STANFORD ARTIFICIAL INTELLIGENCE PROJECT
MEMO AIM-126
COMPUTER SCIENCE DEPARTMENT
REPORT NO. CS 169

EXAMPLES OF FORMAL SEMANTICS

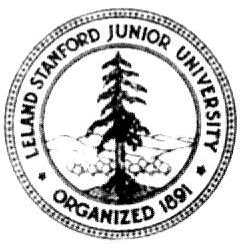
BY

DONALD E. KNUTH

JULY 1970

COMPUTER SCIENCE DEPARTMENT

STANFORD UNIVERSITY



STANFORD ARTIFICIAL INTELLIGENCE PROJECT
MEMO AIM-126
COMPUTER SCIENCE DEPARTMENT REPORT
CS Report 169

EXAMPLES OF FORMAL SEMANTICS

by

Donald E. Knuth

ABSTRACT: A technique of formal definition, based on relations between "attributes" associated with nonterminal symbols in a context-free grammar, is illustrated by several applications to simple yet typical problems. First we define the basic properties of lambda expressions, involving substitution and renaming of bound variables. Then a simple programming language is defined using several different points of view. The emphasis is on "declarative" rather than "imperative" or "algorithmic" forms of definition.

KEYWORDS: Lambda expressions, synthesized attributes, inherited attributes, Turingol, TL/I, information structures.

The research reported here was supported in part by the Advanced Research Projects Agency of the Office of the Secretary of Defense (SD 183), and in part by IBM Corporation.

Reproduced in the USA. Available from the Clearinghouse for Federal Scientific and Technical Information, Springfield, Virginia 22151.
Price: Full size copy $3.00; microfiche copy $ .65.

Perhaps the most natural way to define the "meaning" of strings in a context-free language is to define attributes for each of the nonterminal symbols which arise when the strings are parsed according to the grammatical rules. The attributes of each nonterminal symbol correspond to the meaning of the phrase produced from that symbol. This point of view is expressed in some detail in Knuth (1968a), where attributes are classified into two kinds, "inherited" and "synthesized". Inherited attributes are, roughly speaking, those aspects of meaning which come from the context of a phrase, while synthesized attributes are those aspects which are built up from within the phrase. There can be considerable interplay between inherited and synthesized attributes; the essential idea is that the meaning of an entire string is built up from local rules relating the attributes of each production appearing in the parse of that string. For each production in the context-free grammar, we specify "semantic rules" which define (i) all of the synthesized attributes of the nonterminal symbol on the left hand side of the production, and (ii) all of the inherited attributes of the nonterminal symbols on the righthand side of the production. The initial nonterminal symbol (at the root of the parse tree) has no inherited attributes. Potentially circular definitions can be detected using an algorithm formulated in Knuth (1968a).

The purpose of this paper is to develop these ideas a little further and to present some additional examples of the "inherited attribute - synthesized attribute" approach to formal semantics. The first example defines the class of lambda expressions which have a reduced equivalent,

in terms of a "canonical" reduced form. The second example defines the simple programming language Turingol; this language was defined in Knuth (1968a), in terms of conventional Turing machine quadruples, while the definition in this paper is intended to come closer to the fundamental issues of what computation really is, and to correspond more closely to problems which arise in the definition of large-scale contemporary programming languages.

The formal definitions in this paper are probably not in optimum form, but they seem to be a step in the right direction. It is hoped that the reader who has time to study these examples will be stimulated to develop the ideas further.

1. Lambda expressions

Our first example of a formal definition concerns lambda expressions as discussed by Wegner (1968), restricting the set of variables to the forms x,x',x",x"', etc. Informally, the lambda expressions we consider are either (i) variables standing alone; or (ii) strings of the form λVE, where V is a variable (called a "bound variable") and E is a lambda expression; or (iii) strings of the form (E_1E_2), where E_1 and E_2 are lambda expressions. If E_1 has form (ii), (E_1E_2) denotes functional application, i.e., we may substitute E_2 for all "free" occurrences of V in E, making suitable changes to bound variables within E so that free variables of E_2 do not become bound. For example, $\lambda x(x'x)$ is a lambda expression in which x' is free but x is bound; it has the same meaning as $\lambda x"(x'x")$ by renaming the bound

variable, but $\lambda x'(x'x')$ has a different meaning. The lambda expression $(\lambda x'\lambda x(x'x)x)$ has the same meaning as $(\lambda x'\lambda x"(x'x")x)$, by renaming a bound variable; and this has the same meaning as $\lambda x"(xx")$, by substituting x for x'.

A lambda expression which contains no subexpressions of the form (λVEE_2) is called reduced. Some lambda expressions cannot be converted into an equivalent reduced form; the shortest example is $(\lambda x(xx)\lambda x(xx))$ which goes into itself under substitution. We say a lambda expression is reducible if it is equivalent to some reduced lambda expression. Our goal is to give a formal definition of the class of all reducible lambda expressions; this definition must make precise the notions of "free variables", "bound variables", "renaming", "substitution", etc. Fortunately, it is possible to create such a definition in a fairly natural way, using inherited and synthesized attributes.

Let E be a lambda expression. If E is reducible, our formal definition will define the meaning of E to be a string of characters which is a reduced lambda expression equivalent to E. The definition has the attractive property that two reducible lambda expressions are equivalent if and only if their meanings are exactly identical, character for character. (A proof of this assertion is beyond the scope of this paper, but can be based on the Church-Rosser theorem; cf. Wegner (1968).) The definition is iterative, in that the meaning of E might turn out to be the meaning of another lambda expression E_1; if E is irreducible the process will never terminate, so we will obtain no meaning for E,

but if E is reducible the process will terminate in a finite number of steps. (Again the proof is beyond the scope of this paper, but uses well-known properties of lambda expressions.) It is recursively unsolvable to decide whether or not a given lambda expression is reducible, or if a given lambda expression is equivalent to the reduced form 'x' , so an iterative procedure such as described below is probably the best we can do.

