Acta Informatica 1, 79–110 (1971)
© by Springer-Verlag 1971

Top-Down Syntax Analysis* **

DONALD E. KNUTH

Received March 29, 1971

Summary. The theory and practice of classical "top-down" parsing methods is presented in a tutorial manner.

1. Introduction

Since the earliest days when automatic syntax analysis by computer was first attempted, many people (e.g. Barnett [1], Brooker and Morris [2], Glennie, Conway [3], Schorre [4]) have used a method which has become known as "top-down" analysis. The idea is still popular, and it is being used in many current compilers.

The methods of the authors just cited may be described conveniently in terms of a little computerlike device which we shall the Parsing Machine. The discussion in these lectures will deal only with the *syntactic* properties of the top-down method of analysis, not with the manner in which the syntactic structure is later used to obtain semantic information about the string which is being analyzed. Semantic information is, of course, the real reason why syntactic analysis is done in the first place; and top-down analysis is popular chiefly because it lends itself so conveniently to semantic extensions. However, let us accept this fact on faith, and concentrate only on the syntactic aspects. For further details, see the references cited above and [5, volume 5 and 7].

It is important to mention that we shall be principally concerned with unambiguous context-free grammars, which occur frequently in programming languages and in input data formats for data-processing systems. The purpose of these lectures is to point out some interconnections between theory and practice, and to analyze the situations in which simple top-down syntax-oriented methods can be guaranteed to work.

Section 2 introduces an abstract machine which resembles the interpretive routines often used for top-down syntax analysis. Section 3 shows how BNF grammars define programs for this machine in a natural way. Section 4 examines the problem of proving such programs correct.

---

* This paper is essentially a transscript of five expository lectures which were presented at the NATO International Summer School on Computer Programming, in Copenhagen, Denmark, August, 1967. The author wishes to thank V. Tixier and R. Guedj for their assistance in preparing the first draft of these lecture notes.

** The publication of this paper was supported in part by IBM Corporation.

Section 5 is an exposition of the basic theory of context-free grammars. Section 6 shows how to decide simple properties of grammars, and Section 7 gives graph-theoretic constructions which are useful for grammatical analysis. These results are applied in Section 8 to characterize all grammars for which a "no-backup" program for the abstract machine is valid.

Section 9 contrasts top-down and bottom-up analysis from a theoretical point of view, and Section 10 gives formal definitions of $LR(k)$ and $LL(k)$ grammars. Section 11 shows that the $LL(1)$ languages are precisely those readable without backup by the machine of Section 2. Final observations and research problems are stated in Section 12.

2. The Parsing Machine (PM)

The Parsing Machine is an abstract machine which is designed to analyze strings over a certain alphabet. It scans an "input string" one character at a time, from left to right, according to a program. A Parsing Machine program is made up of a family of procedures calling each other recursively; the program itself is one of these procedures. Each procedure attempts to find an occurrence of a particular syntactic type in the input, and it returns with the value "true" or "false" depending on whether it has been successful or not.

Let the input string be $s_1 s_2 \ldots s_n$, and let s_h be the "current" character being scanned by the machine.

All instructions have three fields: an op-code field, and two addresses, AT and AF. Procedures are written using two types of instructions, corresponding to two different forms of the op-code.

First Type: The op-code is a letter of the alphabet, a.

Second Type: The op-code is the location of a procedure enclosed within square brackets $[A]$.

The effects of these instructions are as follows.

Type 1: **if** $s_h = a$ **then** move past a (i.e., set $h := h+1$) and **go to** AT
else go to AF.

Type 2: call on the procedure which starts in location A (recursively);
if it returns with value true **then go to** AT
else if it returns with the value false **then go to** AF.

Each AT or AF field can contain either a location of an instruction, or one of the two special symbols T or F. If it contains a T the procedure returns with the value true. If it contains an F, the procedure returns with the value false, and h is *reset* to the value it had when the procedure was called. (This implies that the value of h is saved together with the return address, whenever a procedure is called by an op-code of Type 2.)

An example program for the Parsing Machine should help to make these definitions clear. In all our examples, we shall write PM instructions in an ad hoc assembly language, using symbolic addresses and labels. A blank address refers to the location of the instruction which appears on the line that immediately follows the blank address.

