70-25,836

WRIGHT, Jr., Charles Thomas, 1941-
A METHOD FOR THE COMPLETE DEFINITION OF PROGRAMMING LANGUAGES.

Iowa State University, Ph.D., 1970
Computer Science

University Microfilms, A XEROX Company, Ann Arbor, Michigan

THIS DISSERTATION HAS BEEN MICROFILMED EXACTLY AS RECEIVED

A METHOD FOR THE COMPLETE DEFINITION OF PROGRAMMING LANGUAGES

by

Charles Thomas Wright, Jr.

A Dissertation Submitted to the
Graduate Faculty in Partial Fulfillment of
The Requirements for the Degree of
DOCTOR OF PHILOSOPHY

Major Subject: Computer Science

Approved:

Signature was redacted for privacy.

In Charge of Major Work

Signature was redacted for privacy.

Head of Major Department

Signature was redacted for privacy.

Dean of Graduate College

Iowa State University
Ames, Iowa

1970

TABLE OF CONTENTS

		Page
I.	INTRODUCTION	1
II.	SEMANTIC FUNCTIONS FOR CONTEXT-FREE LANGUAGES	4
III.	COMPLETE DEFINITION OF A SIMPLE PROGRAMMING LANGUAGE	22
IV.	DISCUSSION	54
V.	BIBLIOGRAPHY	57
VI.	ACKNOWLEDGMENTS	59

I. INTRODUCTION

Much work has been done in recent years in the area of programming languages. This work has concentrated on three aspects; syntax, semantics, and pragmatics. A complete definition of a programming language essentially consists of a definition of its syntax, semantics, and pragmatics. The complete definition defines the constructions that are the legal sentences in the language (syntax), it defines the meaning of each legal sentence in the language (semantics), and it conveys this meaning to the interpreter of the language (pragmatics). The definition should ideally be unambiguous.

A complete definition of a programming language is not necessarily a useful one. To be useful a complete definition needs to satisfy certain addition criteria. In particular:

(1) It should provide for the isolation and clarification of the various components of the language.

(2) It should lend itself well to a systematic study of the language.

(3) It should readily suggest a means of constructing a translator for the language.

(4) It should provide a clear teaching device for the language.

(5) It should lend itself well to the design and definition of other, similar programming languages.

This dissertation is an investigation of a method for the complete definition of programming languages. It is doubtful that this method satisfies all the above criteria for all programming languages, but it is hoped that it is a step toward better definitions of programming languages. The method lends itself particularly well to the teaching and clarification of programming language concepts.

Chapter II begins with a discussion of the role that context-free grammars play in the definition of formal languages. It is shown how a context-free grammar can be used to specify the syntax of a particular language and also to specify a certain amount of meaning for various sentences in the language. In particular, it is shown how the syntactic rules in a grammar can be augmented by certain semantic rules with the augmented grammar thus giving a syntactic and semantic description of the language involved. Particular attention is given to an approach developed by Knuth (5) whereby the meaning of each nonterminal symbol, X, in a grammar is defined in terms of both successor and predecessor symbols of X in the grammar.

Chapter III shows that Knuth's approach actually works for a simple programming language. The language is first

defined syntactically by a context-free grammar. A simple computer is introduced and semantic functions are defined and inserted into the grammar. The resulting augmented grammar is given as a complete definition of the language. Two examples are given and worked out in detail to illustrate the use of the definition.

Chapter IV contains a discussion of the strengths and weaknesses of this method of language definition. Particular emphasis is placed on the use of this method in the teaching of programming languages.

II. SEMANTIC FUNCTIONS FOR CONTEXT-FREE LANGUAGES

There has been much work done on grammars (often called formal grammars) and the languages (often called formal languages) they define. As a result, there is a certain diversity in the definitions and notation appearing in the literature. The survey paper by Feldman and Gries (2) has been used as a guide in formulating the definitions and notation used in this dissertation. These definitions are:

<u>Definition 1</u>

An alphabet is any finite nonemply set.

<u>Definition 2</u>

A word over an alphabet A is a finite sequence of elements of A. A word has length (possibly zero) and is formed by concatenating elements of A. The word of length zero is called the empty word and is denoted by ε. The set of all words, including ε, over an alphabet A is denoted by A^*.

<u>Definition 3</u>

A phrase-structure grammar is a 4-tuple $G = \langle V_N, V_T, P, S\rangle$, where:

1) V_N = a nonterminal alphabet
2) V_T = a terminal alphabet
3) P = a set of composition rules
4) S = a starting symbol.

Elements of P are called productions and each element

of P is of the form $u \to v$ where u is a word (of positive length) in $(V_N \cup V_T)^*$ and v is a word in $(V_N \cup V_T)^*$. S is an element of V_N.

Definition 4

A context-free grammar is a phrase structure grammar in which each production in P is of the form $u \to v$, where u is in V_N and v is of positive length and is in $(V_N \cup V_T)^*$. It will be assumed that each element of V_N appears as the left part of at least one production (i.e. as the u is a rule $u \to v$). Note that V_T consists of exactly those symbols (in the alphabet) that appear only on the right sides of rules in the grammar.

Definition 5

For strings u, v in $(V_N \cup V_T)^*$ u directly produces v if (and only if) there exist strings x and y both in $(V_N \cup V_T)^*$ and a rule $t \to w$ in P such that

$u = xty$

and $v = xwy$.

This is denoted by $u \Rightarrow v$ and v is considered as a direct derivative of u.

Definition 6

The string u produces v (denoted $u \overset{*}{\Rightarrow} v$) if there exist strings $x_0, x_1, \ldots, x_n$ such that $u \Rightarrow x_0 \Rightarrow x_1 \Rightarrow \ldots x_{n-1} \Rightarrow x_n = v$. The string v is called a derivative of u and the sequence $x_0, x_1, \ldots, x_n$ is a derivation of v.

<u>Definition 7</u>

Any string that can be derived from the starting symbol, S, is called a sentential form.

<u>Definition 8</u>

The language generated by a grammar G is defined as:

$$L(G) = \{x \mid S \overset{*}{\Rightarrow} x \text{ and } x \text{ is in } V_T^*\} .$$

Informally, L is the set of all sentential forms that contain only terminal symbols.

<u>Definition 9</u>

Given a sentential form x of a grammar G. We say that w (w in $(V_N \cup V_T)^*$) is a phrase in x if there exist strings y, w, z all in $(V_N \cup V_T)^*$ and a symbol t in V_N such that:

1) $x = ywz$
2) either $S \overset{*}{\Rightarrow} ytz$ or $S = ytz$
3) $t \overset{*}{\Rightarrow} w$.

<u>Definition 10</u>

A derivation of a sentence in some language is called a parse of that sentence.

A simple example is given to clarify these definitions:

Grammar A

$S \rightarrow E$

$E \rightarrow T$

$E \rightarrow E + T$

$T \rightarrow P$

$T \rightarrow T * P$

$P \rightarrow (E)$

$P \rightarrow I$

$I \rightarrow a$

$I \rightarrow b$

$I \rightarrow c$

$I \rightarrow d$

$V_N = \{S, E, T, P, I\}$

$V_T = \{+, *, (,), a, b, c, d\}$

Starting symbol = S

Some sample sentences in L(A) are: "a", "a + b", "a*(b+c)", "a + b*c". Actually, L(A) consists of arithmetic expressions involving addition and multiplication.

The sentential form "P + T * P" has the following phrases: "P + T * P", "T * P", "P" (the leftmost P).

A parse of a sentence in a language can be represented by a syntax tree for that sentence. The syntax tree can be grown by starting with the starting symbol of the grammar and essentially growing branches representing each step of a

derivation of the sentence.

