excerpt from
Semantics of Context-Free Languages

DONALD E. KNUTH
California Institute of Technology

1968

Abstract

"Meaning" may be assigned to a string in a context-free language by defining "attributes" of the symbols in a derivation tree for that string. The attributes can be defined by functions associated with each production in the grammar. This paper examines the implications of this process when some of the attributes are "synthesized", i.e., defined solely in terms of attributes of the *descendants* of the corresponding nonterminal symbol, while other attributes are "inherited", i.e., defined in terms of attributes of the ancestors of the nonterminal symbol. An algorithm is given which detects when such semantic rules could possibly lead to circular definition of some attributes. An example is given of a simple programming language defined with both inherited and synthesized attributes, and the method of definition is compared to other techniques for formal specification of semantics which have appeared in the literature.

Only section 4 is included, as the rest of the document is peripheral to Turingol. These excerpts are from the article "Semantics of context-free languages" *Mathematical Systems Theory* (1968), 127–145. Errata, *Mathematical Systems Theory* (1971), 95–96.

4 A simple programming language.

Now let us consider an example of how the above techniques of semantic definition can be applied to programming languages. For simplicity let us study a formal definition of a little language that describes Turing machine programs.

A Turing machine (in the classical sense) processes an infinite tape which may be thought of as divided into squares; the machine can read or write characters from a finite alphabet on the tape in the square which is currently being scanned, and it can move the scanning position to the left or right. The following program, for example, adds unity to an integer number, assuming that the square just to the right of the number is to be scanned at the beginning and end of the program:

> **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 hoped that the reader will find this programming language sufficiently self-explanatory that he understands it before any formal definition of the language is given, although of course this is not necessary. The above program is not intended as an example of good programming, rather as an example of the features of the simple language considered in this section.)

Since every programming language must have a name, let us call the language Turingol. Any well-formed Turingol program defines a program for a Turing machine; let us say a Turing machine program consists of

a set Q of "states";
a set Σ of "symbols";
an "initial state" $q_0 \in Q$;
a "final state" $q_\infty \in Q$;

and a "transition function" δ which maps $(Q - \{q_\infty\}) \times \Sigma$ into $\Sigma \times \{-1, 0, +1\} \times Q$. If $\delta(q,s) = (s', k, q')$ we may say informally that, if the machine is in state q and scanning symbol s, it will print symbol s', move k spaces to the right(meaning one space to the left if $k = -1$), and go into state q'. Move formally, a Turing machine program defines a computation on any "initial tape contents", i.e., on any doubly infinite sequence

$$\cdots, a_{-3}, a_{-2}, a_{-1}, a_0, a_1, a_2, a_3, \cdots \tag{1}$$

of elements of Σ, as follows: At any moment of the computation there is a "current state" $q \in Q$ and an integer-valued "tape position" p; initially $q = q_0$ and $p = 0$. If $q \neq q_\infty$, and if $\delta(q, a_p) = (s', k, q')$, the computation proceeds by replacing the value of a_p by s', then by replacing p by $p+k$ and q by q'. If $q = q_\infty$, the computation terminates. (The computation might not terminate; for program (4.1) this happens if and only if $a_j =$ "*one*" for all $j < 0$.)

Now we have a precise definition of Turing machine programs, we wish to define the Turing machine program corresponding to any given Turingol program (and at the same time to define the syntax of Turingol). For this purpose it is convenient to introduce a few abbreviation conventions.

1. The semantic rule **"include** x **in** B**"** associated with a production will mean that x is to be a member of set B, where B is an attribute of the start symbol S of the grammar. The value of B will be the set of all x for which such a semantic rule has appeared corresponding to each appearance of the production in the derivation tree. (The rule may be regarded as an abbreviation for the semantic rule

$$B(X_{p0}) = \bigcup_{j=1}^{n_p} B(X_{pj}) \cup \{x | \text{"}\textbf{include } x \textbf{ in } B\text{"} \text{ is associated with the } p\text{-th production}\} \tag{2}$$

added to each production, with a set B added as a synthesized attribute of each nonterminal symbol, and $B(x)$ the empty set for each terminal symbol. These rules clearly make $B(S)$ the desired set.)