Consider the following grammar for a language that bears some resemblance to "Boolean expressions". (This grammar is written in a modified BNF notation, using "→" in place of "::=" and using capital letters in place of syntactic types enclosed in brackets; for example, the first rule might be rewritten

$$\langle\text{Boolean expression}\rangle ::= \langle\text{relation}\rangle \mid (\langle\text{Boolean expression}\rangle)$$

in BNF notation.)

$$\begin{aligned} &B \to R \mid (B) \\ &R \to E = E \\ &E \to a \mid b \mid (E + E) \end{aligned} \tag{2.1}$$

Using the assembly language described above, we can write a corresponding Parsing Machine program:

loc	op-code	AT	AF
B	$[R]$	T	
	(		F
	$[B]$		F
	)	T	F
R	$[E]$		F
	=		F
	$[E]$	T	F
E	a	T	
	b	T	
	(		F
	$[E]$		F
	+		F
	$[E]$		F
	)	T	F
S	$[B]$		ERROR
	⊣	OK	ERROR

Note the correspondence between the grammar and the PM program. The last two lines of the program correspond to a further grammatical rule

$$S \to B\dashv$$

where "⊣" is a special right delimiter symbol which appears only at the end of the string being analyzed. The procedure S will go to "OK" if the entire string being analyzed is a B followed by ⊣, otherwise it will go to "ERROR".

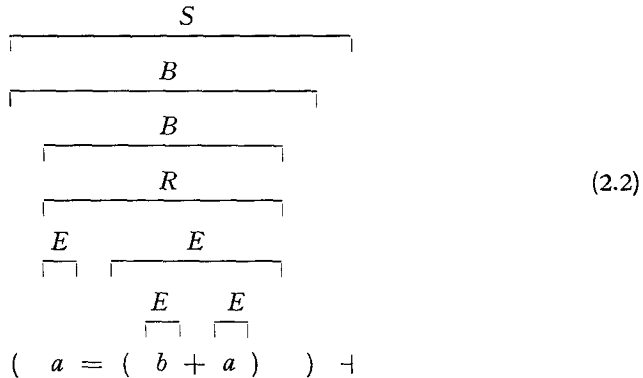
If we set this PM program to work on the input string

$$(a = (b + a))\dashv = s_1\, s_2 \ldots s_{10}$$

starting at location S, the sequence of actions begins as follows. (Initially $h = 1$; i.e., the first character s_1 is being scanned.)

Call B $(h=1)$
    Call R $(h=1)$
        Call E $(h=1)$
            Look for a: no
            Look for b: no
            Look for (: yes, set $h:=2$
            Call E $(h=2)$
                Look for a: yes, set $h:=3$
                Return, true.
            Look for $+$: no
            Return, false; set $h:=1$
        Return, false; set $h:=1$
    Look for (: yes, set $h:=2$
    Call B $(h=2)$
        Call R $(h=2)$

and so on. Ultimately the program will go to the location "OK", and the history of procedure calls with true returns will correspond to the following diagram ("parsing") of the input:



(2.2)

This diagram may be thought of as being constructed from the top to bottom by the Parsing Machine program.

The left portion of each bracket in the diagram is constructed when calling a procedure, and the right portion is completed when returning from that procedure.

A study of this program should convince the reader that if $s_1 \ldots s_n$ is *any* sequence of the letters $\{a, b, (, +,), =, \dashv\}$, with only s_n equal to $\dashv$, the PM program goes to OK when $s_1 \ldots s_n$ is of the form $B\dashv$, and its history of true returns corresponds to a parse diagram; otherwise the PM program goes to ERROR. But this assertion requires proof! There are some grammars for which the corresponding PM program will *not* work correctly, as we shall see. Therefore we want to examine the general question, "For what grammars will the corresponding PM program work?"

3. PM Programming

To analyze this question, we must state carefully what we mean by the "corresponding PM program."