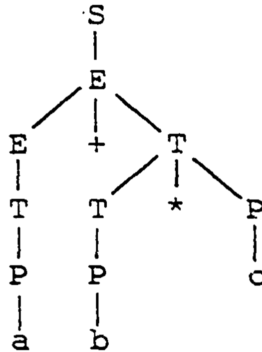
The following example illustrates this process:

A derivation for the string "a + b * c" is

$$S \Rightarrow E \Rightarrow E + T \Rightarrow T + T \Rightarrow P + T \Rightarrow a + T \Rightarrow$$

$$a + T * P \Rightarrow a + P * P \Rightarrow a + b * P \Rightarrow a + b * c$$

The syntax tree for this string is:



This tree is grown a step at a time. The first two steps are:

The parse, or syntax tree, of a sentence illustrates the phrase structure imposed upon that sentence by the grammar generating it. The use of a context-free grammar to impose a phrase structure upon each sentence in the language it defines is basic to the work done in this dissertation.

It should be noted that the sentences in a formal language are merely abstract objects. For such a language to

have meaning (and be useful) it is assumed that a relationship exists between the abstract objects and "concrete" objects whose meaning is well-understood. This relationship serves to give meaning to each abstract object in terms of the concrete objects. The job of importing meaning to a formal language consists of making this relationship explicit.

A natural way to start is to let the phrase structure imposed upon a formal language by its grammar correspond as closely as possible to the "structure" of the language in the "concrete" world.

As noted earlier, Grammar A formally defines a set of abstract objects called arithmetic expressions. There are many other grammars that also formally define this same set of abstract objects. The grammar below is one of them:

<u>Grammar</u> <u>B</u>

S → E

E → T

E → E * P

T → P

T → T + P

P → (E)

P → I

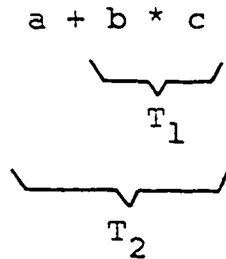
I → a

I → b

I → c

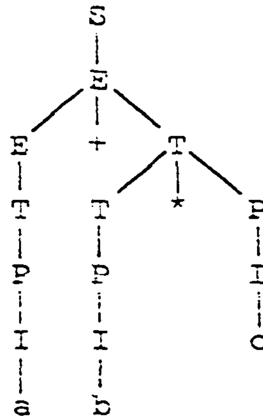
I → d

Let us now examine the abstract object "a + b * c" where "a", "b", "c", "+", "*" are all abstract symbols and relate this object to the "concrete" object "a + b * c" where "a", "b", and "c" are numbers and the "+" and "*" are the arithmetic operators for addition and multiplication. To compute the value of the arithmetic expression "a + b * c" we know we first multiply "b" by "c" and then add this result to "a". We can diagram this process as:

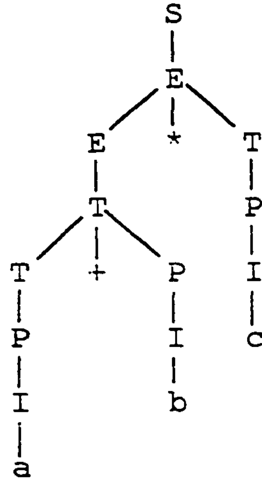


Let us compare this with the two syntax trees we get (for this sentence) from Grammar A and Grammar B.

From Grammar A we get:



From Grammar B we get:



In Grammar A the phrase "T * P" corresponds very nicely to the "b * c" in the actual evaluation of the expression. Similarly, the phrase "E + T" corresponds nicely to the "a + T_1" in the evaluation process.

However, in Grammar B we get no such nice correspondence. We have a phrase "T + P" which we would like to associate with the "a + b" but in the evaluation of the expression we do not add b to a. Actually, Grammar B "suggests" that "a + b * c" be evaluated as:

$$\underbrace{\underbrace{a + b}_{T_1} * c}_{T_2}$$

which is contrary to our "natural" order of evaluation for this expression.

Thus, the structure imposed upon the sentence "a + b * c" by Grammar A serves to illuminate its meaning. On the other hand, the structure that Grammar B imposes upon that same sentence serves, if anything, to hide the meaning of the sentence.

As Wirth and Weber (11) point out, it is natural that the definition of the structure (i.e., the syntax) of a language and the definition of the meaning (i.e., the semantics) of a language be closely related since structural orderings are merely an aid to understanding a sentence. The relationship between syntax and semantics can be made explicit by the insertion of semantic functions into the grammar defining a language. A set of semantic functions can be inserted for each production in the grammar with the semantic functions defining the meaning of the language in much the same way as the syntactic rules define the structure of the language.

An example, using Grammar A, will illustrate this. In Grammar A, the non-terminal symbols in the grammar were:

E - standing for expression

T - standing for term

P - standing for primary

I - standing for identifier.

The terminal symbols in the grammar were:

+ - standing for addition

* - standing for multiplication

(- left grouping symbol

) - right grouping symbol

a,b,c,d - particular symbols standing for integral numbers.

Grammar A tells us how to form arithmetic expressions. We want our semantic functions to tell us how to interpret the meaning of each arithmetic expression formed by Grammar A.

There are many kinds of semantic functions that can be inserted into Grammar A to give a reasonable meaning to each arithmetic expression formed from Grammar A. The particular semantic functions chosen in the example below reflect only one choice. There may be others.

The following conventions are used:

1) For each non-terminal symbol, X, let V(X) denote a semantic function associated with a production whose left part is X.
2) Each semantic function is listed using the convention that the right-hand side of each function is the definition of the left-hand side.
3) The special symbols, "$\oplus$" and "$\otimes$", stand for addition and multiplication respectively.

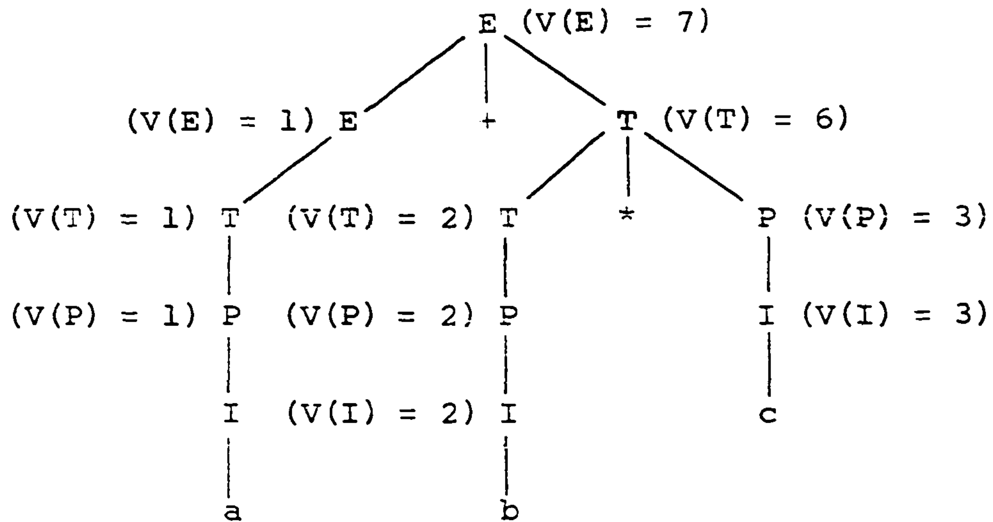
Let us augment Grammar A as follows:

Syntatic Rules	Semantic Functions
E → T	V(E) = V(T)
E → E + T	V(E) = V(E) ⊕ V(T)
T → P	V(T) = V(P)
T → T * P	V(T) = V(T) ⊛ V(P)
P → (E)	V(P) = V(E)
P → I	V(P) = V(I)
l → a	V(I) = 1
I → b	V(I) = 2
I → c	V(I) = 3
I → d	V(I) = 4

The semantic functions defining V(I) in each of the last four productions were chosen arbitrarily. Other numbers could have been chosen.

Assume an arithmetic expression is given and its meaning desired. A syntax tree can be constructed for the expression and the tree can be augmented by inserting the corresponding semantic function into the tree for each production that was used in the construction of the tree. The evaluation of the semantic function associated with the top node of the tree gives the meaning of the expression.

