript bound lists to make them look like bound pair lists in ALGOL, although still restricting them to be integer constants rather than expressions:

<subscript bound list> x ::= <integer> a ':' <integer> b;
y ::= <subscript bound list> z ',' <integer> c
':' <integer> d
$(a^v \leq b^v)$
$(c^v \leq d^v)$
<set of n-tuples> $x^u = \{k: a^v \leq k \leq b^v\}$; $y^u = z^u \times \{k: c^v \leq k \leq d^v\}$

Note still another use of semantic restrictions, this time to insure that each lower bound is not greater than the corresponding upper bound.

Array declarations may be further generalized to allow several arrays to be declared at once, just as type declarations may declare the type of several variables at once. ALGOL involves another kind of generalization: if several arrays have the same bounds, then the bounds need not be specified except for the last of these arrays. Thus, for example,

real array a, b, c[1:100]

has the same meaning as

real array a[1:100], b[1:100], c[1:100]

which in turn has the same meaning as

real array a[1:100]; real array b[1:100]; real array c[1:100]

-- that is, it declares three arrays, a, b, and c, each containing 100 real numbers. In fact the word real may be omitted from such a declaration in ALGOL, so that we could have written simply

array a, b, c[1:100]

in the first place. Syntactic and semantic rules to cover these generalizations may be given as follows:

<array declaration> x ::= 'array' <array list> a; y ::=
    <type> t 'array' <array list> b
<type function> $x^Z = a^Z$; $y^Z = b^Z$
a^t = real; $b^t = t^t$

<array list> x ::= <array segment> s; y ::= <array
    list> z ',' <array segment> t
<type set>^{t}
$s^t = x^t$; $z^t = y^t$; $t^t = y^t$
<type function> $x^Z = s^Z$; $y^Z = z^Z \cup t^Z$

<array segment> x ::= <array identifier> i <subscript bound list> b; y ::= <array identifier> j ',' <array segment> z

<type set>^{t}

$z^t = y^t$

<set of n-tuples> $x^u = b^u$; $y^u = z^u$

<type function> $x^z = \{((i, k), x^t): k \in b^u\}$; $y^z = z^z \cup \{((j, k), y^t): k \in z^u\}$

where subscript bound lists are as defined above.

When a subscripted variable is used, the semantic rules for its value depend upon the form of the semantic rules for its declaration. In ALGOL, we may write

<variable> v ::= <simple variable> s; w ::= <subscripted variable> u

<L-value> $v^l = s^l$; $w^l = u^l$

<value> $v^v(S) = S(v^l(S))$; $w^v(S) = S(w^l(S))$

using the relation between values and L-values given in section 3-2. This makes it necessary to define only L-values, and not ordinary values, of simple and subscripted variables. The L-value of a simple variable, of course, is the variable itself; this is a constant function of the state vector. The L-value of a subscripted variable depends on the evaluation of the subscripts, which in turn depends on the state vector. Let us assume that each subscript list (that is, each sequence of subscripts, separated by commas) has as a semantic attribute an n-tuple of integer values which is a function of the state vector. Such an attribute might be defined, for example (in the absence of side effects, and assuming no type conversion) by

<subscript list> x ::= <subscript expression> s; y ::=
<subscript list> z ',' <subscript expression> t
<n-tuple of values> $x^v(S) = s^v(S)$; $y^v(S) =$
concat($z^v(S)$, ',', $t^v(S)$)
<subscript expression> s ::= <arithmetic expression> a
<value> $s^v(S) = a^v(S)$
($a^t(S) \equiv$ integer)

where a^t is the type of the arithmetic expression a. (In the presence of side effects, we would normally, as we have done before, assume that evaluation takes place from left to right.) Subscripted variables may now be defined by

<subscripted variable> s ::= <array identifier> a '['
<subscript list> t ']'
<L-value> $s^l(S)$ = concat(a, '[', $t^v(S)$, ']')

The rank of an identifier is the number of subscripts that should follow it. By including rank information as part of the environment, we may make semantic restrictions which insure that the number of subscripts in any use of an array name agrees with the number of subscripts appearing at the time it was defined. We simply add to the definition of array segments, given above, the semantic rules

<rank> $x^r = b^n$; $y^r = z^r$
<local environment> $x^z = \{((i, \text{rank}), x^r)\}$; $y^z =$
$z^z \cup \{((j, \text{rank}), y^r)\}$

where b^n is the length (defined in the obvious manner) of the subscript bound list b; if this local environment is synthesized and in-

herited in the manner described in the previous section, then we may include the semantic restriction

$$(s^{\gamma}(a, \underset{\sim}{\text{rank}}) = t^{n})$$

on the rule above for subscripted variables, where s^{γ} is the environment of the subscripted variable s and t^{n} is the length of the subscript list t. We may also include the restriction

$$(v^{\gamma}(i, \underset{\sim}{\text{rank}}) = 0)$$

on the rule for the unsubscripted variable v which is being defined as the identifier i. To complete the picture, we may then include rank as well as type information in the local environment of such an unsubscripted variable, giving the rank of such a variable as zero. These rules assure us that no variable name can occur in a program in both subscripted and unsubscripted form, and that every subscripted variable must be declared as such.

Can semantic restrictions similar to those above be imposed to insure that subscripts never go out of range? Except for constant subscripts, the answer is no, even when the subscript bounds are constant. The problem is that we have no way of knowing, in general, whether a given variable will stay in bounds. If the subscript bounds are constant, we may define a subset of state vectors for which the subscripts are in range as the domain of state vector functions involving subscripted variables. This device may be extended to handle the case in which the subscript bounds may be variable: we associate with each array name two new variables, a lower bound variable and an upper bound variable (only the latter is needed, of course, if the lower bound must be constant, as in FORTRAN). The values of these variables are always the current bounds, and, since these variables

have state vector components, we may, just as before, determine a set of state vectors for which the subscripts are all in range -- a state vector being in that set if it specifies bounds which it then satisfies.