The formal definition involves some more or less standard notation. Let $\underset{\sim}{N}$ be the set of nonnegative integers; $2^{\underset{\sim}{N}}$ is the set of all subsets of $\underset{\sim}{N}$. A <u>string</u> is a sequence of zero or more of the characters

$$x \quad \lambda \quad (\quad ' \quad)$$

and we let ε denote the empty string. The set of all strings is called $\underset{\sim}{T}^*$. A function f is a set of ordered pairs $\{(x,f(x))\}$ whose first components are distinct; $\text{domain}(f) = \{x \mid (x,f(x)) \in f\}$. We write $\emptyset$ for the empty set or empty function;

$$f \uplus g = \{(x,g(x)) \mid x \in \text{domain}(g)\} \cup \{(x,f(x)) \mid x \in \text{domain}(f) \setminus \text{domain}(g)\}$$

denotes the function f "overridden" by the function g . If f is a function taking some subset of $\underset{\sim}{N}$ into $2^{\underset{\sim}{N}}$, and if $S \subseteq \underset{\sim}{N}$, we write

$$\text{image of } S \text{ under } f = \cup \{f(x) \mid x \in S \cap \text{domain}(f)\} \cup \{x \mid x \in S \setminus \text{domain}(f)\} .$$

For example, if $S = \{2,3,4\}$ and $f = \{(2,\{1,4,5\}),(4,\{5,6\})\}$, then image of S under $f = \{1,3,4,5,6\}$.

If n is a nonnegative integer, "var(n)" denotes the string consisting of the letter x followed by n ' characters; thus, var(2) = x" . The number of ' characters is called the <u>index</u> of the variable.

Now we are ready for the formal definition itself; it is convenient to present the definition in a tabular format.

Terminal symbols: x λ (')

Nonterminal symbols: S E V

Start symbol: S

Inherited attributes:

Name of attribute	Type of value	Significance
bound(E)	subset of $\underset{\sim}{N}$	indices of variables whose meaning is bound by the context of E
subst(E)	function from bound(E) into $\underset{\sim}{T}^*$	specifies replacement text for substitutions
substf(E)	function from bound(E) into $2^{\underset{\sim}{N}}$	specifies the indices of free variables in the corresponding replacement text
arg(E)	string	text (if any) used as argument in functional application
argf(E)	subset of $\underset{\sim}{N}$	indices of free variables in arg(E)

Synthesized attributes:

Name of attribute	Type of value	Significance
meaning(S)	string	reduced text of lambda expressions (if it exists)
text(E)	string	string equivalent to E (includes substitutions and reductions)

Synthesized attributes (continued):

Name of attribute	Type of value	Significance
free(E)	subset of $\underset{\sim}{N}$	indices of free variables occurring in E (before substitutions and reductions)
function(E)	true or false	is E explicitly a function?
reduced(E)	true or false	is E reduced?
index(V)	nonnegative integer	number of "primes" in the representation of this variable.

Local variables (used as abbreviations for brevity, in semantic rules 3.2):

Name of variable	Type of value	Significance
mm	nonnegative integer	index chosen as new name of bound variable
rr	subset of $\underset{\sim}{N}$	indices of free variables in ss
ss	string	replacement text

Productions and semantics:

Description	No.	Syntactic Rule	Example	Semantic Rules
Statement	1	S → E	$(\lambda x(xx)\lambda x'(x'x))$	meaning(S) := if reduced(E) then text(E) else meaning(text(E)).
				bound(E) := subst(E) := substf(E) := argf(E) := $\emptyset$.
				arg(E) := ε .

Productions and semantics (continued):

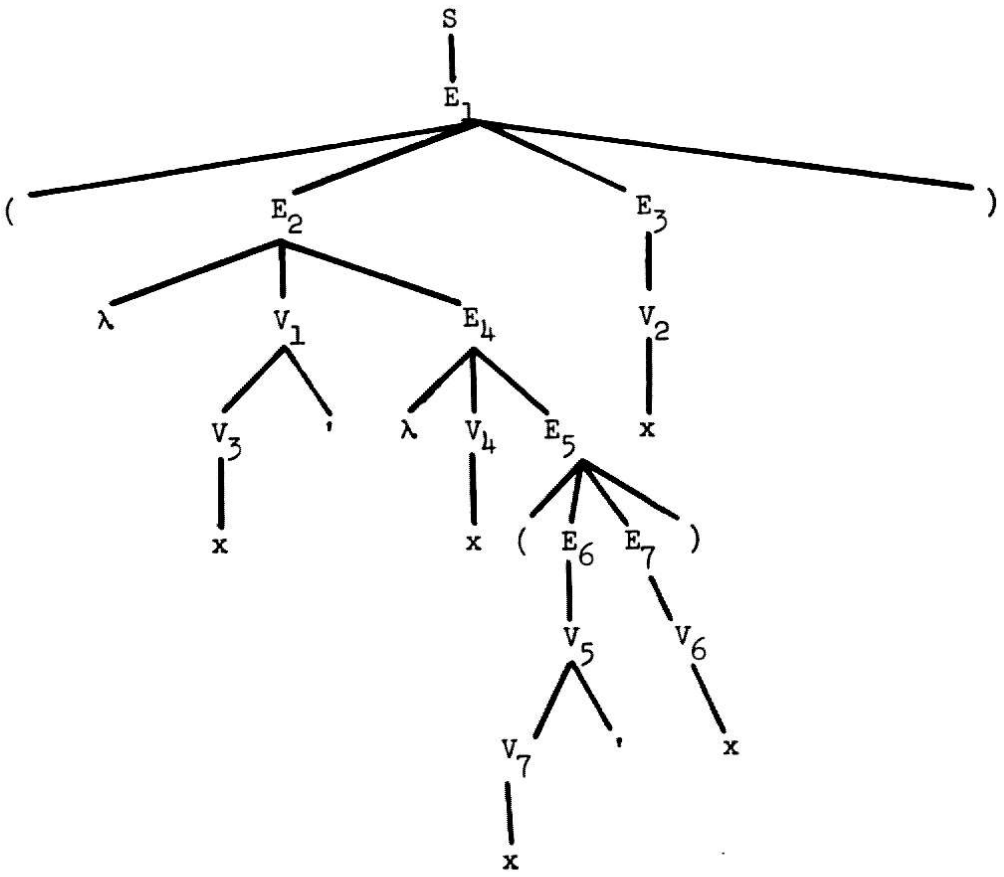
Description	No.	Syntactic Rule	Example	Semantic Rules
Variable	2.1	$V \rightarrow x$	x	index(V) := 0 .
	2.2	$V_1 \rightarrow V_2'$	x'	index(V_1) := index(V_2)+1 .
Expression	3.1	$E \rightarrow V$	x'	function(E) := false . free(E) := {index(V)} . reduced(E) := true . text(E) := if index(V) ∈ bound(E) then subst(E)(index(V)) else var(index(V)) .
	3.2	$E_1 \rightarrow \lambda V E_2$	λx'(x'x)	function(E_1) := true . free(E_1) := free(E_2)\{index(V)} . reduced(E_1) := reduced(E_2) . if arg(E_1) = ε , then (mm := min{k∈N | k∉ image of free(E_1) under substf(E_1)} ss := var(mm) , rr := {mm}) ; else ss := arg(E_1), rr := argf(E_1) . text(E_1) := if arg(E_1) = ε then "λ" ss text(E_2) else text(E_2). bound(E_2) := bound(E_1) ∪ {index(V)} .