The augmented syntax tree for the expression "a + b * c" is:



The above example, following roughly the work of Irons (3), used only one semantic function, V(X), with each syntactic rule. In general, it will be convenient to associate more than one semantic function with each syntactic rule. An example will illustrate this.

A rational number may appear as a sequence of digits optionally followed by a decimal point and then another sequence of digits. A grammar defining a syntax for rational numbers is:

Grammar C

$D \rightarrow 0$

$D \rightarrow 1$

.
.
.

$D \rightarrow 9$

$I \rightarrow I\ D$

$N \rightarrow I$

$N \rightarrow I\ .\ I$

$V_N = \{D, I, N\}$

$V_T = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, .\}$

Starting symbol = N

Here the symbol "D" stands for digit, "I" stands for a sequence of digits, and "N" stands for number. To give each number, N, a meaning we introduce functions L and V as follows:

Each digit, D, has a value, V(D), which is an integer.

Each sequence of digits, I, has a length, L(I), which is an integer.

Each sequence of digits, I, has a value, V(I), which is an integer.

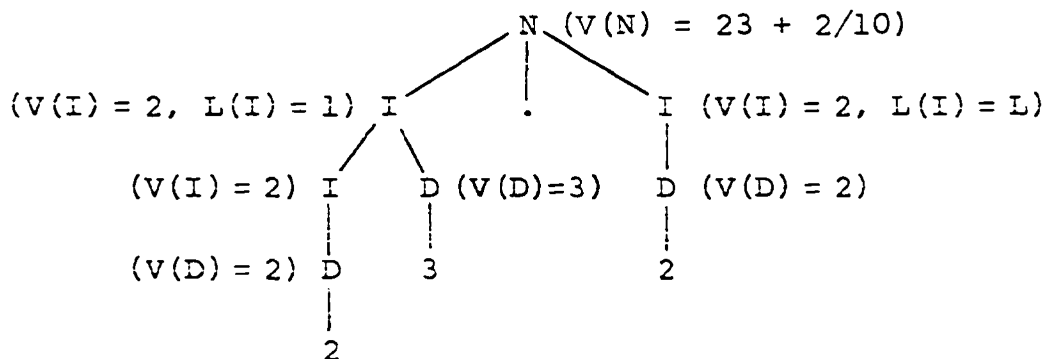
Each number, N, has a value, V(N), which is a rational number.

We now augment Grammar C with the use of the functions L and V as follows:

Syntactic Rules	Semantic Rules
D → 0	V(D) = 0
D → 1	V(1) = 1
. . .	. . .
D → 9	V(9) = 9
I → D	V(I) = V(D) L(I) = 1
I → I D	V(I) = 10 * V(I) + V(D) L(I) = L(I) + 1
N → I	V(N) = V(I)
N → $I_1 . I_2$	V(N) = $V(I_1) + V(I_2)/10L(I_2)$

Note that the subscripts in the last rule are used strictly to differentiate the two I's.

The application of this augmented grammar to obtain the meaning of the string 23.2 is illustrated by that string's augmented syntax tree appearing below:



This particular augmented grammar is adequate to define each rational number. However, it does not entirely agree with our intuitive notion of a rational number because it makes no use of the position of each digit in determining that digit's value. For example, the leftmost "2" in the number 23.2 is given the same value as the rightmost "2" whereas our intuition tells us that the leftmost "2" has value twenty while the rightmost "2" has value two-tenths.

Knuth (5) has outlined an extended approach which more closely follows our intuition. The same grammar can be used but now the semantic functions are somewhat different. Assume the following:

Each digit, D, has a value, V(D), which is a rational number.

Each digit, D, has a scale, S(D), which is an integer.

Each sequence of digits, I, has a value, V(I), which is a rational number.

Each sequence of digits, I, has a length, L(I), which is an integer.

Each sequence of digits, I, has a scale, S(I), which is an integer.

Each number, N, has a value, V(N), which is a rational number.

The augmented grammar for C is now:

Syntactic Rules	Semantic Rules
$D \rightarrow 0$	$V(D) = 0$
$D \rightarrow 1$	$V(D) = 10^{S(D)}$
$D \rightarrow 2$	$V(D) = 2 * 10^{S(D)}$
$D \rightarrow 3$	$V(D) = 3 * 10^{S(D)}$
. . .	. . .
$D \rightarrow 9$	$V(D) = 9 * 10^{S(D)}$
$I \rightarrow D$	$V(I) = V(D)$ $S(D) = S(I)$ $L(I) = 1$
$I_1 \rightarrow I_2D$	$V(I_1) = V(I_2) + V(B)$ $S(D) = S(I_1)$ $S(I_2) = S(I_1) + 1$
$N \rightarrow I_1 . I_2$	$V(N) = V(I_1) + (V(I_2)$ $S(I_2) = - L(I_2)$ $S(I_1) = 0$

The augmented syntax tree for the number "23.2" is shown in Figure 1.

Note that in this augmented syntax tree the value associated with the leftmost "2" is twenty while the value associated with the rightmost "2" is two-tenths.

The above augmented syntax tree has an important feature not present in the augmented syntax tree obtained for the arithmetic expression "a + b * c". In the above tree it was necessary to work the tree in both directions to

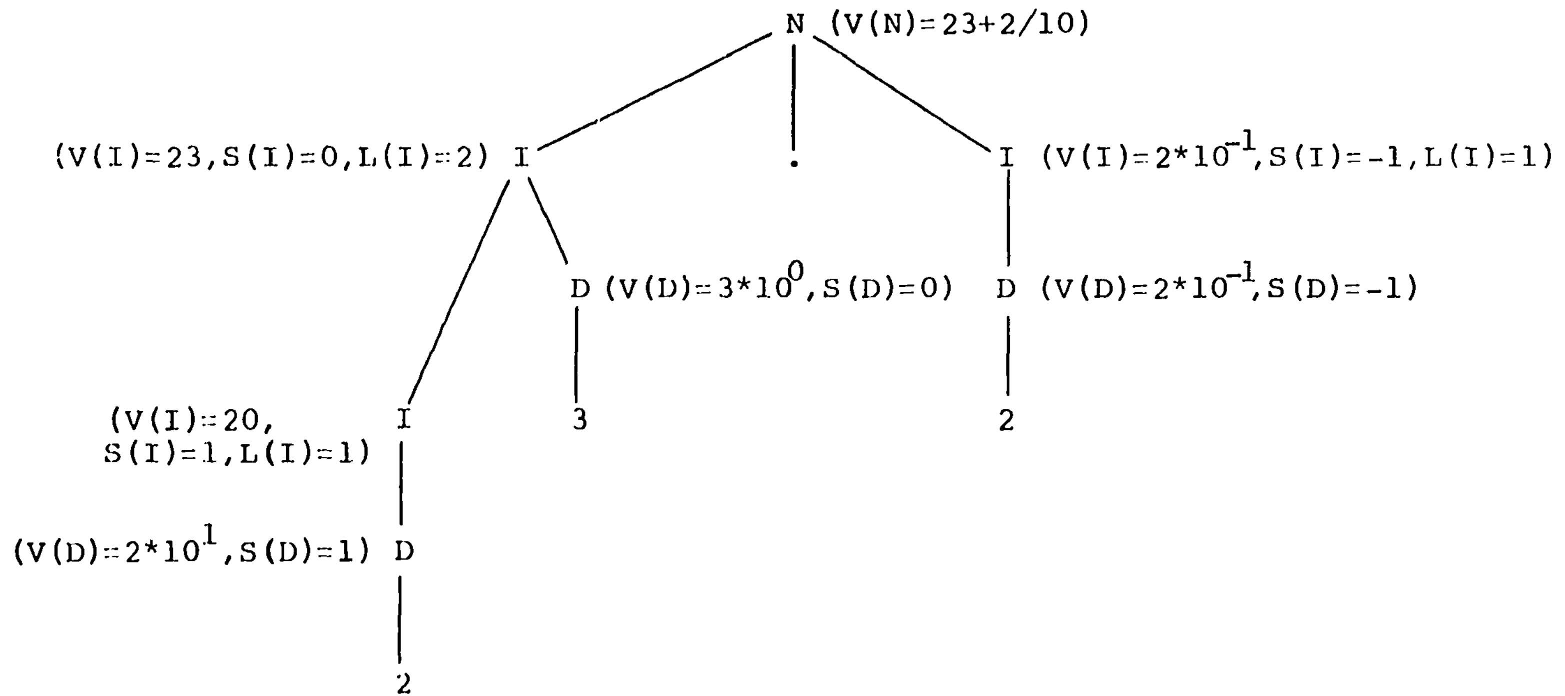


Figure 1. Augmented syntax tree for the number "23.2"

evaluate the semantic functions. The semantic function L was evaluated from the bottom up for the nodes to the right of the decimal point. Then the function S was evaluated from the top down. Finally, V was evaluated from the bottom up.