5-3 Initializing Declarations

We now pass to a kind of declaration which is not found in ALGOL, although it is found in FORTRAN and in PL/I: the declaration which imparts initial values to certain variables. In FORTRAN, this is called the DATA declaration. Thus

DATA P, Q, R/5.0, -7.0, 3.0/

gives the initial value of P as 5.0, of Q as -7.0, and of R as 3.0. In PL/I, the same initial values would be given to the same variables by

DECLARE P INITIAL (5.0), Q INITIAL (-7.0), R INITIAL (3.0)

If every variable in a program is given an initial value in this way, the result is an initial state vector S_0; the final state vector is then $e(S_0)$, where e is the effect of the program. In some languages, and under some operating systems, any variable which is not declared to have an initial value is given one by default. Thus, in SNOBOL, the initial value of every variable is taken to be the null string of characters; while many operating systems clear the data area before execution, that is, they set all cells in this area to zero. In such a case there is always a single initial state vector, whether initializing declarations are present or not.

How should the initial state vector be defined in case the program variables are not all given initial values? One solution is to introduce a new value, "undefined," often denoted by Ω. When Ω appears as an argument of any function, the value of that function is taken to be Ω; in particular, $\Omega + n = \Omega$, $\Omega - n = \Omega$, and so on, for every n. The initial value of any variable is now defined to be Ω whenever it is not defined otherwise. Thus we have an ini-

tial state vector, some or all of whose components may be Ω; certain components of the final state vector, of course, may then conceivably also be Ω. However, normally this does not matter, as long as the undefined components do not represent output of the program. We may note that the use of Ω is often indicated in other situations; for example, it is the k-component of e(S) where e is the effect of any ALGOL statement beginning for k:= and S is any state vector for which this statement exits normally, since in that case the ALGOL report specifies that the value of k is undefined after a normal exit.

If the introduction of Ω seems unnatural, we may choose, instead, to deal with a collection of possible initial state vectors. Let M be the set of all variables of the program, and let $M' \subseteq M$ be the set of all those variables which are initialized either by default or by the use of initializing declarations. Let e be the effect of the given program, as before, and let S_0 be a particular initial state vector, that is, one which assigns to each variable in M' its given initial value. If S is any other initial state vector, that is, if S agrees with S_0 on M', then it is reasonable to demand that e(S) must agree with $e(S_0)$ on some other set $M'' \subseteq M$, which usually, although not necessarily, consists of the output variables of the program; that is,

$$(S|M' = S_0|M') \Rightarrow ((e(S))|M'' = (e(S_0))|M'')$$

This equation may be made into a semantic restriction on any programming language, provided that e, S_0, M', and M'' are appropriate semantic attributes. We have already discussed the effect e of a program as a semantic attribute of it; the set M'' may be determined in a straightforward manner, for example by reference to the kinds of output statements occurring in the given program. It remains for the

initializing declarations to specify S_0 and M^s as semantic attributes. (We note that, in the case of the for statement above, its effect will be a multi-valued function if we carry through the philosophy described above, avoiding the use of Ω.)

We shall determine a semantic attribute of programs which may be called the partial initial state vector. It is a state vector whose domain is M^s, and which is given by semantic rules as a set of ordered pairs. Any state vector of the program whose restriction to M^s is the partial initial state vector, as determined in this way, may then be taken as S_0. In the case of the DATA declaration in FORTRAN, or its equivalent in PL/I, which were introduced at the beginning of this section, the set of pairs

$$\{(P, 5.0), (Q, -7.0), (R, 3.0)\}$$

would correspond to the partial initial state vector. Our first task is to associate a function $\underline{v}$ with each list of values, such that $v(i)$ is the value of the i-th element of the list. Thus for the list

$$5.0, 7.0, 3.0$$

we would have $v(1) = 5.0$, $v(2) = 7.0$, and $v(3) = 3.0$. In the simple case in which each value is a single constant, v may be defined by

$$\begin{aligned} &\langle\text{list of values}\rangle\ \underline{x} ::= \langle\text{constant}\rangle\ \underline{c};\ \underline{y} ::= \langle\text{list of values}\rangle\ \underline{z}\ \text{`,'}\ \langle\text{constant}\rangle\ \underline{d} \\ &\langle\text{length}\rangle\ x^n = 1;\ y^n = z^n + 1 \\ &\langle\text{value function}\rangle\ x^v = \{(1, c^v)\};\ y^v = z^v \cup \{(y^n, d^v)\} \end{aligned}$$

where c^v and d^v are the respective values of $\underline{c}$ and $\underline{d}$. Value functions, like most of the other functions we have introduced as semantic attributes, are given as sets of ordered pairs.

DATA declarations in FORTRAN also permit elements of the form n*v to appear in a list of values, where n is a positive integer and v is any constant; the meaning of this construction is "repeat v, n times." To allow such elements, we might write

<list of values> $\underline{w}$ ::= <constant> $\underline{a}$; $\underline{x}$ ::= <integer> $\underline{i}$ '*' <constant> $\underline{b}$; $\underline{y}$ ::= <list of values> $\underline{u}$ ',' <constant> $\underline{c}$; $\underline{z}$ ::= <list of values> $\underline{v}$ ',' <integer> $\underline{j}$ '*' <constant> $\underline{d}$

<length> $w^n = 1$; $x^n = i^v$; $y^n = u^n + 1$; $z^n = v^n + j^v$

<value function> $w^v = \{(1, a^v)\}$; $x^v = \{(k, b^v): 1 \leq k \leq i\}$; $y^v = u^v \cup \{(y^n, c^v)\}$; $z^v = v^v \cup \{(k, d^v): v^n < k \leq z^n\}$

The DATA declaration itself is now defined by

<DATA declaration> $\underline{d}$ ::= 'DATA' <list of variables> $\underline{x}$ '/' <list of values> $\underline{y}$ '/'

<partial initial state vector> $d^p = x^p$

$x^v = y^v$

$(x^n = y^n)$

where a list of variables, in the simplest case, is given by

<list of variables> $\underline{x}$ ::= <identifier> $\underline{i}$; $\underline{y}$ ::= <list of variables> $\underline{z}$ ',' <identifier> $\underline{j}$

<length> $x^n = 1$; $y^n = z^n + 1$

<list of values>^{v}

<partial initial state vector> $x^p = \{(i, x^v(1))\}$; $y^p = z^p \cup \{(j, y^v(y^n))\}$

We have required here, by the semantic restriction $(x^n = y^n)$, that the

length of the list of values be the same as that of the list of variables. This restriction may be generalized to $x^n \leq y^n$ without further altering the semantic rule.