2. The semantic rule **"define** $f(x) = y$**"** associated with a production will mean that y is to be the value of the function f evaluated at x, where f is an attribute of the start symbol S of the grammar. If two rules occur defining $f(x)$ for the *same* value of x, this is an error condition, and any derivation tree which allows this condition to occur may be said to be *malformed.* Furthermore, f may be used as a function in any other semantic rules, with the proviso that $f(x)$ may only appear when f has been defined at x; any derivation tree which calls for an undefined value of $f(x)$ is *malformed.* (This type of rule is important, for example, to ensure that there is agreement between the declaration and use of identifiers. In the example below this convention implies that programs are malformed if the same identifier is used twice as a label or if a **go to** statement specifies an identifier which is not a statement label. The rule may essentially be thought of as **"include** (x, y) **in** f**"**, as in 1, if f is regarded as a set of ordered pairs; additional checks for malformedness are also included. We may regard "well-formed or malformed" as an attribute of S; appropriate semantic rules analogous to (4.3) which completely specify this **"define** $f(x) = y$**"** convention are readily constructed and left to the reader.)

3. The function **"newsymbol"** appearing in any semantic rule will have, as its value, an abstract element which for each evaluation of **"newsymbol"** is different from the abstract element produced by other evaluations of **newsymbol**. (This convention can readily be expressed in terms of other semantic rules, e.g., by making use of the l attributes of (2.3) which has a different value at each node of a tree. The function **newsymbol** serves as a convenient source of "raw material" for constructing sets.)

We have observed that conventions 1, 2, 3 can be replaced by other constructions of semantic rules which do not use these conventions, so they are not "primitives" for semantics. But they are of fairly wide utility, since they correspond to concepts which are often needed, so they may be regarded as fundamental aspects of the techniques for semantic definition presented in this paper. The effect of using there conventions is to reduce the number of attributes that explicitly mentioned and to avoid unnecessarily long rules.

Now it is a simple matter to present a formal definition of the syntax and semantics of Turingol.

Nonterminal symbols: P (program), S (statement), L (list of statements), I (identifier), O (orientation), A (alphabetic character), D (declaration).

Terminal symbols: a b c d e f g h i j k l m n o p q r s t u v w x y z . , : ; " " { } **tape alphabet is print go to if the symbol then move left right one square**

Start symbol: p

Attributes:

Name of attribute	*Type of value*	*Purpose*
Q	Set	States of the program
Σ	Set	Symbols of the program
q_0	Element of Q	Initial state
q_∞	Element of Q	Final state
δ	Function from $(Q - q_\infty) \times \Sigma$ into $\Sigma \times \{-1, 0, +1\} \times Q$	Transition function
label	Function from strings of letters into elements of Q	State table for statement labels
symbol	Function from strings of letters into elements of S	Symbol table to tape symbols
follow	Element of Q	State immediately following statement or list of statements
d	± 1	Direction
text	String of letters	Identifier
start	Element of Q	State at the beginning of a statement or list of statements (an inherited attribute).

Productions and semantics: See Table 1.

Notice that two states correspond to each statement S: start(S) is the state corresponding to the first instruction of the statement (if any), and it is an inherited attribute of S; follow(S) is the state which "follows" the statement, the state which is normally reached after the statement is executed. In the case of a "goto statement", however, the program does not transfer to follow(S), since the action of the statement is to change control to another place; follow(S) may be said to follow statement S "statically" of "textually", not "dynamically" during a run of the program.

In Table 1, follow(S) is a synthesized attribute; it is possible to give similar semantic rules in which follow(S) is inherited, although a less efficient program would be obtained for null statements (see Rule 4.4). Similarly, both start(S) and follow(S) could be synthesized attributes, but at the expense of additional instructions in the Turing machine program for statement lists (Rule 6.2).

This example would be somewhat simpler if we had used a less standard definition of Turing machine instructions. The definition we have used requires reading, printing, and shifting in each instruction, and also makes the Turing machine into a kind of "one-plus-one-address computer" in which each instruction specifies the location (state) of the next instruction.

Table 1.