It was necessary to evaluate the semantic functions in both directions in the tree because of the way the semantic functions were defined in the augmented grammar. Some of the functions were defined for non-terminal symbols appearing on the right side of the corresponding production while other of the functions were defined for non-terminal symbols appearing on the left side of the corresponding production. The augmented grammar for Grammar A, on the other hand, had semantic functions defined for non-terminal symbols appearing always on the left sides of the corresponding productions. It will be shown in the next chapter that this more generalized method of defining semantic functions lends itself well to the definition of programming languages.

III. COMPLETE DEFINITION OF A SIMPLE PROGRAMMING LANGUAGE

It remains to be shown that the method discussed in the previous chapter lends itself well to the definition of an actual programming language. This chapter does that by proceeding as follows:

1) First, a simple programming language, called Progol, is introduced and discussed. The syntax for Progol is given using a context-free grammar.
2) A simple computer, called Mickey, is introduced and discussed as a vehicle for expressing the meaning of Progol programs.
3) Some conventions used in the definition of the semantic functions are introduced and motivated.
4) Semantic functions are defined and discussed for each syntactic rule in the grammar.
5) Finally, examples are given demonstrating the actual application of the augmented grammar defining Progol.

The actual structure of Progol was chosen to demonstrate that the method of Chapter II works for an actual programming language. Progol is not a particularly practical language but it does contain many features that practical languages must contain. It is felt that the extension of Progol into

a practical language (such as Algol) would be, with only a few exceptions, completely straightforward.

The grammar for Progol is as follows:

Prog	→	begin Body end
Body	→	Decl ; Body
Body	→	List
List	→	Stat
List	→	List ; Stat
Stat	→	Id ← Exp
Stat	→	goto Id
Stat	→	if Exp ≠ 0 then Stat
Stat	→	read(Id)
Stat	→	print(Id)
Stat	→	begin List end
Stat	→	Id : Stat
Exp	→	Term
Exp	→	Exp + Term
Term	→	Pr
Term	→	Term * Pr
Pr	→	(Exp)
Pr	→	Id
Id	→	A

Id → B

.
.
.

Id → Z

Decl → <u>integer</u> Id

Decl → Decl $ Id

V_N = {Prog, Body, List, Stat, Exp, Term, Pr, Id}

V_T = {<u>begin</u>, <u>end</u>, ;, ←, <u>goto</u>, <u>if</u>, ≠, <u>then</u>, <u>read</u>, (,), <u>print</u>, :, +, *, A, B, ..., Y, Z, integer, $}

Starting Symbol = Prog

Note that various elements of V_N and V_T contain more than one character. This has been done for clarity and it should be understood that such multiple character elements are to be regarded as single symbols in the grammar.

The elements of V_N stand for various components in Progol. This correspondence is:

Symbol	Language Component
Prog	Program
Body	Body of a program
List	List of statements
Stat	Statement
Exp	Arithmetic expression
Term	Term
Pr	Primary
Id	Identifier
Decl	Declaration

The semantic functions defining the meaning of each Progol program must ultimately define each Progol program in terms of some representation that is well-understood. For our purposes the semantic functions will define Progol programs in terms of machine language programs for the Mickey computer. Mickey is a fixed word length, single address computer with one accumulator. Its instructions are executed in sequential order unless interrupted by a transfer instruction. It is assumed that the instructions and operations of Mickey are simple enough so as to be well-understood. A description of the Mickey instruction set is:

Instruction	Description
LDA A	Load accumulator with the contents of a cell A
STA A	Store the contents of the accumulator into cell A
ADD A	Add the contents of cell A to the contents of the accumulator
MPY A	Multiply the contents of the accumulator by the contents of cell A
BZA A	If the accumulator is zero then transfer control to the instruction located in cell A
BRU A	Transfer control to the instruction located in cell A
IN A	Read an integral number into cell A
OUT A	Print the contents of cell A
HLT	Halt

Certain "built-in" semantic functions and conventions will be used in the definition of Progol. These are:

1) The semantic function <u>tempcell</u> will have as its value an address in a "variable storage area" in the Mickey computer. Each evaluation of <u>tempcell</u> produces a different address. The main use of <u>tempcell</u> is to select addresses for variable storage.

2) The semantic function <u>insert</u>(X, Y, Tab) will mean insert the ordered pair (X, Y) into a table named Tab. The element Y can then be accessed using the

form Tab(X). The main use of <u>insert</u> is to build tables.

3) The semantic functions for a particular syntactic rule are evaluated in the order they appear. Arguments for a particular semantic function are evaluated from left to right.

4) In general, the processing of the nodes in a syntax tree begins with the top node and proceeds in the order the nodes are referenced in the semantic functions. The processing of the nodes for declarations is an exception to this rule.

5) A special concatenation operation, "||", is used in forming the Mickey instructions.

The semantic functions for the programming language, Progol, will now be given. It will also be shown how these functions are used to generate Mickey programs from Progol programs.

The production

Prog → <u>begin</u> Body <u>end</u>

has the associated semantic functions

start(Body) = a_0

insert(follow(Body),'HLT',M) .

The first of these says that the first instruction of a program is to be assigned some initial address, a_0. The second of these says to insert a halt instruction (into a table

representing the memory of the computer) as the last instruction in the generated Mickey program.

The production

$$\text{Body} \rightarrow \text{List}$$

is defined by the functions

$$\text{start(List)} = \text{start(Body)}$$

$$\text{follow(Body)} = \text{follow(List)} .$$

These say that the first instruction of List is also the first instruction of Body and that the instruction following Body is the instruction following List.

For the production

$$\text{Body}_1 \rightarrow \text{Decl} \ ; \ \text{Body}_2$$

we have the functions

$$\text{start}(\text{Body}_2) = \text{start}(\text{Body}_1)$$

$$\text{follow}(\text{Body}_1) = \text{follow}(\text{Body}_2) .$$

Note here that these semantic functions are independent of Decl.

For the production

$$\text{List} \rightarrow \text{Stat}$$

we have the associated semantic functions

$$\text{start(Stat)} = \text{start(List)}$$

$$\text{follow(List)} = \text{follow(Stat)} .$$

For the production

$$\text{List}_1 \rightarrow \text{List}_2 \ ; \ \text{Stat}$$

we have

$$start(List_2) = start(List_1)$$

$$start(Stat) = follow(List_2)$$

$$follow(List_1) = follow(Stat) .$$

These say that the first instruction of $List_2$ is also the first instruction of $List_1$, that the first instruction of Stat is to immediately follow the last instruction of $List_2$, and that the instruction following Stat is also the instruction following $List_1$.

The production

$$Stat_1 \rightarrow Id : Stat_2$$

has as its functions

<u>insert</u>(text(Id),start($Stat_1$),Lab)

$$start(Stat_2) = start(Stat_1)$$

$$follow(Stat_1) = follow(Stat_2) .$$

Note that a label corresponding to Id is inserted into the label table. The address associated with that label is the address of the first instruction in $Stat_1$. It is an error if an attempt is made to insert a label into the label table when it is already there. This prohibits multiply-defined labels.