Productions and semantics (continued):

Description	No.	Syntactic Rule	Example	Semantic Rules
				$subst(E_2) := subst(E_1) \uplus \{(index(V), ss)\}$.
				$substf(E_2) := subst(E_1) \uplus \{(index(V), tt)\}$.
				$arg(E_2) := \varepsilon$.
				$argf(E_2) := \emptyset$.
	3.3	$E_1 \rightarrow (E_2 E_3)$	(x'x)	$function(E_1) :=$ false .
				$free(E_1) := free(E_2) \cup free(E_3)$.
				reduced(E_1) := if function(E_2) then false else reduced(E_2) $\wedge$ reduced(E_3) .
				text(E_1) := if function(E_2) then text(E_2) else "(" text(E_2) text(E_3) ")" .
				$bound(E_2) := bound(E_3) := bound(E_1)$.
				$subst(E_2) := subst(E_3) := subst(E_1)$.
				$substf(E_2) := substf(E_3) := substf(E_1)$.
				$arg(E_2) := text(E_3)$.
				$argf(E_2) :=$ image of $free(E_3)$ under $substf(E_3)$.
				$arg(E_3) := \varepsilon$.
				$argf(E_3) := \emptyset$.

In rule 1, "meaning(text(E))" stands for meaning(S) in the derivation tree which arises when text(E) is parsed.

As an example of this formal definition, consider finding the "meaning" of $(\lambda x'\lambda x(x'x)x)$. We have the following parse tree, giving integer subscripts to the nonterminal symbols:



The semantic rules define the attributes as follows:

$$index(V_2) = index(V_3) = index(V_4) = index(V_6) = index(V_7) = 0;$$
$$index(V_1) = index(V_5) = 1 .$$

Node	bound	subst	substf	arg	argf	function	free	reduced	text
E_1	$\emptyset$	$\emptyset$	$\emptyset$	ε	$\emptyset$	false	{0}	false	$\lambda x'(xx')$
E_2	$\emptyset$	$\emptyset$	$\emptyset$	x	{0}	true	$\emptyset$	true	$\lambda x'(xx')$
E_3	$\emptyset$	$\emptyset$	$\emptyset$	ε	$\emptyset$	false	{0}	true	x
E_4	{1}	{(1,x)}	{(1,{0})}	ε	$\emptyset$	true	{1}	true	$\lambda x'(xx')$
E_5	{0,1}	F	G	ε	$\emptyset$	false	{0,1}	true	(xx')
E_6	{0,1}	F	G	x'	{1}	false	{1}	true	x
E_7	{0,1}	F	G	ε	$\emptyset$	false	0	true	x'

where $F = \{(0,x'),(1,x)\}$, $G = \{(0,\{1\}),(1,\{0\})\}$. Hence meaning(S) = meaning($\lambda x'(xx')$) , and we must parse $\lambda x'(xx')$. A similar but much simpler derivation shows that meaning($\lambda x'(xx')$) = $\lambda x'(xx')$.

Some of the semantic rules can be eliminated by making the syntax more complicated. For example, the class of reduced lambda expressions is defined by

$$S \rightarrow \lambda VS \mid N$$
$$N \rightarrow (NS) \mid V$$
$$V \rightarrow x \mid V'$$

and the class of nonreduced lambda expressions can be defined similarly. But it seems unwise in general to play such games with the syntax, and in fact as semantic rules become better understood we will probably go the other way and simplify syntax at the expense of semantics.

The above syntax and semantics shows that inherited and synthesized attributes can interact to provide a natural solution to a rather complicated problem. But since they define lambda expressions in terms of lambda expressions, it may be argued that they do not come to grips with the problem of what lambda expressions really mean. Another alternative is discussed below.

2. Turingol

A simple little language that describes Turing machine programs was introduced in Knuth (1968a), where a semantic definition based on quadruples was given. The following example program gives the flavor of this "Turingol" language; it is a program designed to add unity to the binary integer which appears just left of the initially scanned square:

```
tape alphabet is blank, one, zero, point;
print "point";
go to carry;
test: if the tape symbol is "one" then
      {print "zero";
       carry: move left one square; go to test};
print "one";
realign: move right one square;
   if the tape symbol is "zero" then go to realign.
```

It is worthwhile to search for a formal definition of Turingol which goes more deeply into the essential nature of computation itself, instead of assuming the knowledge of an artificial representation of Turing machines based on quadruples. The mapping from Turingol to quadruples is nontrivial and worthy of attention, but it is only part of the problem. Therefore we shall now consider some approaches to the "total" problem of a Turingol definition.

One way to define Turingol, which we shall criticize later, is to introduce an intermediate language called TL/I; we can define Turingol in terms of TL/I and then we can define TL/I in terms of "conceptual computation". TL/I is a machine-like language, consisting essentially of sequential instructions whose operation codes are print, move, if, jump, and stop. The example TL/I program below is almost self-explanatory, so we shall turn immediately to the formal definition of Turingol.

It is convenient to let the symbol ν denote any positive integer, and to let the symbol σ stand for any string of alphabetic letters. These quantities could be syntactically defined, and we could make use of their attributes value(ν) , text(σ) , but for simplicity we may ignore such elementary operations and we can identify numbers and letter strings with their representations. (In other words we are assuming the existence of a "lexical scanning" mechanism, which must exist in some primitive form anyway to recognize the terminal symbols. We could in the same way have dispensed with index(V) in the lambda expression example above.)