The rule for a list of variables may be generalized to permit implied looping, so that, for example, the statement

```
DATA (A(I), I=1,100)/100*0/
```

may be used to set the variables A(1) through A(100) to zero. The generalized rule must be accompanied by a rather subtle semantic restriction: we would like to be able to replace A(I), in the above example, by A(I+1), by A(I, I), or even simply by I, but not by A(I, J); that is, we do not want the statement

```
DATA (A(I, J), I=1,100)/100*0/
```

to be legal (since this is merely a declaration and the value of J would be unspecified). We may obtain the names of the variables which should appear in the partial initial state vector by finding the L-values of the variable which appears, applied to state vectors in which the component corresponding to the implied loop index successively assumes all integer values between the given lower and upper bound. The semantic restriction mentioned above is then the statement that the L-values of this variable applied to any two state vectors are the same whenever those state vectors have the same loop index component. Assuming that a variable v (whether simple or subscripted) always has an L-value v^l, a rule for lists of variables may then be given as

<list of variables> w ::= <identifier> c; x ::= '('
<variable> a ',' <identifier> i '=' <integer> s
',' <integer> e ')'; y ::= <list of variables> u ','
<identifier> d; z ::= <list of variables> y ',('
<variable> b ',' <identifier> j '=' <integer> t
',' <integer> f ')'

<length> $w^n = 1$; $x^n = e^v - s^v + 1$; $y^n = u^n + 1$;
$z^n = v^n + f^v - t^v + 1$

$(s \leq e)$

$(t \leq f)$

<list of values>^{v}

<partial initial state vector> $w^p = \{(c, w^v(1))\}$;
$x^p = \{(a^\ell(S_k), x^v(k-s+1)): s \leq k \leq e\}$, where $S_k(i) = k$; $y^p = u^p \cup \{(d, y^v(y^n))\}$; $z^p = v^p \cup \{(b^\ell(S_k), z^v(v^n+k-t+1)): t \leq k \leq f\}$, where $S_k(j) = k$

$(s \leq S(i) = S'(i) \leq e \Rightarrow a^\ell(S) = a^\ell(S'))$

$(t \leq S(j) = S'(j) \leq f \Rightarrow b^\ell(S) = b^\ell(S'))$

This rule may easily be generalized to DATA declarations such as

DATA (A(I), B(I), I=1,99,2)/100*0/

in which two or more variables may appear before the loop index, and in which the step size may be greater than 1. We note that a list of variables is also used, for example, in a FORTRAN READ statement such as

READ (5, 22) (A(I), B(I), I=1,99,2)

but here the semantics is slightly different; in particular, the semantic restriction made above is not applicable in this case, since a READ statement is executable and makes reference to current values of all variables.

5-4 Procedures and Parameters

There are two general forms of syntactic rules describing the interaction between a program and its subroutines. In the first, as exemplified by FORTRAN, programs are grouped into "collections of programs" which may be defined simply by

<collection of programs> ::= <program> | <collection of programs> <program>

The end of a collection of programs is normally an end-of-file card or something similar. In the second form, as exemplified by ALGOL, there is a hierarchical structure of procedures. There is an outermost procedure, and every procedure except the outermost is subordinate to some other procedure. Both these forms may be combined in the description of a single language, as in PL/I.

A program has an effect, which we wish to associate with the name of that program. In the simplest case, we have a function from procedure names into effects. This function is an attribute of any collection of programs, and is inherited by each program in the collection and ultimately by each expression in such a program. When an expression contains a function reference, the value of that reference, as a semantic attribute of it, is determined as a function of the state vector by evaluating the effect, as determined by the function described above, and then applying the resulting state vector to the function name as a variable. The effect itself becomes the side effect of the function reference. In a more general case, we can have a second function from procedure names into exit indices, and this may be referenced to find the exit index of the given function reference. (This extends the philosophy expressed at the beginning of section 4-3.)

In line with the discussion in section 5-1, we shall include both of the above functions as part of the environment. If y is the environment, then y(n, effect) will be the effect of the procedure with name n, and y(n, exit) will be the exit index of that procedure. The contribution to the environment which is made by a single subroutine is then specified by a rule such as

<subroutine> s ::= 'SUBROUTINE' <subroutine name> n <body> b
<local environment> $s^z = b^z \cup \{((n, \text{effect}), b^e), ((n, \text{exit}), b^x)\}$

where b^e and b^x are, respectively, the effect and the exit index of the body b of the subroutine, and b^z is the local environment of the body, including type, rank, and other information contributed by declarations in the subroutine. If we agree to use the superscript z for the local environment of a main program and of a function, as well as a subroutine, the synthesis of the local environment and the inheriting of the global environment may be given by rules such as

<complete collection of programs> ::= <collection of programs> c <end marker>
$c^y = c^z$

<collection of programs> x ::= <program> p; y ::= <collection of programs> z <program> q
<local environment> $x^z = p^z$; $y^z = z^z \cup q^z$
<environment>^{y}
$p^y = x^y$; $z^y = y^y$; $q^y = y^y$

<program> x ::= <main program> m; y ::= <function> f; z ::= <subroutine> s
<local environment> $x^z = m^z$; $y^z = f^z$; $z^z = s^z$
<environment>^{y}
$m^y = x^y$; $f^y = y^y$; $s^y = z^y$

In ALGOL, there is, as we have seen, a separate environment for each block. Procedure declarations in ALGOL are treated on the same basis as any other declaration. The collection of all the declarations in a block, together with the word begin at the beginning of the block, is called the block head. Its syntax and semantics are

<block head> x ::= 'begin' <declaration> d; y ::= <block head> z ';' <declaration> e

<local environment> $x^Z = d^Z$; $y^Z = z^Z \cup e^Z$

where declarations are described by

<declaration> w ::= <type declaration> t; x ::= <array declaration> a; y ::= <switch declaration> s; z ::= <procedure declaration> p

<local environment> $w^Z = t^Z$; $x^Z = a^Z$; $y^Z = s^Z$; $z^Z = p^Z$

The local environment p^Z of the procedure declaration p consists of pairs of the type ((n, effect), e) and ((n, exit), x), just as the local environment t^Z of the type declaration t, for example, consists of pairs of the type ((n, type), t).