For the production

$$Stat \rightarrow Id \leftarrow Exp$$

we have

start(Exp) = start(Stat)

<u>insert</u>(follow(Exp),'LDA'||result(Exp),M)

<u>insert</u>(follow(Exp)+1,'STA'||Symbol(text(Id)),M)

follow(Stat) = follow(Exp)+2 .

These functions generate and place in M the instructions to store the result of Exp into the address associated with Id.

The production

Stat → <u>goto</u> Id

is defined by the semantic functions

follow(Stat) = start(Stat)+1

<u>insert</u>(start(Stat),'BRU'||Lab(text(Id)),M) .

The second of these functions causes the label table to be searched for a label corresponding to Id. If that label is found then the transfer instruction is generated. If, however, the label is not found then the generation of the transfer is deferred until the label is inserted into the label table (by some other semantic function). If the label is never inserted into the label table then an error results.

The production

$Stat_1$ → <u>if</u> Exp ≠ 0 <u>then</u> $Stat_2$

is defined by the functions

start(Exp) = start($Stat_1$)

insert(follow(Exp),'LDA'||result(exp),M)

start($Stat_2$) = follow(Exp)+2

insert(follow(Exp)+1,'BZA'||follow($Stat_2$),M)

follow($Stat_1$) = follow($Stat_2$) .

These functions generate two instructions. The first loads the result of Exp into the accumulator. The second transfers control to the instruction following $Stat_1$ if the accumulator contains a zero. Note that the second instruction is not actually generated until after the instructions for $Stat_2$ are generated and the address of the instruction following $Stat_2$ is known.

The production

Stat → read(Id)

is defined by the functions

insert(start(Stat),'IN'||Symbol(text(Id)),M)

follow(Stat) = start(Stat)+1 .

The production

Stat → print(Id)

is defined by the functions

insert(start(Stat),'OUT'||Symbol(text(Id)),M)

follow(Stat) = start(Stat)+1 .

The production

Stat → begin List end

is defined by the functions

start(List) = start(Stat)

follow(Stat) = follow(List) .

Note that the instruction following Stat is the instruction following List.

The generation of code to evaluate expressions involves not only the creation of the code but also the selection of storage cells to hold the final result and all temporary results. These storage cells will be selected from the same area of the computer's memory by the function tempcell with no attempt being made to use cells in an optimal manner. Note that result(Exp) is defined to be the address of the value of Exp.

For the production

Exp → Term

we have the semantic functions

start(Term) = start(Exp)

follow(Exp) = follow(Term)

result(Exp) = result(Term) .

For the production

$Exp_1 \rightarrow Exp_2 + Term$

we have the functions

$start(Exp_2) = start(Exp_1)$

$start(Term) = follow(Exp_2)$

<u>insert</u>(follow(Term),'LDA'||result(Exp_2),M)

<u>insert</u>(follow(Term)+1,'ADD'||result(Term),M)

result(Exp_1) = <u>tempcell</u>

<u>insert</u>(follow(Term)+2,'STA'||result(Exp_1),M)

$follow(Exp_1) = follow(Term)+3$.

The Mickey instructions that are generated by these functions load the accumulator with the value of Exp then add to that the value of Temp. Then a storage cell is selected and an instruction is generated to store the contents of the accumulator (and hence the value of Exp) into that selected cell.

For the production

$Term \rightarrow Pr$

we have the functions

$start(Pr) = start(Term)$

$follow(Term) = follow(Pr)$

$result(Term) = result(Pr)$.

For the production

$Term_1 \rightarrow Term_2 * Pr$

we have the functions

start($Term_2$) = start($Term_1$)

start(Pr) = follow($Term_2$)

<u>insert</u>(follow(Pr),'LDA'||result($Term_2$),M)

<u>insert</u>(follow(Pr)+1,'MPY'||result(Pr),M)

result($Term_1$) = <u>tempcell</u>

<u>insert</u>(follow(Pr)+2,'STA'||result($Term_1$),M)

follow($Term_1$) = follow(Pr)+3 .

For the production

Pr → (Exp)

we have the functions

start(Exp) = start(Pr)

follow(Pr) = follow(Exp)

result(Pr) = result(Exp) .

For the production

Pr → Id

we have the functions

follow(Pr) = start(Pr)

result(Pr) = Symbol(text(Id)) .

Note that no Mickey instructions are required to calculate the value of Id and, thus, the value of Pr. Therefore, the instruction following Pr is the instruction starting Pr.

For the production

Id → A

and those like it we have functions like

text(Id) = 'a' .

The function, text(Id), is just a device mapping a letter from its external representation into its representation within the Mickey computer.

For the production

Decl → <u>integer</u> Id

we have the function

<u>insert</u>(text(Id),<u>tempcell</u>,Symbol) .

For the production

$Decl_1$ → $Decl_2$ $ Id

we have the function

<u>insert</u>(text(Id),<u>tempcell</u>,Symbol) .

The functions associated with declarations essentially insert the declared identifier (and its associated address) into the symbol table.

The final augmented grammar for Progol appears in Table 1.

Let us now examine the application of this augmented grammar to two simple examples. The two examples are:

Table 1. Augmented grammar for Progol

Syntactic Rules	Semantic Rules
Prog → begin Body end	start(Body) = a_o insert(follow(Body),'HLT',M)
Body → List	start(List) = start(Body) follow(Body) = follow(List)
$Body_1$ → Decl : $Body_2$	start($Body_2$) = start($Body_1$) follow($Body_1$) = follow($Body_2$)
List → Stat	start(Stat) = start(List) follow(List) = follow(Stat)
$List_1$ → $List_2$; Stat	start($List_2$) = start($List_1$) start(Stat) = follow($List_2$) follow($List_1$) = follow(Stat)
$Stat_1$ → Id : $Stat_2$	insert(text(Id),start($Stat_1$),Lab) start($Stat_2$) = start($Stat_1$) follow($Stat_1$) = follow($Stat_2$)
Stat → Id ← Exp	start(Exp) = start(Stat) insert(follow(Exp),'LDA'||result(Exp),M) insert(follow(Exp)+1,'STA'||Symbol(text(Id)),M) follow(Stat) = follow(Exp)+2

Table 1 (Continued)

Syntactic Rules	Semantic Rules
Stat → goto Id	follow(Stat) = start(Stat)+1
	insert(start(Stat),'BRU'||Lab(text(Id)),M)
Stat$_1$ → if Exp≠0 then Stat$_2$	start(Exp) = start(Stat$_1$)
	insert(follow(Exp),'LDA'||result(Exp),M)
	start(Stat$_2$) = follow(Exp)+2
	insert(follow(Exp)+1,'BZA'||follow(Stat$_2$),M)
	follow(Stat$_1$) = follow(Stat$_2$)
Stat → read(Id)	insert(start(Stat),'IN'||Symbol(text(Id)),M)
	follow(Stat) = start(Stat)+1
Stat → print(Id)	insert(start(Stat),'OUT'||Symbol(text(Id)),M)
	follow(Stat) = start(Stat)+1
Stat → begin List end	start(List) = start(Stat)
	follow(Stat) = follow(List)
Exp → term	start(term) = start(Exp)
	follow(Exp) = follow(term)
	result(Exp) = result(term)

Table 1 (Continued)

<table>
<tr><th>Syntactic Rules</th><th>Semantic Rules</th></tr>
<tr><td>$Exp_1 \rightarrow Exp_2 + term$</td><td>start($Exp_2$) = start($Exp_1$)
start(term) = follow(Exp_2)
insert(follow(term),'LDA'||result(Exp_2),M)
insert(follow(term)+1,'ADD'||result(term),M)
result(Exp_1) = tempcell
insert(follow(term)+2,'STA'||result(Exp_1),M)
follow(Exp_1) = follow(term)+3</td></tr>
<tr><td>$term \rightarrow Pr$</td><td>start(Pr) = start(term)
follow(term) = follow(Pr)
result(term) = result(Pr)</td></tr>
<tr><td>$term_1 \rightarrow term_2 * Pr$</td><td>start($term_2$) = start($term_1$)
start(Pr) = follow($term_2$)
insert(follow(Pr),'LDA'||result($term_2$),M)
insert(follow(Pr)+1,'MPY'||result(Pr),M)
result($term_1$) = tempcell
insert(follow(Pr)+2,'STA'||result($term_1$),M)
follow($term_1$) = follow(Pr)+3</td></tr>
<tr><td>$Pr \rightarrow (Exp)$</td><td>start(Exp) = start(Pr)
follow(Pr) = follow(Exp)
result(Pr) = result(Exp)</td></tr>
</table>