The definition below involves "global" quantities, which may be regarded as attributes of the start symbol at the root of the parse tree although their values are accessible at any node. All actions on global quantities can be reduced to a sequence of semantic rules relating appropriate local attributes, but it is simpler and more natural to abbreviate these rules using global quantities.

Global quantities may be global variables, global sets, or global counters. If ξ is a global variable and α is an expression, the notation

$$\underline{\text{define}}\ \xi \equiv \alpha$$

stands for a definition of ξ. (A string of the language is "semantically erroneous" if its parse tree causes any global variable ξ to be defined more than once, or if any undefined global variable is used in an expression.)

If Σ is a global set, the notation

$$\alpha \epsilon \Sigma$$

denotes inclusion of the value of expression α in the set Σ. If κ is a global counter, the notation

$$\kappa + \alpha$$

denotes increase of the value of κ by the value of the integer expression α. Global sets start out empty, and global counters start out zero; when they appear in expressions, their value denotes the accumulated result of all inclusion or increasing operations specified in the entire parse tree. (Note that two inclusion or increasing operations can be done in any order.)

Here finally is a formal definition of Turingol in terms of TL/I:

Terminal symbols: σ . , : ; { } " " tape alphabet is print move left right one square go to if the symbol then

Nonterminal symbols: P S L D O

Start symbol: P

Inherited attributes:

Name of attribute	Type of value	Significance
init(S), init(L)	positive integer	'address' of beginning of this statement or list.

Synthesized attributes:

fin(S), fin(L)	positive integer	'address' following this statement or list.
index(D)	positive integer	number of symbols in declaration.
d(O)	left or right	a direction.

Global variables:

label(σ), for all σ	positive integer	address associated with the identifier σ.
symbol(σ), for all σ	positive integer	symbol number associated with the identifier σ.

Global counter:

nsymb	integer	number of symbols declared in this program

Productions and semantics:

Description	No.	Syntactic Rule	Example	Semantic Rules
Declarations	1.1	D → tape alphabet is σ	tape alphabet is helen	index(D) := 1. nsymb + 1. define symbol(σ) ≡ 1.
	1.2	$D_1 \rightarrow D_2,\sigma$	tape alphabet is helen, phyllis, pat	index(D_1) := index(D_2)+1 . nsymb + 1 . define symbol(σ) ≡ index(D_1) .
Print statement	2.1	S → print "σ"	print "pat"	(init(S) : print, symbol(σ)) ∈ objprog. fin(S) := init(S)+1.
Move statement	2.2	S → move 0 one square	move left one square	(init(S) : move, d(0)) ∈ objprog. fin(S) := init(S)+1.
	2.2.1	0 → left	left	d(0) := left.
	2.2.2	0 → right	right	d(0) := right.
Go statement	2.3	S → go to σ	go to pieces	(init(S) : jump, label(σ)) ∈ objprog. fin(S) := init(S)+1.
Null statement	2.4	S →		fin(S) := init(S).

Productions and semantics (continued):

Description	No.	Syntactic Rule	Example	Semantic Rules
Conditional statement	3.1	S_1 → if the tape symbol is "σ" then S_2	if the tape symbol is "phyllis" then print "pat"	(init(S) : if, symbol(σ), fin(S_2)) ∈ objprog. init(S_2) := init(S_1)+1. fin(S_1) := fin(S_2).
Labeled statement	3.2	S_1 → σ : S_2	pieces: move left one square	define label(σ) := init(S_1). init(S_2) := init(S_1). fin(S_1) := fin(S_2).
Compound statement	3.3	S → {L}	{print "pat"; go to pieces}	init(L) := init(S). fin(S) := fin(L).
List of statements	4.1	L → S	print "pat"	init(S) := init(L). fin(L) := fin(S).
	4.2	L_1 → L_2;S	print "pat"; go to pieces	init(L_2) := init(L_1). init(S) := fin(L_2). fin(L_1) := fin(S).
Program	5	P → D;L.	tape alphabet is helen, phyllis, pat; print "pat".	init(L) := 1. (fin(L) : stop) ∈ objprog.

Example: The Turingol program for binary addition results in setting the global quantities

nsymb = 4	symbol(blank) = 1
label(carry) = 5	symbol(one) = 2
label(test) = 3	symbol(zero) = 3
label(realign) = 8	symbol(point) = 4

and "objprog" is the (unordered) set of 11 strings

(1: print, 4)

(2: jump, 5)

(3: if, 2, 7)

(4: print, 3)

(5: move, left)

(6: jump, 3)

(7: print, 2)

(8: move, right)

(9: if, 3, 11)

(10: jump, 8)

(11: stop)

This set of strings is a TL/I program.

Now we can present a definition of TL/I. For this purpose it is handy to extend context-free syntax slightly, allowing the production

A → set of B

where A and B are nonterminal symbols. This means that A can be, instead of a string (an ordered sequence), a set (unordered) of quantities having the form B .

A doubly-infinite tape, divided into squares and initially containing positive integers in each square, is manipulated by the actions of a TL/I program. There is a pointer which designates a square on the tape. One formal definition of TL/I is based on these concepts and an English language description of the operations to be done, as follows.

Nonterminal symbols: P , S , C

Terminal symbols: ν ( ) , : if print jump move left right

Start symbol: P

Inherited attribute: loc (a positive integer denoting the current position within the program)

Synthesized attributes: meaning (English language description of operations)

Global variables: action(ν) (English language description of operations starting at step ν)