First assume that, as in the above example, all BNF rules have been written in the *standard form*

$$X \to Y_1 \,|\, Y_2 \,|\, \ldots \,|\, Y_m \,|\, Z_1 Z_2 \ldots Z_n \qquad (3.1)$$

where $m, n \geqq 0$, $m+n>0$, and the Y's and Z's are either terminal characters (i.e. letters of the alphabet) or nonterminal symbols (i.e., syntactic types). The righthand side of (3.1) has $m+1$ alternatives; if $m=0$, it has the simple form

$$X \to Z_1 Z_2 \ldots Z_n$$

If $n=0$ the string $Z_1 Z_2 \ldots Z_n$ is to be regarded as the *empty* string. The PM program corresponding to a rule in standard form consists of the following $m+n$ instructions:

loc	op-code	AT	AF
X	$[Y_1]$	T	
	$[Y_2]$	T	
	$\vdots$	$\vdots$	
	$[Y_m]$	T	*
	$[Z_1]$		F
	$\vdots$		$\vdots$
	$[Z_{n-1}]$		F
	$[Z_n]$	T	F

(3.2)

Brackets around Y_j or Z_j in these op-codes should be removed when Y_j or Z_j is a terminal symbol. The address denoted by "*" is to be replaced by "T" if $n=0$, otherwise it should be left blank.

The specification (3.2) has to be modified in the trivial case when both $m=0$ and $n=0$; then the rule is $X \to \varepsilon$, where ε denotes the empty string, and the procedure X should always return true without advancing h. The latter effect can be achieved by the PM program

$$\begin{array}{cccc} X & [Q] & T & T \\ Q & a & F & F \end{array}$$

where a is any terminal letter. Such anomalies are unimportant to the theory, and they disappear when semantic operations are added to the Parsing Machine's repertoire (see [5, volume 5]).

When a BNF rule is not in standard form, suppose that it has the form

$$X \to \alpha_1 \,|\, \ldots \,|\, \alpha_m \,|\, Z_1 Z_2 \ldots Z_n \qquad (3.3)$$

where $\alpha_1, \ldots, \alpha_m$ represent strings of terminal or nonterminal symbols. Then we can change it into standard form by introducing new nonterminal symbols $Y_1, \ldots, Y_m$, adding the rules

$$\begin{array}{c} Y_1 \to \alpha_1 \\ \cdots\cdots\cdots \\ Y_m \to \alpha_m \end{array}$$

and replacing (3.3) by (3.1). For example if our BNF grammar has a rule

$$X \to A\,B \,|\, C\,D$$

we change it to the two rules

$$\begin{array}{l} X \to Y \,|\, C\,D \\ Y \to A\,B \end{array}$$

This allows the PM to back up if it has found an A which is not followed by B, so that it can try the other alternative CD.

It is important to observe that the order in which the PM rules are listed can drastically affect the behavior of the machine. For example, if we have a production

$$X \to a \mid ab \tag{3.4}$$

the PM program will never recognize the string ab as X, since it will return true once it finds the first alternative a. This production might therefore be written

$$X \to ab \mid a$$

and transformed into standard form.

A better idea is perhaps to avoid making the machine back up, by "factoring" this rule into

$$\begin{aligned} X &\to aB \\ B &\to b \mid \varepsilon \end{aligned} \tag{3.5}$$

Further problems can still arise, however, if there is also another rule

$$Y \to Xbb \tag{3.6}$$

present in the grammar. Then it becomes impossible for the PM to know whether X should be a or ab, without looking ahead to see how many b's follow. This can lead to serious difficulties, which we will consider later; fortunately in many practical situations these pathological problems do not arise.

The principle of "factoring" which is shown in (3.5) is of some importance in simplifying and speeding up PM programs. It is convenient to rewrite (3.5) as

$$X \to a \llbracket b \mid \varepsilon \rrbracket$$

making use of "meta-brackets" $\llbracket$ and $\rrbracket$ to group alternatives together so that it is unnecessary to give a special name (like B in (3.6)) to the new syntactic type.

Consider now the following rule:

$$X \to a \mid bck \mid bdk \mid befik \mid beghik \mid befjk \mid beghjk$$

This can be factored as

$$X \to a \mid b \llbracket c \mid d \mid e \llbracket f \mid gh \rrbracket \llbracket i \mid j \rrbracket \rrbracket k \tag{3.7}$$

The introduction of factors in this case does not require procedure calls; only branching is necessary, since the following PM program can be written for X:

X	a	T	
	b		F
	c	X_1	
	d	X_1	
	e		F
	f	X_2	
	g		F
	h	X_2	F
X_2	i	X_1	
	j	X_1	F
X_1	k	T	F

It can be shown that simplifications of this kind can always be made if we redefine standard form (3.1) so that any of the Z's may be factored quantities which themselves are in standard form. Rule (3.7) is an example of this more general kind of standard form.