Table 1 (Continued)

Syntactic Rules	Semantic Rules
Pr → Id	follow(Id) = start(Pr) result(Pr) = Symbol(text(Id))
Id → A	text(Id) = 'a'
.	.
.	.
.	.
Id → Z	text(Id) = 'z'
Decl → integer Id	insert(text(Id),tempcell,Symbol)
$\mathrm{Decl}_1 \rightarrow \mathrm{Decl}_2 \ \$ \ \mathrm{Id}$	insert(text(Id),tempcell,Symbol)

Program A

```
begin
  integer A $ B $ C ;
    read(A);
    read(B);
    C ← A + B * C
end
```

Program B

```
begin
  integer A ;
    L : read(A);
    if A ≠ 0 then begin
                      print(A);
                      goto L
                  end;
    print(A)
end
```

The syntax tree (with some of the nodes numbered for later reference) for Program A is found in Figure 2.

The application of the augmented grammar to Program A yields the traces as shown in Table 2. Assume a_o is 1 initially.

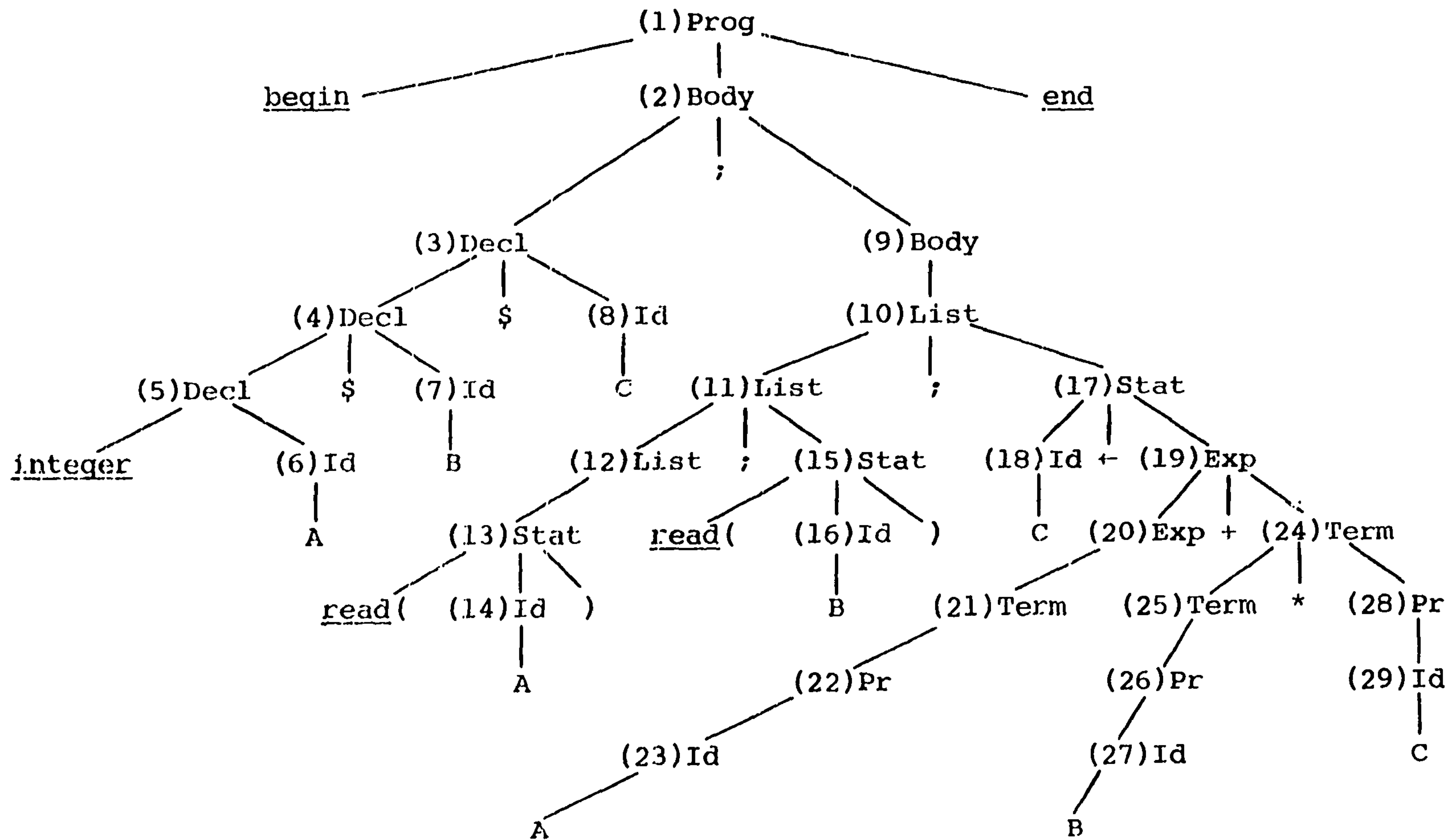


Figure 2. Syntax tree for Program A

Table 2. Application of the augmented grammar to Program A trace yields

Line	Node	Production	Semantic Results
1	1 (Prog)	Prog → <u>begin</u> Body <u>end</u>	start(Body) = 1
2	2 (Body)	$Body_1$ → Decl ; $Body_2$	start(Body) = 1
3	3 (Decl)	$Decl_1$ → $Decl_2$ $ Id	
4	4 (Decl)	$Decl_1$ → $Decl_2$ $ Id	
5	5 (Decl)	Decl → <u>integer</u> Id	
6	6 (Id)	Id → A	text(Id) = 'a'
7	5 (Decl)	(line 5)	<u>tempcell</u> = T1
			('a', T1) in Symbol
8	7 (Id)	Id → B	text(Id) = 'b'
9	4 (Decl)	(line 4)	<u>tempcell</u> = T2
			('b', T2) in Symbol
10	8 (Id)	Id → C	text(Id) = 'c'
11	3 (Decl)	(line 3)	<u>tempcell</u> = T3
			('c', T3) in Symbol
12	9 (Body)	Body → List	start(List) = 1
13	10 (List)	$List_1$ → $List_2$; Stat	start($List_2$) = 1

Table 2 (Continued)

Line	Node	Production	Semantic Results
14	11 (List)	$List_1 \rightarrow List_2$; Stat	start($List_2$) = 1
15	12 (List)	List → Stat	start(Stat) = 1
16	13 (Stat)	Stat → read(Id)	
17	14 (Id)	Id → A	text(Id) = 'a'
18	13 (Stat)	(line 16)	(1, IN T1) in M
			follow(Stat) = 2
19	12 (List)	(line 15)	follow(List) = 2
20	11 (List)	(line 14)	start(Stat) = 2
21	15 (Stat)	Stat → read(Id)	
22	16 (Id)	Id → B	text(Id) = 'b'
23	15 (Stat)	(line 21)	(2, IN T2) in M
			follow(Stat) = 3
24	11 (List)	(line 14)	start(Stat) = 3
25	10 (List)	(line 13)	start(Stat) = 3
26	17 (Stat)	Stat → Id ← Exp	start(Exp) = 3
27	19 (Exp)	$Exp_1 \rightarrow Exp_2$ + term	start(Exp_2) = 3

Table 2 (Continued)