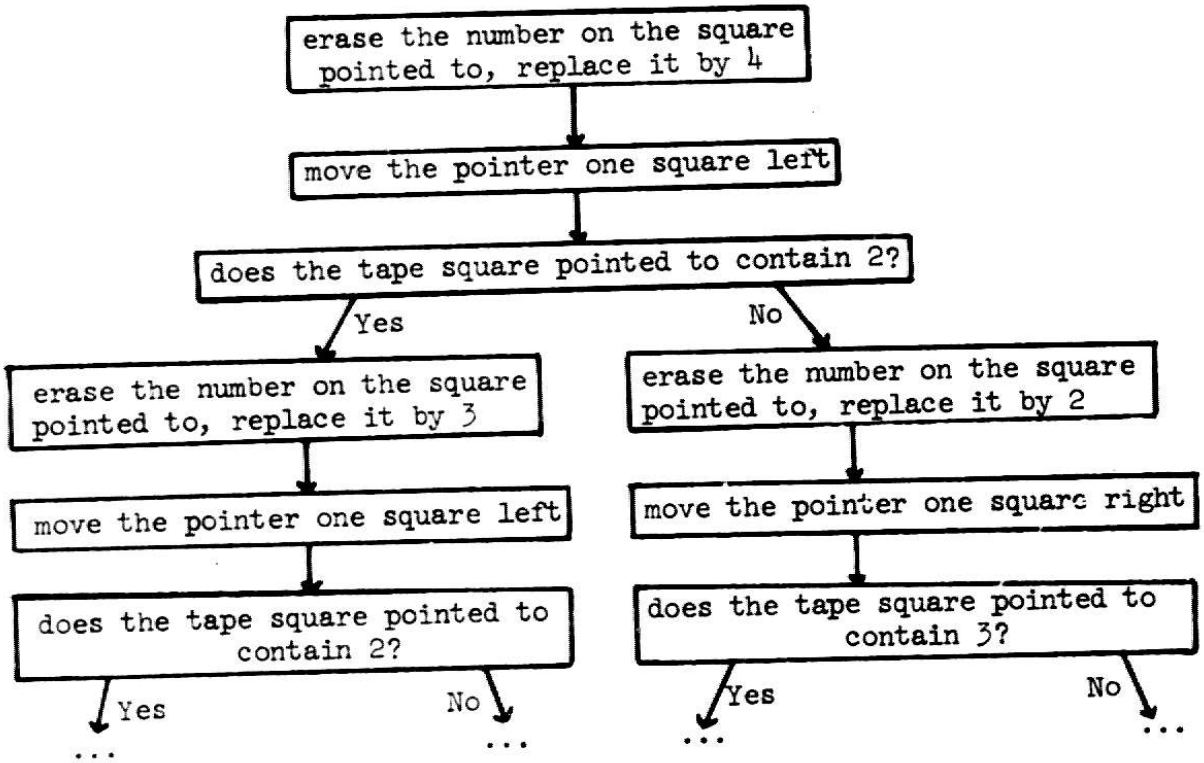
Productions and semantics:

Description	No.	Syntactic Rule	Example	Semantic Rules
Program	1	P → set of S	{(1: print,3), (2: stop)}	meaning(P) := "perform action(1)".
Statement	2	S → (ν:C)	(2: stop)	define action(ν) ≡ meaning(C). loc(C) := ν.
Command	3.1	C → if,ν_1,ν_2	if, 2, 7	meaning(C) := "if the tape square pointed to contains ν_1, then perform action(loc(C)+1); otherwise perform action(ν_2)".
	3.2.1	C → move, left	move, left	meaning(C) := "move the pointer one square left, then perform action(loc(C)+1)".
	3.2.2	C → move, right	move, right	meaning(C) := "move the pointer one square right, then perform action(loc(C)+1)".
	3.3	C → print, ν	print, 3	meaning(C) := "erase the number on the square pointed to, replace it by ν, and then perform action(loc(C)+1)".
	3.4	C → jump, ν	jump, 5	meaning(C) := "perform action(ν)".
	3.5	C → stop	stop	meaning(C) := "stop".

This example has some more or less undesirable properties, if not outright errors, although we can rectify the situation in several interesting ways. If we take the definition literally as it stands, most TL/I programs will have "infinite" meanings; i.e., meaning(P) will never be defined in a finite number of steps and we need to consider a limiting process. Thus, the meaning of our binary addition example comes out to be

> "perform "erase the number of the square pointed to, replace it by 4, and then perform "perform "move the pointer one square left, then perform "perform "if the tape square pointed to contains 2, then perform "replace ..."; otherwise perform "erase the number on the square pointed to, replace it by 2, and then perform "move ..." " " " " " " " .

Being infinite, this doesn't really constitute an English sentence, nor does it read too well! It is essentially an infinite branching structure:



Instead of this infinite branching structure we can take another point of view; rather than expanding the "action" parts of the meanings, we can consider the set of defined actions as a table which constitutes the meaning. Our example program then means "perform action(1)", where

action(1) = "erase the number on the square pointed to, replace it by 4, and then perform action(2)".

action(2) = "perform action(5)".

...

action(11) = "stop".

We can now imagine a man who performs the process specified by a TL/I program, given a doubly infinite tape and a set of defined actions as above; with one hand he points to the action he is currently doing, while his other hand points to a place on the tape (and holds a pencil and an eraser). This is the "ambidextrous man" model of computation.

At this point we can make some observations about the two-level definition of Turingol that appears above. Is it really necessary to introduce something like TL/I, or should we have gone directly to, say, the infinite branching structure or the ambidextrous-man model? A glance at the definitions shows that, indeed, we could have done things in one step. The introduction of TL/I serves only to provide a convenient shorthand, or a conceptual level slightly higher than the base, in which to think about Turing machines, but it is really so close to our ultimate models of computation that it could have been avoided. For more sophisticated languages than Turingol it becomes increasingly more important and helpful to introduce intermediate levels of semantics.

But are our "ultimate" models of computation correct? Some people believe that the user of a programming language should not really understand his program in terms of a TL/I-like list of rules, or a branching structure or flowchart, he should really think of it in terms very close to the source language itself. The ambidextrous man of our model should perhaps be directly interpreting the source language. Such a viewpoint is defensible, but on the other hand it seems to be asking for too much built-in sophistication on the part of the user. He acquires such sophistication only after gaining more experience with programming languages; grade school children can understand simple machine languages but they are not ready for Algol. Perhaps that is the reason many computer science educators are reporting that introductory courses in programming are usually more successful if the students are first taught a simple machine-like language before they learn algebraic languages. They need to understand the underlying principles of computation (what computers do) before seeing "problem oriented languages". Therefore it is likely that the models discussed above aren't too primitive. Furthermore as a practical reality, a person programming well in some current language (FORTRAN, COBOL, ALGOL, PL/I, SNOBOL, etc.) should perhaps think of his program in some terms related to its actual machine representation, so that he knows what different constructions really "cost" him.

If the models aren't too primitive, are they too sophisticated? For example, positive integers should perhaps be defined in terms of Peano's postulates, etc.; maybe all the concepts should be further

formalized in terms of set theory or category theory. This takes things from a domain children can understand into a more formal area which is able to support mechanical proof procedures. In this paper our concern is with finding a natural conceptual basis for definitions; the basis must correspond to the way we actually think about computation, otherwise the related formalisms are not likely to be fruitful. Suitable formalisms will correspond to the natural conceptual basis rather closely, so we need not choose a more primitive formalism.