Another simplification can be made when we have the "closure" operator $A*$, meaning "zero or more occurrences of A in a row", i.e. ε or A or AA or AAA, etc. The corresponding syntactic rule is

$$A* \to AA* \mid \varepsilon$$

which, by our previous conventions, must be expanded into the following rather long PM program, where Y corresponds to $AA*$:

loc	op-code	AT	AF
$A*$	$[Y]$	T	T
Y	$[A]$		F
	$[A*]$	T	F

A much faster code can be written which obviously is equivalent except that it saves a great many subroutine calls:

$$A* \qquad [A] \qquad A* \qquad T \tag{3.8}$$

We can extend the definition of standard form, (3.1), further, so that each Z is allowed to be also of the form $W*$ where W is a single symbol (terminal or nonterminal).

The two simplifications just discussed, namely factoring and closure, are instances of a general programming rule which allows us to replace a procedure call by a "go to" when this call is the last act of another procedure.

As an example, consider the ALGOL 60 definition of an unsigned number:

$$\begin{aligned} D &\to 0|1|2|3|4|5|6|7|8|9 \\ U &\to DU' \\ U' &\to U|\varepsilon \\ P &\to \cdot U \\ P' &\to P|\varepsilon \\ S' &\to +|-|\varepsilon \\ E &\to {}_{10}S'U \\ E' &\to E|\varepsilon \\ M &\to P|UP' \\ N &\to E|ME'. \end{aligned} \tag{3.9}$$

Here N is ALGOL's ⟨unsigned number⟩, E is ⟨exponent part⟩, P is ⟨fraction part⟩, etc. A slight change has been made to the definition of U, ⟨unsigned integer⟩, since ALGOL's definition $U \to UD|D$ would get the PM into a loop. This phenomenon is called "left recursion", which is the bane of top-down analysis; left recursion is analyzed further below.

Grammar (3.9) may be factored into the following standard form:

$$D \to 0|1|2|3|4|5|6|7|8|9$$
$$U \to D D^*$$
$$P \to \cdot U$$
$$E \to {}_{10}[\![+|-|\varepsilon]\!]\, U$$
$$N \to E\,|\,[\![P|U[\![P|\varepsilon]\!]]\!]\,[\![E|\varepsilon]\!]$$

The corresponding PM program is

loc	op-code	AT	AF
D	0	T	
	1	T	
	⋮		
	9	T	F
U	$[D]$	U'	F
U'	$[D]$	U'	T
P	$\cdot$	U	F
E	$_{10}$		F
	$+$	U	
	$-$	U	U
N	$[E]$	T	
	$[P]$	$N1$	
	$[U]$		F
	$[P]$	$N1$	$N1$
$N1$	$[E]$	T	T

and it runs much more efficiently than the PM program corresponding to (3.9). Note that the above code involves another simplification, in that procedure P was not written

P	$\cdot$		F
	$[U]$	T	F

Exercise 1.* For which set of strings does the following PM program go to "OK", starting at S?

loc	op-code	AT	AF
A	a		T
	$[A]$		F
	b	T	F
B	$[A]$		T
	c	F	T
C	b		T
	$[C]$		F
	c	T	F
S	$[B]$	ERROR	
D	a	D	
	$[C]$		ERROR
	$\dashv$	OK	ERROR

* Answers to the exercises appear at the close of this paper.

4. The Partial Back-Up Problem

Examples (3.4) and (3.6) show that the Parsing Machine's limited back-up capability makes it unsuitable for general BNF grammars. But the example of Boolean expressions in Section 2 shows that the PM can handle a reasonably wide range of grammars of practical interest, and we now return to the question posed at the end of that section: "For which BNF grammars, converted into standard form (3.1) by the technique of (3.3) and then converted into PM programs by the definition (3.2) supplemented by the inclusion of a right delimiter $\dashv$ symbol as in Section 2, does the corresponding PM program accept precisely the strings belonging to the language defined by the grammar?" (In other words, we go to OK if the string is in the language, otherwise to ERROR.)