Let us now turn our attention to parameters. A formal parameter in a program is somewhat like a variable of that program; in particular, it has a state vector component. However, when a formal parameter is used as a primary expression, its semantics are not the same as the semantics of an ordinary variable, unless the parameter is called by value. If the parameter is called by name, in the ALGOL sense, then, as a primary expression, it may have a side effect; in fact, it must have a side effect if the corresponding actual parameter does. Likewise, if the corresponding actual parameter makes any abnormal exit, then the formal parameter must be considered as

making an abnormal exit as well.

We shall model this behavior, semantically, by altering the form of the values of formal parameters. When a formal parameter is called by value, its values will be taken to be the values of the corresponding actual parameter. When a formal parameter is called by name, however, its value is an n-tuple, consisting of all of its necessary semantic attributes. If it appears as a primary expression, its value is an ordered triple (u, e, x), where u is its L-value as studied in section 3-2, e is its effect, and x is its exit index. The ordinary value is then obtained from the L-value in the manner described in section 3-2. This ordered triple of state vector functions determines the semantics of the formal parameter as a primary expression.

It is also, of course, possible for a formal parameter to be called by location, as in FORTRAN. Call by location does not lead to side effects or escape, but it must be treated differently from call by value. We shall consider the values of parameters called by location to be variables, that is, elements of the domain of the state vector functions under consideration. When such a parameter is used, the corresponding component of the current state vector becomes its L-value, from which the ordinary value is obtained just as before.

An actual parameter called by value as in ALGOL is evaluated once and for all, just prior to the subroutine or function call. There are other slightly different forms of call by value; for example, one of the FORTRAN compilers for the IBM 360 allows the user to call any parameter by value, but this value is returned at the end of the subroutine. Thus if α is the actual parameter and β the formal parameter, the compiler effectively sets $\beta=\alpha$ at the start of the subroutine -- but it also sets $\alpha=\beta$ at the end of the subroutine. (This eliminates

the most important disadvantage of call by value in ALGOL, namely that a parameter called by value does not have the intended effect if it is used on the left side of the assignment symbol.)

The correspondence between actual and formal parameters may be treated as assigning the formal paramet er to the actual one, no matter how the parameters are called. If they are called by value, the new value of the formal parameter becomes the value of the actual parameter. For call by location, the new value of the formal parameter is the actual parameter itself (as a variable). If that variable is subscripted, the new value is that variable which results from evaluating the subscripts and taking the element of the array with integer subscripts as found by evaluation. For call by name, the new value of the formal parameter is the n-tuple, as described above, of semantic attributes of the corresponding actual parameter. This assignment of formal to actual parameters may be considered as an executable statement which is executed just before the given subroutine is called (and, in the case of IBM 360 FORTRAN mentioned above, another executable statement executed afterwards). Like any executable statements, these may have abnormal exits, as, for example, when an actual parameter called by value is an expression involving a function which escapes.

In order to distinguish between parameters called in different ways, and between parameters and other variables, we may introduce a new keyword, pcm (for "parameter cal ling method"), analogous to type and rank. If v is any variable and y is the environment, then y(v, pcm) shall be name, loc, or value for v called respectively by name, by location, or by value, or none if v is not a parameter. A simplified rule for primary expressions, omitting all such expressions other than simple variables, may then be given as

<primary> p ::= <variable> v

<environment>^{y}

$v^{y} = p^{y}$

<value> $p^{v}(S) = S(v^{\ell}(S))$

<side effect> $p^{s}(S) = v^{s}(S)$

<exit index> $p^{x}(S) = v^{x}(S)$

<variable> v ::= <simple variable> s

<environment>^{y}

<L-value> $v^{\ell}(S)$ = if $v^{y}(s, \text{pcm}) = \text{name}$ then $u(S)$, where $S(s) = (u, e, x)$, else if $v^{y}(s, \text{pcm}) = \text{loc}$ then $S(s)$ else s

<side effect> $v^{s}(S)$ = if $v^{y}(s, \text{pcm}) = \text{name}$ then $e(S)$, where $S(s) = (u, e, x)$, else S

<exit index> $v^{x}(S)$ = if $v^{y}(s, \text{pcm}) = \text{name}$ then $x(S)$, where $S(s) = (u, e, x)$, else normal

The parameter calling method information is introduced into the environment in a straightforward manner. Any formal parameter v gives rise to the ordered pair ((v, pcm), loc) in FORTRAN, or ((v, pcm), name) in ALGOL. Any formal parameter v occurring on the value list in ALGOL (that is, following the keyword value) gives rise, in addition, to the ordered pair ((v, pcm), value), and the corresponding pair ((v, pcm), name) is removed from the local environment. Any ordinary variable v in either FORTRAN or ALGOL gives rise to the pair ((v, pcm), none).

The effect and the exit index of a procedure with parameters are functions of the list of actual parameters. When such a procedure is called, a list of actual parameters will be given, and to this list we apply these two functions, obtaining the effect and the exit index of the procedure call. When a procedure is defined, it has an effect and an exit index which do not depend on actual parameters, but which

involve the values of formal parameters. This effect and exit index are combined with those of the operations which initialize each formal parameter to the corresponding actual parameter; these latter operations, of course, are themselves functions of the actual parameter list. The result will be an effect and exit index of the procedure as applied to its own environment, and this must then be altered, as it is for blocks, to produce the effect and exit index of the procedure as applied to the environment of each procedure call. Note that initialization of a parameter called by name must always exit normally, whereas initialization of a parameter called by value involves evaluation of the corresponding actual parameter and therefore may not exit normally. Even if the procedure does nothing whatsoever, it may, in such a case, exit abnormally.

5-5 Storage Mappings

In some languages, it is possible to treat declarations as if they were executable statements. This is particularly true in block structure languages if we take into account the usual method of implementing such languages on computers. At any given time during execution of a program in such a language, the first k locations in memory, for some k, will be in use. When we enter a block, we encounter various declarations. Each of these requires the allocation of, let us say, j new cells of storage, and these are normally cells $k+1$ through $k+j$. The implementation will involve a variable (let us call it SIZE) whose value at any given time is the total number of cells in use at that time. In this case, the value of SIZE should be $k+j$ after the declaration, where it was k before the declaration. Thus we may think of executing any declaration by increasing the value of SIZE by the number of calls which that declaration requires.