3. A Digression

Definition of programming language semantics by means of synthesized and inherited attributes is intended to correspond closely to the way people understand that language; the problem of producing compilers for that language is not a main goal, for it is possible to understand the meaning of a language without having to understand how to write a compiler for it. The success of context-free grammars as a model for syntax is based on its natural intuitive appeal (since the syntactic tree structures form a first approximation to the semantic structures), not on the fact that parsing algorithms can be devised for such grammars. A grammar is "declarative" rather than "imperative"; it expresses the essential relationships between things without implying that these relationships have been deduced using any particular algorithm. In general, we want to avoid any preoccupation with bits, advancing pointers, building and unbuilding lists, when such things have little or nothing to do with the intrinsic meaning of the language we are defining. On the

other hand once a "natural" mode of definition has been found, the next step should be to make practical use of it in the automation of software production; it is a happy circumstance when an intuitive description of a system can be almost automatically transformed into a practical working model based on that system. Much work remains to be done on the question of whether formal definitions such as those of this paper can be converted automatically to decent software programs; the following example may be useful as a test case for such techniques.

Consider the probelm of writing an assembler for TL/I, converting a TL/I program into a sequence of bits suitable for interpretation by instructions on a microprogrammed computer. To make the problem interesting, we shall assume that we want to compress the length of the code, letting the number of bits to represent addresses and symbols be a parameter. The following "formal semantics" specifies this transformation precisely, in a problem-oriented fashion.

Let Memory(n,k) stand for the sequence of k bit positions

Memory(n) Memory(n+1) ... Memory(n+k-1) ,

and let Binary(n,k) stand for the sequence of k bits representing $(n \bmod 2^k)$ in binary notation. Let length(α) denote the length of string α . TL/I can now be defined as follows:

Nonterminal symbols, terminal symbols, start symbol: As before.

Synthesized attributes: code(C) , a string of bits representing a coded instruction. length(C) , the number of bits in code(C) .

Global variables:

Memory(ν) , a bit representing part of the encoded program.

loc(ν) , a positive integer representing the first bit location of an instruction.

a , a positive integer representing the size of address specifications.

b , a nonnegative integer representing the size of symbol specifications.

nsymb , number of symbols (computed by the Turingol definition).

Global counters:

addrs , the number of address fields in the program,

bits , the number of bits in non-address fields of the program.

Productions and semantics:

Syntactic Rule	Semantic Rules
P → set of S	define loc(1) ≡ 1. a := $\min\{k\epsilon\tilde{N} \mid \text{bits} + k\cdot\text{addrs} \leq 2^k\}$. b := $\min\{k\epsilon\tilde{N} \mid \text{nsymb} \leq 2^k\}$.
S → (ν:C)	Memory(loc(ν),length(C)) := code(C). define loc(ν+1) ≡ loc(ν)+length(C).
C → if, ν_1,ν_2	code(C) := 00 Binary(ν_1-1,b) Binary(loc(ν_2),a). bits + (b+2); addrs + 1; length(C) := 2+b+a.
C → move, left	code(C) := 010; bits + 3; length(C) := 3.
C → move, right	code(C) := 011; bits + 3; length(C) := 3.
C → print, ν	code(C) := 10 Binary(ν-1,b); bits + (2+b). length(C) := 2+b.
C → jump, ν	code(C) := 11 Binary(loc(ν),a). bits + 2; addrs + 1; length(C) := 2+a.
C → stop	code(C) := 11 Binary(0,a); length(C) := 2+a. bits + 2; addrs + 1.

Note that these rules specify a three pass process (first we count the addrs , then we can compute a and the loc's , then we can fill in the addresses) in a compact "declarative" manner.

4. Information structures

The above definitions have adhered to old fashioned ways to represent information (integers, functions, sets, etc.), while Computer Science suggests that we ought to use some slightly different models and develop their formalisms further. A wide variety of applications (see, for example, Knuth (1968b), Chapter 2) suggests that it is useful to represent the information in the real world, and its structural interrelationships, by means of things called "nodes". Each node consists of several "fields", which contain values; the values may be integers, strings, sets, etc., but (more importantly) the values may be references (i.e., pointers or links) to other nodes. The idea of references can be and has been formalized in various ways in terms of classical concepts, but recent experience suggests the usefulness of regarding references themselves as primitives. This often frees us from making arbitrary but conceptually irrelevant choices when we are representing information; for example, index sets are often used in mathematics when they really don't belong, and integers were used in our definition of lambda expressions and Turingol above although we really wanted only unique labels and a notion of order.

Let us therefore consider making semantic definitions in terms of the proper data structures. A study of the Turingol-to-TL/I example shows

that we should replace the set of location-instruction pairs in TL/I by a string of nodes (i.e., an ordered sequence of nodes). Each node corresponds to an instruction; jump and if nodes contain references to other nodes. We can concatenate strings of nodes just like strings of letters; so, for example, we can do away with the inherited attribute "init(S)" . The semantics for rule 4.2, $L_1 \rightarrow L_2;S$, becomes simply "meaning(L_1) := meaning(L_2) meaning(S)" . In this way we obtain a more appealing (and more simple) formal definition, because all attributes are synthesized except for those which are implicitly present in global quantities. This idea of node strings containing pointers between the nodes, instead of absolute addresses which have to be determined by strict sequence rules, has been very successful in some studies recently conducted by the author on an experimental compiler-generating language.

Instead of making a complete listing of Turingol's semantics from the string-of-nodes point of view, it is perhaps even more interesting to consider the slightly more complicated problem of translating Turingol into a "self-explanatory flowchart". We may regard the meaning of a Turingol program as a set of nodes whose structure is that of a flowchart, easily readable by any ambidextrous man who wants to perform the algorithm. We use the notation

$$\underset{\sim}{\text{new}}(\xi_1 := \eta_1;\ \xi_2 := \eta_2;\ \ldots;\ \xi_m := \eta_m)$$

to denote the creation of a new node with m fields; the field named ξ_j contains the value η_j , for $1 \le j \le m$. The value of new(...) is

a reference to this node. It is interesting to compare the Turingol definition below with the definition above, since the inherited-vs.-synthesized roles of "init" and "fin" are reversed.