Unfortunately this problem is *unsolvable*, i.e., there is no effective algorithm which decides (from a given grammar) whether or not the PM program will always work.

Proof. Let $\alpha_1, \ldots, \alpha_m, \beta_1, \ldots, \beta_m$ be strings of a's and b's; and let $z_1, \ldots, z_m$, a, b, x be the terminal letters of our alphabet. Consider the following rules:

$$
\begin{aligned}
&A \rightarrow z_1\alpha_1|\ldots|z_m\alpha_m|z_1 A\,\alpha_1|\ldots|z_m A\,\alpha_m\\
&B \rightarrow z_1\beta_1|\ldots|z_m\beta_m|z_1 B\beta_1|\ldots|z_m B\beta_m\\
&C \rightarrow A\,x\\
&D \rightarrow B\,x\,x\\
&E \rightarrow C|D\\
&S \rightarrow E\dashv
\end{aligned}
$$

Here A represents the strings

$$L(A) = \{z_{i_n} \ldots z_{i_2} z_{i_1} \alpha_{i_1} \alpha_{i_2} \ldots \alpha_{i_n}\}$$

for $n = 1, 2, 3, \ldots$; and B similarly represents

$$L(B) = \{z_{i_n} \ldots z_{i_2} z_{i_1} \beta_{i_1} \beta_{i_2} \ldots \beta_{i_n}\}.$$

The whole language S is $A\,x\dashv$ or $B\,x\,x\dashv$.

Consider a string α belonging to A; the PM will recognize that αx belongs to E.

Consider α not belonging to A but belonging to B; the PM will recognize that $\alpha x x$ belongs to E.

Consider α belonging to both A and B; $\alpha x x$ will not be recognized as belonging to E, although it does.

Therefore the partial back-up method will work for this language if and only if $L(A) \cap L(B) = \emptyset$. This happens if and only if there do not exist indices $i_1, \ldots, i_n$, $n > 0$, such that $\alpha_{i_1} \ldots \alpha_{i_n} = \beta_{i_1} \ldots \beta_{i_n}$.

But this is "Post's Correspondence Problem," and it is well known that no effective algorithm can decide if such indices exist. If we could solve the partial back-up problem, we could solve Post's problem, but that is impossible. This completes the proof.

Although the partial back-up problem is unsolvable, we can of course solve it in special cases. Sufficient conditions which can be used in practical situations

are given in [5, volume 5]. The most important special case of the partial back-up problem is the "No backup problem" which we will solve below.

A method of top-down analysis which includes "full backup", i.e. which works on all BNF grammars that are not left recursive, has been elegantly described by Floyd [6]; we will not treat Floyd's general method here, since most cases of practical interest for programming languages can be done with little or no backing up.

5. Context-Free Grammars

At this point it is convenient to introduce (in a somewhat more careful manner than in the previous sections) the basic definitions of context-free grammars, together with some notations associated with the mathematical theory of languages. We are going to solve a special case of an unsolvable problem, so it is worthwhile to prepare ourselves for this task.

An alphabet X is a set of distinguishable symbols, and X^* denotes the set of all strings on the alphabet X, i.e. all sequences $x_1 \ldots x_n$ for $n \geqq 0$, where each x_j is in X. It is convenient to denote strings of symbols by lower case Greek letters $\alpha, \beta, \ldots$; as we have already observed, the empty (or "null") string is denoted by ε. The *length* of a string α, written $|\alpha|$, is the number of symbols it contains. When α and β are strings, their concatenation $\alpha\beta$ is the string obtained by writing the symbols of β in order after the symbols of α. It follows from these definitions, for example, that

$$|\varepsilon| = 0, \qquad |\alpha\beta| = |\alpha| + |\beta|.$$

A set of strings is usually denoted by a capital letter, such as $A, B, \ldots$. The concatenation of two sets of strings is defined by the rule

$$A\,B = \{\alpha\beta \mid \alpha \in A \text{ and } \beta \in B\}. \tag{5.1}$$

Note that

$$A\{\varepsilon\} = \{\varepsilon\}A = A$$

and

$$A\,\emptyset = \emptyset A = \emptyset.$$

(The symbol $\emptyset$ denotes the empty set.)