It is expected, in addition, that executing a declaration will change the current assignment of program variables to storage cells. This assignment is called the storage mapping; it is a function from variables to cells. If the value of SIZE is k, a declaration allocating j new cells will augment the current storage mapping so that it maps certain variable names into the cell numbers $k+1$ through $k+j$. Each cell has a current value at any time, and the function from cells to their current values is called the content of the store. The current value of any variable, in this situation, is found by reference to both the current storage mapping and the current content of the store. If M is the set of all variables of the program, C the set of all storage cells, and V the set of all possible values, then the current storage mapping is a function $\alpha: M \rightarrow C$ and the current content

of the store is a function $\sigma: C \to V$, while the current value of any variable $x \in M$ is $\sigma(\alpha(x))$.

Entry into a block and exit from a block, in this situation, must also be treated as executable statements. When we exit from a block through its end statement, SIZE must be decreased by the amount of space allocated for that block. This amount may change from one entry into the block to another, particularly if the block contains array declarations with variable array bounds. The amount of space allocated for any block is thus the value of a second program variable (let us call it BSIZE). Every time a declaration is executed, SIZE and BSIZE are both increased, and by the same amount. In addition, BSIZE must be initialized to zero at the beginning of the block, which makes entry into any block executable as well. At the beginning of the outermost block, both SIZE and BSIZE are initialized to zero.

When exit is made from two or more blocks in succession, SIZE must be decreased by the sum of several quantities, all of which are essential components of the current state of the computation. If we are inside several blocks, the value of BSIZE for each of these blocks must be stacked. Experience with block structure languages shows that only one stack is necessary, and we may treat it as a variable whose values are sequences of those things that are currently stacked. Such a sequence may be null, in which case we denote the value of the stack by nil. If the stack is called STACK, the effect of pushing down X is given by

$$e(S) = S', \text{ where } S'(z) = S(z) \text{ for } z \neq \text{STACK}$$
$$\text{and } S'(\text{STACK}) = \underline{\text{if}}\ S(\text{STACK}) = (x_1, x_2, \ldots, x_n)\ \underline{\text{then}}\ (X, x_1, x_2, \ldots, x_n)\ \underline{\text{else}}\ (X)$$

In the "else" case, of course, it is assumed that S(STACK) = nil. The effect of popping up X is then given by

$e(S) = S'$, where $S'(z) = S(z)$ for $z \neq X$, STACK, and

$S'(X) = x_1$, $S'(STACK) = (x_2, \ldots, x_n)$,

where $S(STACK) = (x_1, x_2, \ldots, x_n)$

and this operation is undefined if $S(STACK) =$ nil.

Using this concept of stacking, we may now define the effects of the operations associated with a block. We assume that the current storage mapping is a variable, SMAP, whose value is a set of pairs. That portion of the current storage mapping which represents a contribution from the declarations in the current block will be given as another variable, BMAP, of the same form as SMAP. The value of BMAP or of SMAP may be denoted by nil if it is the null set of pairs. In the absence of multiple use of names, the effects of entry into a block, of a typical declaration, and of normal exit from a block may be described as follows:

Begin: push BMAP; BMAP:=nil; push BSIZE; BSIZE:=0

integer x: SIZE:=SIZE+1; SMAP:=SMAP $\cup$ {(x, S(SIZE))};
BSIZE:=BSIZE+1; BMAP:=BMAP $\cup$ {(x, S(SIZE))}

End: SIZE:=SIZE-BSIZE; pop BSIZE; MAP:=MAP-BMAP
(where the minus sign denotes the difference of sets of pairs, that is, removal of all pairs in BMAP from those in MAP); pop BMAP

If multiple use of names is permitted, we may remove from SMAP any pair which conflicts with a newly inserted pair at the time of insertion (that is, when executing a declaration) and add this pair to another set of pairs called UMAP (U for "unused"). The operation MAP:=MAP-BMAP now becomes MAP:=(MAP-BMAP) $\cup$ UMAP. When we begin a block, we push UMAP and set it to nil; when we end a block, we pop UMAP. Care must be taken, of course, to pop quantities in reverse

order from the order in which they were pushed.

The effect of an abnormal exit from a block depends upon the block level of the exit point. An exit from a block by going to the next outer block level is treated exactly as a normal exit; the effect (as outlined above) then becomes a part of the side effect of the conditional or unconditional transfer. If transfer is made to a point which is n block levels outside the current level, this must be treated as n normal exits in succession, from innermost to outermost. ALGOL does not allow transfer into a block, but where this is allowed (as, for example, in the algebraic language CPL) it is treated as a succession of entries into blocks, from outermost to innermost, or as a single entry into a block for a transfer into the block level immediately inside the current one.

Executable statements, in the presence of a storage mapping, have effects much like those they would have otherwise. If the current storage mapping is α, the new storage mapping is α', the current content of the store is σ, and the new content of the store is σ', we may write $e(\alpha, \sigma) = (\alpha', \sigma')$, where e is the effect of an executable statement. In ALGOL, it is always assumed that $\alpha' = \alpha$, unless the executable statement transfers to an outer block level. Even if the execution of the statement involves evaluation of an expression which contains a function reference that changes the storage mapping, normal exit from the referenced function is expected to reset the storage mapping to what it was before the reference is made. In PL/I, on the other hand, there is a function called ALLOCATE, which may be called at any time, and whose purpose is effectively to augment the current storage mapping; there is then another function, called FREE, which undoes what ALLOCATE does.