Nonterminal symbols, terminal symbols, start symbol: As before.

Inherited attributes:

fin(S), fin(L)	reference to node which follows statement or list.

Synthesized attributes:

init(S), init(L)	reference to node which begins statement or list.
index(D)	positive integer, the number of symbols in declaration.
d(O)	"left" or "right", a direction.

Global variables:

label(σ), for all σ	reference to the node corresponding to the label identifier σ.
symbol(σ), for all σ	positive integer, the symbol number associated with the identifier σ.

Fields of nodes: The COMMAND field contains strings of words and numbers explaining what to do when reaching this node; the YES, NO, and NEXT fields contain references to other nodes. All nodes generated by the new operation constitute the "meaning" of a Turingol program.

Productions and semantics:

No.	Syntactic Rule	Semantic Rules
1.1	D → tape alphabet is σ	index(D) := 1; define symbol(σ) ≡ 1.
1.2	D_1 → D_2,σ	index(D_1) := index(D_2)+1, define symbol(σ) ≡ index(D_1).
2.1	S → print "σ"	init(S) := new(COMMAND := "Erase the number on the square pointed to, and replace it by symbol(σ); then go on to the NEXT node." ; NEXT := fin(S)).
2.2	S → move O one square	init(S) := new(COMMAND := "Move the pointer one square to the d(O); then go on to the NEXT node."; NEXT := fin(S)).
2.2.1	O → left	d(O) := left.
2.2.2	O → right	d(O) := right.
2.3	S → go to σ	init(S) := new(COMMAND := "Go on to the NEXT node."; NEXT := label(σ)).
2.4	S →	init(S) := "new(COMMAND := "Go on to the NEXT node."; NEXT := fin(S)).
3.1	S_1 → if the tape symbol is "σ" then S_2	init(S_1) := new(COMMAND := "If the tape square pointed to contains symbol(ν), then go on to the YES node; otherwise go on to the NO node."; YES := init(S_2); NO := fin(S_1)). fin(S_2) := fin(S_1).
3.2	S_1 → σ:S_2	init(S_1) := init(S_2); fin(S_2) := fin(S_1). define label(σ) := init(S_1).

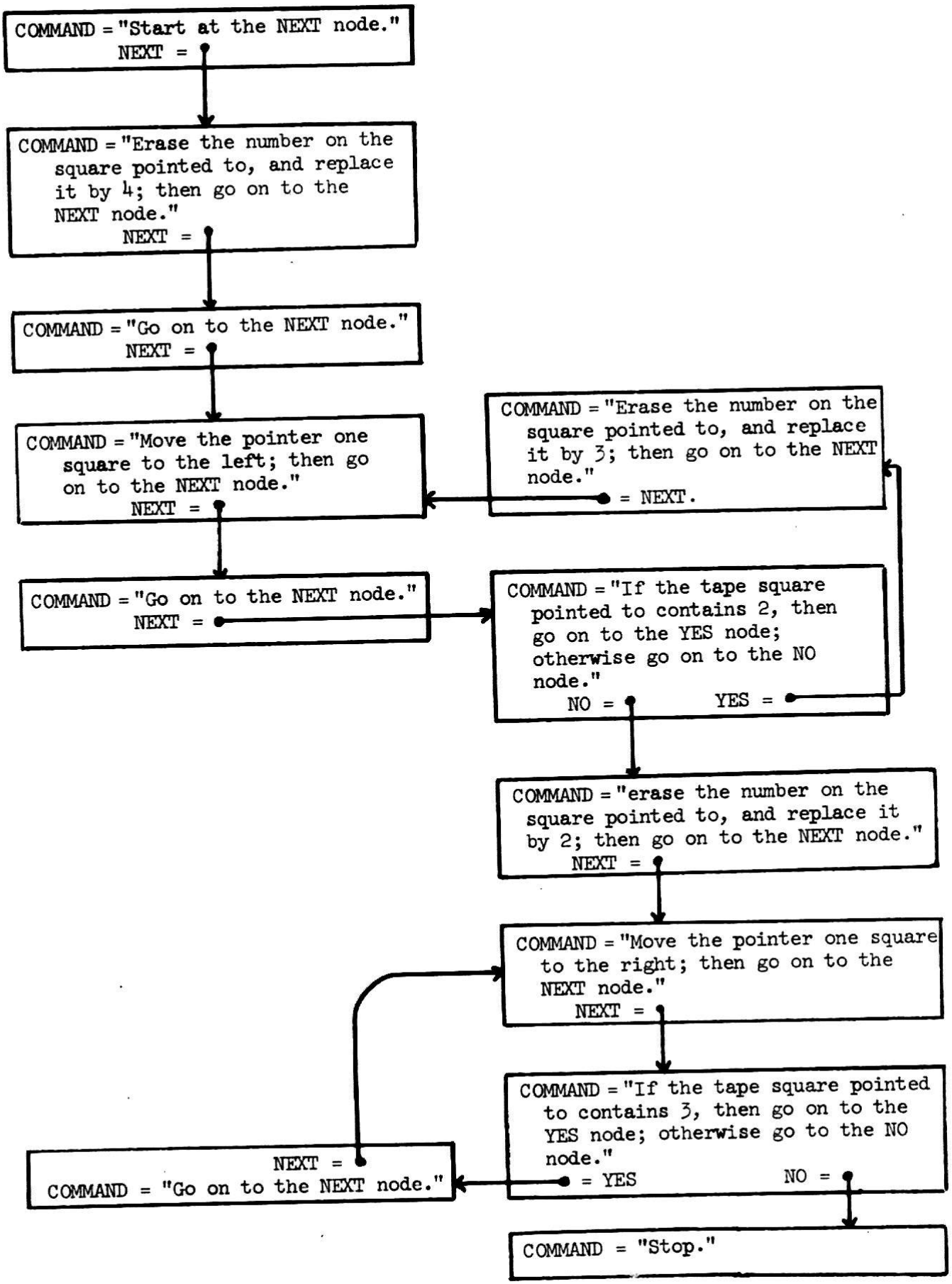
Productions and semantics (continued):

No.	Syntactic Rule	Semantic Rules
3.3	S → {L}	init(S) := init(L); fin(L) := fin(S).
4.1	L → S	init(L) := init(S); fin(L) := fin(S).
4.2	$L_1 \rightarrow L_2;S$	$init(L_1) := init(L_2)$; $fin(L_2) := init(S)$; $fin(S) := fin(L_1)$.
5	P → D;L.	new(COMMAND := "Start at the NEXT node."; NEXT := init(L)). fin(L) := new(COMMAND := "Stop.").