Line	Node	Production	Semantic Results
28	20 (Exp)	Exp → Term	start(Term) = 3
29	21 (Term)	Term → Pr	start(Pr) = 3
30	22 (Pr)	Pr → Id	
31	23 (Id)	Id → A	text(Id) = 'a'
32	22 (Pr)	(line 30)	follow(Pr) = 3
			result(Pr) = Tl
33	21 (Term)	(line 29)	follow(Term) = 3
			result(Term) = Tl
34	20 (Exp)	(line 28)	follow(Exp) = 3
			result(Exp) = Tl
35	19 (Exp)	(line 27)	start(Term) = 3
36	24 (Term)	$Term_1 \rightarrow Term_2 * Pr$	$start(Term_2) = 3$
37	25 (Term)	Term → Pr	start(Pr) = 3
38	26 (Pr)	Pr → Id	
39	27 (Id)	Id → B	text(Id) = 'b'

Table 2 (Continued)

Line	Node	Production	Semantic Results
38	26 (Pr)	(line 38)	follow(Pr) = 3
			result(Pr) = T2
39	25 (Term)	(line 37)	follow(Term) = 3
			result(Term) = T2
40	24 (Term)	(line 36)	start(Pr) = 3
41	28 (Pr)	Pr → Id	
42	29 (Id)	Id → c	text(Id) = 'c'
43	28 (Pr)	(line 41)	follow(Pr) = 3
			result(Pr) = T3
44	24 (Term)	(line 36)	(3, 'LDA' T2) in M
			(4, 'MPY' T3) in M
			result($Term_1$) = T4
			(5, 'STA' T4) in M
			follow($Term_1$) = 6

Table 2 (Continued)

Line	Node	Production	Semantic Results
45	19 (Exp)	(line 27)	(6, LDA T1) in M
			(7, ADD T4) in M
			result(Exp_1) = T5
			(8, STA T5) in M
			follow(Exp_1) = 9
46	17 (Stat)	(line 26)	(9, LDA T5) in M
47	18 (Id)	Id → c	text(Id) = 'c'
48	17 (Stat)	(line 46)	(10, STA T3) in M
			follow(Stat) = 11
49	10 (List)	(line 13)	follow($List_1$) = 11
50	9 (Body)	(line 12)	follow(Body) = 11
51	2 (Body)	(line 2)	follow($Body_1$) = 11
52	1 (Prog)	(line 1)	(11, HLT) in M

The final contents of the tables are:

Symbol

'a'	T1
'b'	T2
'c'	T3

M

1	IN	T1
2	IN	T2
3	LDA	T2
4	MPY	T3
5	STA	T4
6	LDA	T1
7	ADD	T4
8	STA	T5
9	LDA	T5
10	STA	T3
11	HLT	

The syntax tree for Program B is shown in Figure 3.

The complete trace for Program B is shown in Table 3.

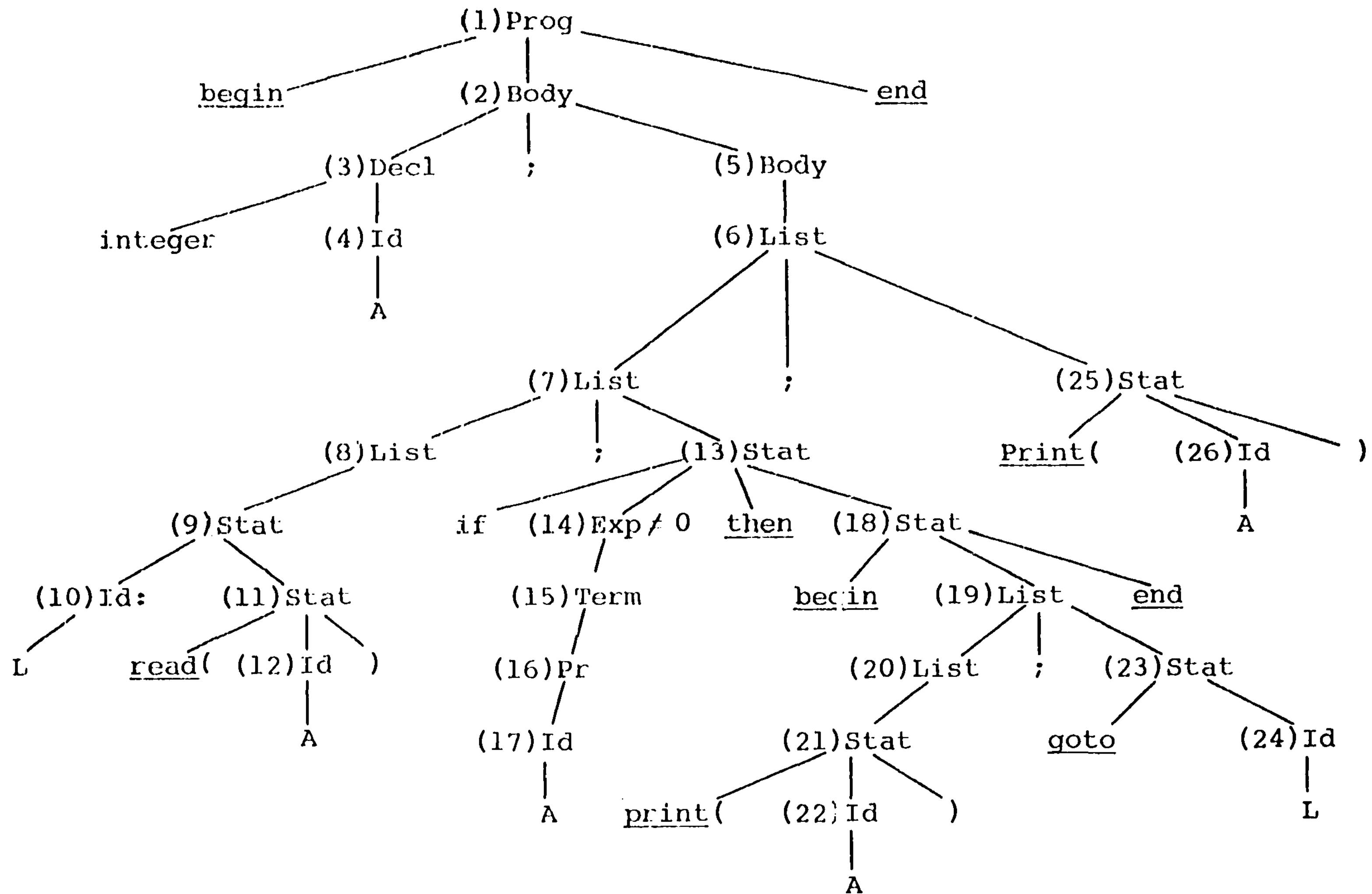


Figure 3. Syntax tree for Program B

Table 3. Complete trace for Program B

Line	Node	Production	Results
1	1 (Prog)	Prog → begin Body end	start(Body) = 1
2	2 (Body)	$Body_1$ → Decl ; $Body_2$	start($Body_2$) = 1
3	3 (Decl)	Decl → integer Id	
4	4 (Id)	Id → A	text(Id) = 'a'
5	3 (Decl)	(line 3)	('a', T1) in Symbol
6	5 (Body)	Body → List	start(List) = 1
7	6 (List)	$List_1$ → $List_2$; Stat	start($List_2$) = 1
8	7 (List)	$List_1$ → $List_2$; Stat	start($List_2$) = 1
9	8 (List)	List → Stat	start(Stat) = 1
10	9 (Stat)	$Stat_1$ → Id : $Stat_2$	
11	10 (Id)	Id → L	text(L) = 'L'
12	9 (Stat)	(line 10)	('L', 1) in lab
			start($Stat_2$) = 1
13	11 (Stat)	Stat → read(Id)	
14	12 (Id)	Id → A	text(Id) = 'a'

Table 3 (Continued)

Line	Node	Production	Results
15	11 (Stat)	(line 13)	(1, IN T1) in M
			follow(Stat) = 2
16	9 (Stat)	(line 10)	follow(Stat) = 2
17	8 (List)	(line 9)	follow(List) = 2
18	7 (List)	(line 8)	start(Stat) = 2
19	13 (Stat)	Stat → <u>if</u> Exp ≠ 0	start(Exp) = 2
		<u>then</u> Stat	
20	14 (Exp)	Exp → Term	start(Term) = 2
21	15 (Term)	Term → Pr	start(Pr) = 2
22	16 (Pr)	Pr → Id	
23	17 (Id)	Id → A	text(Id) = 'a'
24	16 (Pr)	(line 22)	follow(Pr) = 2
			result(Pr) = T1
25	15 (Term)	(line 21)	follow(Term) = 2
			result(Term) = T1
26	14 (Exp)	(line 20)	follow(Exp) = 2
			result(Exp) = T1