Description	*No.*	*Syntactic Rule*	*Example*	*Semantic Rules*
Letters	1.1	$A \to a$	a	text(A) = a.
		...	(similarly for all letters)	
	1.26	$A \to z$	z	text(A) = z.
Identifiers	2.1	$I \to A$	m	text(I) = text(A).
	2.2	$I \to IA$	*marilyn*	text(I) = text(I)text(A).
Declarations	3.1	$D \to$ **tape alphabet is** I	**tape alphabet is** *marilyn*	**define** symbol(text(I)) = **newsymbol;** **include** symbol(text(I)) **in** S.
	3.2	$D \to D, I$	**tape alphabet is** $marilyn, jayne, birgitta$	**define** symbol(text(I)) = **newsymbol;** **include** symbol(text(I)) **in** S.
Print statement	4.1	$S \to$ **print** "I"	**print** "*jayne*"	**define** δ(start(S), s) = (symbol(text(I)), 0, follow(S)) for all $s \in \Sigma$; follow(S) = **newsymbol;** **include** follow(S) in Q.
Move statement	4.2	$S \to$ **move** O **one square**	**move left one square**	**define** δ(start(S),s)=(s, d(O), follow(S)) for all $s \in \Sigma$; follow(S) = **newsymbol;** **include** follow(S) **in** Q.
	4.2.1	$O \to$ **left**	**left**	d(O) = -1.
	4.2.2	$O \to$ **right**	**right**	d(O) = $+1$.
Go statement	4.3	$S \to$ **go to** I	**go to** *boston*	**define** δ(start(S,s) = (s, 0, label(text(I)) for all $S \in \Sigma$; follow(S) = **newsymbol;** **include** follow(S) **in** Q.
Null statement	4.4	$S \to$		follow(S) = start(S).
Conditional statement	5.1	$S_1 \to$ **if the tape symbol is "**I**" then** S_2	**if the tape symbol is "***marilyn***" then print "***jayne***"**	**define** δ(start(S_1),s) = (s, 0, follow(S_2)) for all $s \in \Sigma -$ symbol(text(I)); **define** δ(start(S_1),s) = (s, 0, start(S_2)) for s = symbol(text(I)); start(S_2) = **newsymbol;** follow(S_1) = follow(S_2); **include** start(S_2) **in** Q.
Labeled statement	5.2	$S_1 \to I : S_2$	*boston* : **move left one square**	**define** label(text(I)) = start(S_1); start(S_2) = start(S_1); follow(S_1) = follow(S_2).
Compound statement	5.3	$S \to \{L\}$	{**print** "*jayne*";; **go to** *boston*}	start(L) = start(S); follow(S) = follow(L).
List of statements	6.1	$L \to S$	**print** "*jayne*"	start(S) = start(L); follow(S) = follow(L).
	6.2	$L_1 \to L_2;\ S$	**print** "*jayne*";; **go to** *boston*	start(L_2) = start(L_1); follow(L_2) = **newsymbol;** **include** follow(L_2) in Q; start(S) = follow(L_2); follow(L_1) = follow(S).
Program	7	$P \to D;\ L.$	**tape alphabet is** $marilyn, jayne, birgitta$; **print** "*jayne*".	q_0 = **newsymbol;** **include** q_0 **in** Q; start(L) = q_0; q_∞ = follow(L).

The method of defining semantic rules in this example, with an inherited "first(S)" and a synthesized "follow(S)" attribute, lends itself readily also to computers or automata in which the $(n+1)$st instruction normally is performed after the n-th. Then (follow(S) $-$ start(S)) would be the number of instructions "compiled" for statement S.

This definition of Turingol seems to approach the desirable goal of stating almost exactly the same things which would appear in an informal programmer's manual explaining the language, except that the description is completely formal and unambiguous. In other words, this definition perhaps corresponds to the way we actually understand the language in our minds. The Definition 4.1 of a print statement, for example, might be freely rendered in English as follow:

"A statement may have the form

print "I"

where I is an identifier. This means that, whenever this statement is executed, the tape symbol on the currently scanned square will be replaced by the symbol denoted by I, regardless of what symbol was being scanned; afterwards the program will continue with a new instruction, which is defined (by other rules) to be the instruction following this statement."