We now define "powers" of a set of strings:

$$A^n = \textbf{if } n = 0 \textbf{ then } \{\varepsilon\} \textbf{ else } A\,A^{n-1}. \tag{5.2}$$

Two further operations of importance are the *closure* A^* and the *positive closure* A^+ of a set of strings:

$$A^* = \lim_{n \to \infty} [A^0 \cup A^1 \cup \cdots \cup A^n] = \bigcup_{n \geqq 0} A^n \tag{5.3}$$

$$A^+ = \bigcup_{n \geqq 1} A^n \tag{5.4}$$

Note that

$$A^+ = A\,A^* = A^*A, \qquad A^* = \{\varepsilon\} \cup A^+.$$

A *context-free grammar* $\mathscr{G}$ has four parts:

(a) A *terminal alphabet* T, whose elements are denoted here by lower case letters $a, b, c, \ldots$ and occasionally by special symbols auch as parentheses and plus signs.

(b) A *nonterminal alphabet* N, whose elements are denoted here by upper case letters $A, B, C, \ldots$.

(c) An *initial symbol* S, which is a nonterminal symbol that represents the "sentences" of the language defined by G.

(d) A set of *productions* $\mathscr{P}$, which is the most important part of the grammar $\mathscr{G}$. A production is a relation denoted by

$$A \to \theta \tag{5.5}$$

(read, "A directly produces θ"), where $A \in N$ and $\theta \in (N \cup T)^*$; i.e., A is a nonterminal symbol and θ is a string of terminals and/or nonterminals.

Each set of productions $\mathscr{P}$ defines a relation on the strings $(N \cup T)^*$; we say

$$\alpha A \omega \to \alpha \theta \omega \tag{5.6}$$

(with respect to $\mathscr{P}$) if $A \to \theta$ is a production of $\mathscr{P}$. In other words, we say that $\varphi \to \psi$ if and only if there are strings $\alpha, \omega, A, \theta$ such that $\varphi = \alpha A \omega$, $\psi = \alpha \theta \omega$, and $A \to \theta$ is in $\mathscr{P}$.

The relation $\varphi \to \psi$ between strings, defined in (5.6), can be extended as follows: We say

$$\varphi \to^* \psi \tag{5.7}$$

if $\varphi = \psi$, or if $\varphi \to \psi$, or if there is another string ξ such that $\varphi \to \xi \to \psi, \ldots$, or in general if

$$\varphi = \varphi_0, \quad \varphi_j \to \varphi_{j+1} \text{ for } 0 \leqq j < n, \text{ and } \varphi_n = \psi \tag{5.8}$$

for some $n \geqq 0$ and some strings $\varphi_0, \varphi_1, \ldots, \varphi_n$. Relation (5.7) may be read, "φ produces or equals ψ". Similarly we write

$$\varphi \to^+ \psi \tag{5.9}$$

if (5.8) holds for some $n \geqq 1$ and some strings $\varphi_0, \ldots, \varphi_n$. Relation (5.9) may be read, "φ produces ψ"; it excludes the case $n = 0$, which in (5.8) is the trivial case that $\varphi = \psi$. Note the analogy between (5.7)–(5.9) and (5.3)–(5.4).

In general if r denotes any relation between members of any set, we obtain the *reflexive transitive closure* r^* and the *transitive closure* r^+ of r as a new relation which is often of interest, by using definitions (5.7), (5.8), (5.9) and replacing "$\to$" by r.

Now we are ready to define the significance of a context-free grammar $\mathscr{G} = (T, N, S, P)$. The *language* $L(\mathscr{G})$ defined by $\mathscr{G}$ is the set

$$L(\mathscr{G}) = \{\tau \in T^* \mid S \to^+ \tau\}, \tag{5.10}$$

i.e., the set of all terminal strings which the initial symbol produces.

If θ is any string of terminals and/or nonterminals, we also write

$$L(\theta) = \{\tau \in T^* \mid \theta \rightarrow^* \tau\} \tag{5.11}$$

with respect to an understood context-free grammar $\mathcal{G}$.