Table 3 (Continued)

Line	Node	Production	Results
27	13 (Stat)	(line 19)	(2, LDA T1) in M
			start(Stat) = 4
28	18 (Stat)	Stat → begin List end	start(List) = 4
29	19 (List)	List$_1$ → List$_2$; Stat	start(List$_2$) = 4
30	20 (List)	List → Stat	start(Stat) = 4
31	21 (Stat)	Stat → print(Id)	
32	22 (Id)	Id → A	text(Id) = 'a'
33	21 (Stat)	(line 31)	(4, OUT T1) in M
			follow(Stat) = 5
34	20 (List)	(line 30)	follow(List) = 5
35	19 (List)	(line 29)	start(Stat) = 5
36	23 (Stat)	Stat → goto Id	follow(Stat) = 6
37	24 (Id)	Id → L	text(Id) = 'L'
38	23 (Stat)	(line 36)	(5, BRU 1) in M
39	19 (List)	(line 35)	follow(List$_1$) = 6
40	18 (Stat)	(line 28)	follow(Stat) = 6

Table 3 (Continued)

Line	Node	Production	Results
41	13 (Stat)	(line 19)	(3, BZA 6) in M
			follow($Stat_1$) = 6
42	7 (List)	(line 8)	follow(List) = 6
43	6 (List)	(line 7)	start(Stat) = 6
44	25 (Stat)	Stat → print(Id)	
45	26 (Id)	Id → A	text(Id) = 'a'
46	25 (Stat)	(line 44)	(6, OUT T1) in M
			follow(Stat) = 7
47	6 (List)	(line 7)	follow(List) = 7
48	5 (Body)	(line 6)	follow(Body) = 7
49	2 (Body)	(line 2)	follow(Body) = 7
50	1 (Prog)	(line 1)	(7, HLT) in M

The final contents of the tables are:

Symbol

'a' T1

Lab

'L' 1

M

1	IN	T1
2	LDA	T1
3	BZA	6
4	OUT	T1
5	BRU	1
6	OUT	T1
7	HLT	

Note that in the generation of code for the conditional statement in Program B it was necessary to reserve a cell for the conditional instruction and then insert the instruction later. The semantic function, follow, was particularly useful in this process.

IV. DISCUSSION

This dissertation investigated a method for the complete definition of programming languages. It was first shown how syntax and semantics can be related by the insertion of semantic functions into a context-free grammar defining a language. A complete definition of a simple programming language was then given to show the workability of this approach.

Unfortunately, there are no universally accepted guidelines for the evaluation of methods for programming language definition. A set of criteria taken from de Bakker (1) will be used to evaluate the definition of Progol given in Chapter III. This evaluation is as follows:

1. A first criterion is the scope of the method. Is it applicable to all programming languages or only to a certain subclass of them?

In the sense that the method discussed in this dissertation depends upon a language being defined syntactically by a context-free grammar, then this method falls short of satisfying this criterion. However, the properties of programming languages that keep them from being context-free can quite often be handled as semantic problems. For example, the requirement that a variable be declared can be enforced by a semantic function. In such cases, this method works quite well. A more difficult problem is the

problem of defining a programming language whose syntax can change while a program is being run. This method cannot handle such languages.

2. A general criterion of great importance, particularly from the standpoint of teaching, concerns the notions of readability, transparency, conciseness, and elegance of the description. Is the description readable, transparent, concise, and elegant?

These notions are vague and hence subjective. It is this writer's opinion that the description of Progol that was given in Chapter III does satisfy the above criteria. Others might not agree. Lepgard (6), McCarthy (8), and Wirth and Weber (12) are examples of descriptions of programming languages with which this description can be compared.

3. Is it possible to leave the definition of certain parts of the language either completely open, or to give only a partial definition of them? Does the definition provide information on the division of the actions to be performed at compile time or at run time?

A judicious choice of built-in semantic functions presumably could allow for certain parts of a language to be undefined. It is debatable whether this is desirable or

not.

4. How much insight is gained into the properties of the language concepts by formally describing them? Is it possible to reflect independent concepts by more or less independent parts of the description?

Knuth (5) shows that the theory of graphs (see Knuth, 4) can be used to detect dependent language components in languages defined by the approach used in this dissertation. The use of graph theory for this purpose is quite appealing to this writer and deserves to be explored further.

5. Does a descriptive method lend itself well to rigorously proving things about the languages it is used to define?

One of the reasons for formally defining programming languages is to be able to prove things about them and their translators. Knuth (5) shows that a given definition can be mechanically tested for completeness and circularities. It remains to be shown just how well this approach lends itself to other types of proofs. Painter (9) gives a proof of the correctness of a compiler for a simple programming language and his techniques appear to be adaptable to proving the same thing about the compiler presented in this work. London (7) is an excellent bibliography on work done in this area.

V. BIBLIOGRAPHY

1. de Bakker, J. W. Semantics of programming languages. In Tou, J. T., ed. Advances in information systems sciences. Volume 2. Pp. 173-227. New York, New York, Plenum Press. 1969.

2. Feldman, J. and Gries, D. Translator writing systems. Communications of the Association for Computing Machinery 11, No. 2: 77-113. 1968.

3. Irons, E. T. A syntax directed compiler for Algol 60. Communications of the Association for Computing Machinery 4, No. 1: 51-55. 1961.

4. Knuth, D. E. The art of computer programming. Volume 1. Fundamental algorithms. Reading, Massachusetts, Addison-Wesley Publishing Company. 1968.

5. Knuth, D. E. Semantics of context-free languages. Mathematical Systems Theory 2, No. 2: 127-145. 1968.

6. Lepgard, H. F. A formal system for defining the syntax and semantics of computer languages. Unpublished Ph.D. thesis. Cambridge, Massachusetts, Library, Massachusetts Institute of Technology. 1969.

7. London, R. L. Bibliography on proving the correctness of computer programs. Technical Report No. 64. Madison, Wisconsin, Computer Science Department, University of Wisconsin. 1969.

8. McCarthy, J. A formal definition of a subset of Algol. In Steel, T. B., ed. Formal language description languages for computer programming. Pp. 1-12. Amsterdam, Netherlands, North-Holland Publishing Company. 1966.

9. Painter, J. A. Semantic correctness of a compiler for an Algol-like language. Unpublished Ph.D. thesis. Stanford, California, Library, Stanford University. 1967.

10. Wegner, P. Notes for a lecture on theories of semantics. An unpublished paper presented at a symposium on programming languages definition, San Francisco, California, August, 1969. New York, New York, Special Interest Group on Programming Languages, Association for Computing Machinery. 1969.

11. Wirth, N. and Weber, H. Euler: a generalization of Algol, and its formal definition: Part I. Communications of the Association for Computing Machinery 9, No. 1: 11-23. 1966.

12. Wirth, N. and Weber, H. Euler: a generalization of Algol, and its formal definition: Part II. Communications of the Association for Computing Machinery 9, No. 2: 89-99. 1966.

VI. ACKNOWLEDGMENTS

I express my sincere appreciation to Professor Roy Keller for his many helpful suggestions and detailed guidance of this work. I also would like to express my appreciation to Professor Robert Stewart for his stabilizing presence and encouragement during my career as a graduate student. A special acknowledgment is due Professor Donald Fitzwater (now at the University of Wisconsin) for the significant contribution he has made to my growth as a student.

I would like to thank Mrs. Donald Phipps for an excellent job of typing, Mr. William Thomas for the fine work he did on the drawings, and the Iowa State University Thesis staff for their able assistance in the final editing of the manuscript.