The idea that block entry and exit, as well as declarations, should be executable is quite well known to those who design com-

pilers and interpreters for block structure languages. It is well known, for example, that whereas GO TO α in FORTRAN may be compiled into a single unconditional transfer instruction, the corresponding statement in ALGOL is true only if α is at the current block level. If it is not, the object code must handle exit from one or more block levels. We have preferred, however, to view this aspect of block structure languages as basically concerned with the implementation (although it is not implementation-dependent, since at least every known implementation must take account of it). Thus, in the semantic models we have constructed, we have sought to avoid treating block entry and exit as being executable. In fact, as we have seen, this is possible if each block is regarded as if it were a single statement, which has an effect and an exit index. Under these conditions, it is not necessary to introduce a storage mapping into the model. However, if a storage mapping is introduced, and if we are willing to accept the consequences of doing so (that is, treating block entry and exit, and declarations, as executable statements in the manner suggested above), it becomes easier for us to model certain advanced features of block structure languages, as will now be described.

In ALGOL, arrays are permitted to have dimensions (that is, lower and upper subscript bounds) which are variable, and which may, in fact, change during a single run. Consider the following program:

```
begin integer k, i; ininteger(k); if k=0 then go to done;
begin integer array a[1:k];
for i:=1 step 1 until k do ininteger(a[i]);
sort(a, k);
for i:=1 step 1 until k do outinteger(a[i]) end;
done: end
```

This program reads in, sorts, and writes out one or more collections of integers, each of which is preceded by its length. The procedures ininteger(x) and outinteger(x) respectively read and write the integer x, while the procedure sort(a, k) sorts the array a of length k. Suppose now that we denote by e the effect of the sort procedure. What collection of state vectors constitutes the domain and range of e? If we write e(S) = S', then S and S' must have an a[i]-component for each i, $1 \leq i \leq k$. However, since the various collections of integers may have different lengths, corresponding to different values of k, it is not clear how to specify, once and for all, a set of state vectors upon which the effect e acts. One solution is to require all such state vectors to have an a[i]-component for every integer i, and to require further that the a[i]-component have the value Ω (see section 5-3) whenever i is currently out of range. However, this slight subterfuge must then be repeated for every other such array. If a storage mapping is used, the execution of the declaration integer array a[1:k] causes k cells to be allocated in storage. These cells may be different for different entrances to the block, even if arrays with variable bounds are not declared in the block, if such arrays are declared in some outer block. The content of the store, in this situation, is always a mapping from the set C of storage locations to the set V of values.

Sharing of temporary storage is explicitly specified as part of the model if a storage mapping is used. Suppose that two subroutines (procedures) are called, and each of these involves temporary variables. It is customary, in implementations of block structure languages, for these temporary variables to occupy the same positions in memory. (To be truly temporary, such variables must not, in ALGOL, be declared as own.) In our semantic models, we have not taken account

of this sharing. If a storage mapping is used, however, then, when the first subroutine is entered, certain cells are allocated, and when exit is made from that subroutine, these cells are released. When the second subroutine is entered, the same cells are allocated again. Of course, they may be allocated in different ways; they may now contain real numbers, for example, where previously they contained Boolean quantities or integers. In the storage-mapping model, the set of values must be a universal set, which contains anything that could conceivably be a value of something.

Multiple use of names may also be handled "cleanly" using a storage mapping, even in the presence of recursion. Whenever a name is re-used, whether it was previously used in a different block or procedure or recursively in the same procedure, the preceding use of the name is effectively stacked. If the re-use is recursive, this can happen to an arbitrary number of recursion levels. Similarly, upon exit from a procedure, whether this exit reduces the recursion level or not, the preceding use of each name re-used in that procedure is recovered from the stack.

Notes

The semantics of declarations are discussed briefly in [Strachey 66] and in [Burstall 70]. The term "environment" is used in [Landin 64], in a slightly different sense than we use it; an environment, in Landin's sense, includes value information, that is, it is more like a state vector than like our concept of environment. In the Vienna method, the type, rank, parameter calling method, and so on, of a variable are attributes (in the PL/I sense) of that variable, but the (current) value is also. No distinction is made in the Vienna method between attributes that change (such as the current value) and attributes that normally do not (such as the rank). In APL, in fact, the rank of a variable can change as a program runs.

Our term "state vector domain" is reminiscent of the "domain of interpretation of a program scheme" as studied in [Rutledge 64] and in [Luckham, Park and Paterson 70], and the input, program, and output domains of [Manna 69]. The word "domain" unfortunately has two meanings in mathematics; we use it here in its first meaning, as (apparently) synonymous with "set" (compare also "alphabet" and "universe") rather than as the specific set of all first elements of the set of ordered pairs specifying a function, that is, the set of all x for which f(x) is defined (for the domain of f).

Arrays may be thought of as single variables whose values are sequences of elements; we have preferred to think of them as sequences of variables. Our treatment of initializing declarations seems rather obvious, but the subject seems to have excited little interest, although such declarations are mentioned in passing in [Strachey 66]. The use of a special symbol to denote "undefined" is part of McCarthy's theory of computation, and appears, for example, in [McCarthy 65].

A partial function which may be extended to a total function by defining it as Ω (or "undefined") on a recursively definable set acts more like a total function, from the viewpoint of mathematical logic, than like a partial function (an example of such a function is the effect of a statement which makes reference to a subscripted variable). Procedures and parameters are treated axiomatically in [Hoare 71]; they are also mentioned in [Strachey 66], in which our "call by location" is referred to as "call by reference."

The concept of a storage mapping, and the use of state vectors which are mappings from a set of variables through a set of locations to a set of values, has been studied by quite a number of people. It appears first, informally, in [Strachey 66], and more explicitly in [Park 68], where the state of a computation is defined to be a pair $\langle \mathcal{L}, C \rangle$; here $\mathcal{L}$ maps the currently legal expressions (among which are the variables) into some set of locations, while C maps each location into its current value. Similarly, [Kaplan 68] defines the program state vector of a program π to be the ordered pair (M_π, ξ), M maps program variables into the positive integers and ξ maps positive integers into values. The "unique names" of [Burstall 70] also bear a certain resemblance to an intermediate set of locations.

EXERCISES

1. In COBOL, the type of a variable is called its picture. The picture is a string of characters, each of which is either 9, X, or Z (and possibly others, which we shall ignore for the moment). If the length of this string is n, the length of each value of the given variable must be n, and each character in such a value must be numeric, alphanumeric, or alphabetic if the corresponding character of the picture is respectively 9, X, or Z. Thus, for example, if the picture of D is 'XX99Z', then 'AB12J', 'Y511Y', and '2256A' are legal values of D, but 'GH248', 'CLOUD', and 'TR2' are not.