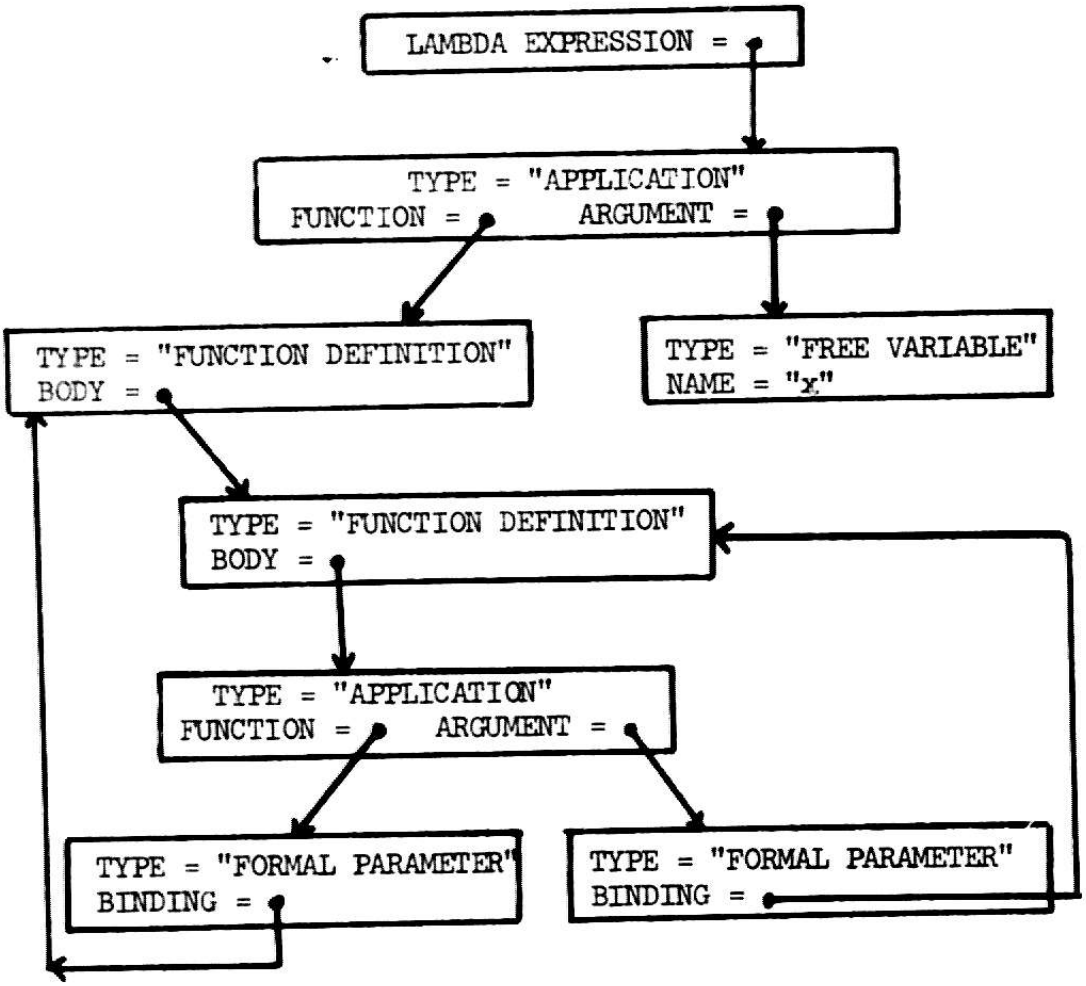
This definition will produce the following flowchart from the binary addition example:

(The flowchart contains three "go on" nodes which seem redundant, although there are cases such as "loop: go to loop" which show that they cannot be eliminated entirely.)

symbol(blank) ≡ 1, symbol(one) ≡ 2, symbol(zero) ≡ 3, symbol(point) ≡ 4.



We could also consider lambda expressions again from the standpoint of appropriate information structure. It is an easy exercise to define the semantics so that, for example, the lambda expression $(\lambda x'\lambda x(x'x)x)$ becomes the structure



Thus, bound variables become "formal parameter" nodes which refer back to the appropriate function definition in which the variable is bound. Such a structure gives the essential content of the lambda notation, except for the definition of functional application which can now be given in various ways in terms of the node structures.

5. Summary

We have discussed several examples in which rather complicated functions have been defined on context-free languages. In each case it was possible to give a definition which is concise, in the sense that hardly anything is defined that isn't "necessary"; and at the same time the definitions seem to be intuitive, in the sense that they mirror the structure by which we "understand" the function. These definitions are based on assigning attributes to the nonterminal symbols of a context-free grammar, and relating the attributes which correspond to each production.

We have also discussed some of the choices for a semantic basis of programming languages. If we are interested in information-theoretic properties of algorithms, we may prefer an infinite branching structure as a computational model; if we are interested in representations of algorithms which are analogous to real live computer programs, we may prefer an "ambidextrous man" (essentially an automaton) model; if we are interested in the underlying structure, we may prefer an flowchart model. Other models are also possible. Whatever the model, formal definition via attributes seems to be helpful.

At the present time these ideas are being used and extended by Wayne Wilner, to construct a formal definition of a major programming language, SIMULA 67. The complexity of this language (over 300 productions in the syntax) makes the semantics slightly less transparent than the examples in this paper, but in fact the definition turns out to be simpler than anticipated.

Perhaps the main direction for future work which is suggested by the examples of this paper is to devise a suitable "context-free grammar" for arbitrary node structures instead of just strings. Attribute-definition on such grammars may lead to a very natural declarative language for problem solving in terms of relevant structures.

Knuth, Donald E. (1968a). "Semantics of Context-Free Languages," Mathematical Systems Theory 2, 127-145.

Wegner, Peter (1968). Programming Languages, Information Structures, and Machine Organization. McGraw-Hill, 401 pp.

Knuth, Donald E. (1968b). The Art of Computer Programming, vol. 1: Fundamental Algorithms. Addison-Wesley, 632 pp.