Let us introduce picture information into the environment of a variable by using the keywords picture (analogous to type) and length. If v is a variable and y is the environment, then y(v, length) should be the length of the picture of v, and, if y(v, length) = n, then y(v, picture, i), for $1 \leq i \leq n$, should be the set of all these characters which are allowable as the i-th character of v.

Variables may be given pictures by a SIZE clause (SIZE IS n, where n is an integer giving the length of the picture), a CLASS clause (CLASS IS NUMERIC, CLASS IS ALPHANUMERIC, or CLASS IS ALPHABETIC, in each case specifying all characters of the picture to be respectively 9, X, or Z), or a PICTURE clause (PICTURE IS s, where s is a character string to be used as the picture). Formulate syntactic and semantic rules which specify the picture information to be included in the local environment in each of the above cases. The word IS may always be omitted from any of the above clauses. Make the simplifying assumption that either PICTURE alone, or both SIZE and CLASS (in that order), must be specified.

2. Consider the simplified COBOL syntactic rule

<ADD statement> s ::= 'ADD ' <variable> a ' TO '
<variable> b ' GIVING ' <variable> c

Here, for example, ADD X TO Y GIVING Z sets Z equal to the sum of X and Y. Let s^y(v, length) and s^y(v, picture, i) be as defined in problem 1 above, where v is any of a, b, and c.

(a) Formulate a semantic restriction on pictures which assures that no alphabetic characters can ever be added in the above rule.

(b) Formulate a semantic restriction which insures that the picture of the variable c can never be too short to hold the result. (Note: This restriction is relaxed in actual COBOL systems; overflow is held to be a run-time error.)

(c) In part (b) above, what happens if ADD is replaced by MULTIPLY? By DIVIDE?

(d) Consider the simplified COBOL syntactic rule

<MOVE statement> s ::= 'MOVE ' <variable> a ' TO ' <variable> b

which is such that MOVE X TO Y sets the new value of Y equal to the value of X. Using the terminology above, formulate a semantic restriction which insures that the new value of Y will always conform to the picture of Y. (Note that it should be allowed, for example, to move a numeric quantity into an alphanumeric character position, but not vice versa.)

3. Generalize the rule for subscript lists given in section 5-2, so as to allow side effects, escape, and real subscripts which are converted to integer form by truncation.

4. Suppose that we wish to regard array names as variables whose values are sequences. For example, if A is given in FORTRAN by

DIMENSION A(4), the values of A are to be sequences (v_1, v_2, v_3, v_4), where v_i is the current value of A(i) for $1 \leq i \leq 4$. Each such sequence may be regarded as a function f with domain $\{1, 2, 3, 4\}$, where $f(i) = v_i$ for each i.

(a) Formulate a semantic rule to accompany the first syntactic rule of section 5-2, which defines the type of the variable v to be the set of all functions f as outlined above.

(b) Same as (a) above, with the second semantic rule of section 5-2 (for multiple arrays). The set of n-tuples of the subscript bound list should here be the domain of the functions corresponding to the determined sequences.

(c) Formulate a semantic rule for the value (not the L-value) of a subscripted variable involving a general subscript list, as defined in section 5-2, in accordance with (b) above.

(d) If the values of array names are sequences, it is necessary to treat assignments to unsubscripted variables and assignments to subscripted variables in different ways. In words, explain why.

5. Explain the extensions which would have to be made to the syntactic and semantic rules for machine instructions given in section 3-5 in order to take into account space-reserving and initializing declarations of the forms

k BSS n

k DATA n

where k is a label and n is an integer. The first of these reserves n cells (for an array requiring n words, for example); the second reserves one cell and gives it the initial value n.

6. It is quite common, in the study of programming languages, to regard variables as having properties such as type and rank, and

then to regard the value of a variable as a property of it in the same sense that its type and rank are. Suppose that in our formulation of the environment y of a variable v we used the keyword value, so that y(v, value) is the value of v. What would this imply about v? Could we use this for the values of all variables?

7. Suppose that the function f(x, y) is defined by if $x \neq 0$ then output(y). Suppose that we now call f as a procedure by writing f(0, g(x)), where g(x) is a function which outputs x. When f(0, g(x)) is called, will x be output or not? (Note: The answer to this question depends on whether x and y are called by value, by location, or by name. Discuss.)

8. The following parameter calling method, which we shall refer to as delayed call by value, has been proposed. At the start of any procedure, a flag is set to zero, corresponding to each parameter called in this manner; the value of the actual parameter is not calculated. If, in the body of the procedure, the value of the parameter is needed, the flag is tested. If it is zero, the value of the formal parameter is set to the value of the actual parameter, and the flag is set to one. If it is one, the value of the formal parameter has presumably already been calculated in this manner, and hence it is simply retrieved.

(a) How does this proposed method affect the answer to the preceding question?

(b) Discuss the speed of the proposed method as compared with that of calling by name.

9. Suppose that a stack is represented by an array called STACK of size n, together with a single variable LSTACK whose value is the

current length of the stack, which is an integer between 0 and n inclusive. Under these conditions, describe the effect of pushing X and the effect of popping X. Also describe their exit index, given that exit is to be made to STACKERROR if pushing down causes stack overflow or if an attempt is made to pop an empty stack; otherwise exit is normal. Assume that the statement index of STACKERROR is i.

10. (a) Extend the specification given in section 5-5 of the effect of the declaration integer X to integer arrays, including multiple arrays. Assume that each variable or array is given by a separate declaration.

(b) In the treatment of declarations as executable statements, are any special difficulties caused by the fact that a variable name may be re-used with change in rank (for example, that it may be a two-dimensional array name inside an inner block, and a one-dimensional array name outside that block)? Explain.

REFERENCES

Backus 59 — Backus, J. W., The syntax and semantics of the proposed international algebraic language of the Zurich ACM-GAMM Conference, Proc. Internat. Conf. on Inf. Processing, UNESCO, 1959, pp. 125-132.

Burstall 70 — Burstall, R. M., Formal description of program structure and semantics in first order logic, Machine Intelligence 5 (1970), pp. 79-98.

Cooper 69 — Cooper, D. C., Program scheme equivalences and second-order logic, Machine Intelligence 4 (1969), pp. 3-15.

de Bakker 69 — de Bakker, J. W., Semantics of programming languages, in Advances in Information Systems Science 2, Plenum Press, New York-London, 1969, pp. 173-227.

Elgot and Robinson 64 — Elgot, C. C., and Abraham Robinson, Random-access stored-program machines, an approach to programming languages, Journal of the ACM 11, 4 (1964), pp. 365-399.

Elgot 68 — Elgot, C. C., Abstract algorithms and diagram closure, in Programming Languages, F. Genuys, ed., Academic Press, London-New York, 1968, pp. 1-42.

Engeler 67 — Engeler, E., Algorithmic properties of structures, Mathematical Systems Theory, 1, 3 (1967), pp. 183-195.

Ershov 60 — Ershov, A. P., Operator algorithms, I, Problems of Cybernetics 3 (1960), pp. 697-763.

Feldman and Gries 68 — Feldman, J. A., and D. Gries, Translator writing systems, Communications of the ACM 11, 2 (1968), pp. 77-113.

Floyd 63 — Floyd, R. W., Syntactic analysis and operator precedence, Journal of the ACM 10 (July 1963), pp. 316-333.

Hoare 71 — Hoare, C. A. R., Procedures and parameters: an axiomatic approach, in Lecture Notes in Mathematics 188, Semantics of Algorithmic Languages, Springer-Verlag, Berlin-Heidelberg-New York, 1971, pp. 102-116.

Igarashi 71 — Igarashi, S., Semantics of ALGOL-like statements, in Lecture Notes in Mathematics 188, Semantics of Algorithmic Languages, Springer-Verlag, Berlin-Heidelberg-New York, 1971, pp. 117-177.

Kaphengst 59 — Kaphengst, H., Eine abstrakte programmgesteuerte Rechenmaschine, Zeitschr. f. math. Logik und Grundlagen d. Math. 5 (1959), pp. 366-379.

Kaplan 68 — Kaplan, D. M., Some completeness results in the mathematical theory of computation, Journal of the ACM 15, 1 (1968), pp. 124-134.

Knuth 68 — Knuth, D. E., Semantics of context-free languages, Mathematical Systems Theory 2, 2 (1968), pp. 127-145.

Knuth 71 — Knuth, D. E., Semantics of context-free languages -- correction, Mathematical Systems Theory 5, 1 (1971), pp. 95-96.

Landin 64 — Landin, P. J., The mechanical evaluation of expressions, Computer Journal 6, 4 (Jan. 1964), pp. 308-320.

Landin 66 — Landin, P. J., The next 700 programming languages, Communications of the ACM 9, 3 (1966), pp. 157-166.

Lucas and Walk 69 — Lucas, P., and K. Walk, On the formal description of PL/I, Annual Review in Automatic Programming 6, 3 (1969).

Luckham, Park and Paterson 70 — Luckham, D. C., D. M. R. Park, and M. S. Paterson, On formalized computer programs, J. Computer and Systems Sci. 4, 3 (1970), pp. 220-249.

Manna 69 — Manna, Z., The correctness of programs, J. Computer and Systems Sci. 3, 2 (1969), pp. 119-127.

Maurer 66 — Maurer, W. D., A theory of computer instructions, Journal of the ACM 13, 2 (1966), pp. 226-235.

Maurer 72 — Maurer, W. D., A semantic extension of BNF, International Journal of Computer Mathematics, Sept. 1972.

Mazurkiewicz 71 — Mazurkiewicz, A. W., Proving algorithms by tail functions, Inf. and Control 18, 3 (1971), pp. 220-226.

McCarthy 63 — McCarthy, J., Towards a mathematical science of computation, Information Processing 1962, Proc. of IFIP Congress 62, North-Holland, Amsterdam, 1963, pp. 21-28.

McCarthy 65 — McCarthy, J., Problems in the theory of computation, Proc. of IFIP Congress 65 (Washington), Spartan Books, 1965, pp. 219-222.

McCarthy 66 — McCarthy, J., A formal description of a subset of ALGOL, in Formal Language Description Languages for Computer Programming, T. B. Steel, Jr., ed., North-Holland, Amsterdam, 1966, pp. 1-12.

McKeeman 66 — McKeeman, W. M., An approach to computer language design, Technical Report CS-48, Computer Science Dept., Stanford University, August 1966.

Naur et al. 60 — Naur, P., et al., Report on the algorithmic language ALGOL 60, Communications of the ACM 3 (1960), pp. 299-314.

Naur et al. 63 — Naur, P., et al., Revised report on the algorithmic language ALGOL 60, Communications of the ACM 6 (Jan. 1963), pp. 1-17.

Naur 66 — Naur, P., Proof of algorithms by general snapshots, BIT 6, 4 (1966), pp. 310-316.

Park 68 — Park, D. M. R., Some semantics for data structures, Machine Intelligence 3 (1968), pp. 351-371.

Perlis and Samelson 59 — Perlis, A. J., and K. Samelson, Report on the algorithmic language ALGOL, Num. Math. 1 (1959), pp. 41-60.

Podlovchenko 62 — Podlovchenko, R. I., On transformations of program schemes and their application to programming, Problems of Cybernetics 7 (1962), pp. 161-188.

Rutledge 64 — Rutledge, J. D., On Ianov's program schemata, Journal of the ACM 11, 1 (1964), pp. 1-9.

Scott 70 — Scott, D., Outline of a mathematical theory of computation, Proc. 4th Annual Conf. on Information Sciences and Systems, Princeton, 1970, pp. 169-176.

Strachey 66 — Strachey, C., Towards a formal semantics, in Formal Language Description Languages for Computer Programming, T. B. Steel, Jr., ed., North-Holland, Amsterdam, 1966, pp. 198-216.

Wagner 68 — Wagner, E. G., Bounded action machines: toward an abstract theory of computer structure, J. Computer and System Sci. 2, 1 (1968), pp. 13-75.

Wirth and Weber 66 — Wirth, N., and H. Weber, EULER: a generalization of ALGOL, and its formal definition, Communications of the ACM 9 (Jan.-Feb. 1966), pp. 13-25, 89-99.

I N D E X