We say that

$$A \rightarrow \theta_1 \mid \theta_2 \mid \ldots \mid \theta_n \tag{5.12}$$

is a *rule* of the grammar $\mathcal{G}$ if and only if

$$\{A \rightarrow \theta_1, A \rightarrow \theta_2, \ldots, A \rightarrow \theta_n\}$$

is the set of all productions of $\mathcal{G}$ whose lefthand side is A.

The reader should be able to see the connection between context-free grammars, as defined here, and BNF syntax specifications as in the ALGOL Report. The only difference is in the notational conventions.

As an example of a context-free grammar, consider the following rules

$$\begin{aligned} &E \rightarrow L [\![+ L]\!]^* \\ &L \rightarrow P\,P^* \\ &P \rightarrow a \mid b \mid (E) \end{aligned}$$

written in terms of the factoring and closure conventions of the previous section. From now on we will eliminate these conventions, in order to make the theory simpler to develop without decreasing our power of expression; the above grammar can be written

$$\begin{aligned} &E \rightarrow L\,L' \\ &L' \rightarrow + L\,L' \mid \varepsilon \\ &L \rightarrow P\,P' \\ &P' \rightarrow P\,P' \mid \varepsilon \\ &P \rightarrow a \mid b \mid (E) \\ &S \rightarrow E \dashv \end{aligned} \tag{5.13}$$

Note the introduction of the last rule, according to the conventions of our Parsing Machine.

The set of productions (5.13) may be said to define a context-free grammar $\mathcal{G}$ whose six terminal symbols are

$$a,\ b,\ +,\ (,\ ),\ \dashv$$

and whose six nonterminal symbols are

$$E,\ L',\ L,\ P',\ P,\ S.$$

The initial symbol is S. The grammar has six *rules* and ten *productions*. (Note the distinction between a rule and a production, see (5.12).) The language defined by $\mathcal{G}$ resembles simple arithmetic expressions; a typical element of $L(\mathcal{G})$ is

$$a(b + a\,b) \dashv$$

6. The Null String Problem

One of the first things we can do with a context-free grammar is to determine which nonterminal symbols can produce the null string; i.e., given a nonterminal symbol A, does $A \to^+ \varepsilon$ or not?

A simple "marking" algorithm applies to this problem. We can imagine all nonterminal symbols as either "marked" or "unmarked", where initially all are unmarked. Now we repeatedly do the following operation: Find a production in $\mathscr{P}$ whose lefthand side is unmarked, and whose righthand string contains nothing but marked symbols. (In particular, ε is such a string. Terminal symbols are regarded as unmarked.) If no such productions exist, the algorithm terminates; otherwise, mark the nonterminal symbol on the left of the production which was found, and repeat the process.

At the conclusion of this algorithm, a nonterminal A can produce ε if and only if it is marked. For it is clear that every marked nonterminal produces ε. Conversely, if $A \to^+ \varepsilon$ in n steps, then if $n=1$, A must be marked; and if $n>1$, we have some θ such that $A \to \theta \to^+ \varepsilon$. Here each symbol in θ must produce ε in less than n steps, so by induction on n each symbol in θ is marked; hence A is marked.

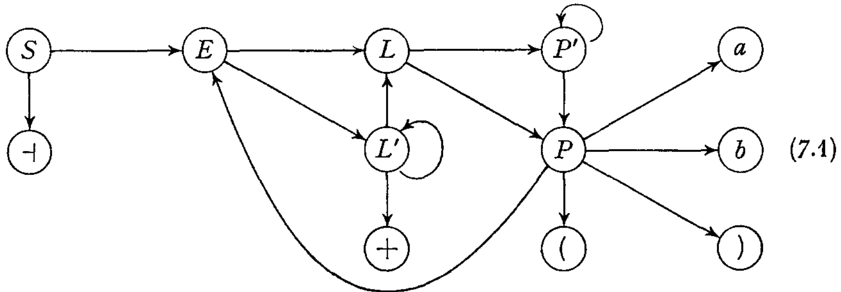
Essentially the same algorithm can be used to determine whether or not $L(A)$ is empty, i.e. whether A produces any terminal strings or not. We use the same procedure, except that all terminal symbols are regarded as if they were marked.

7. Directed Graphs

A *directed graph* is defined by a set of *vertices* and a set of ordered pairs of vertices called *arcs*. Each arc may be thought of as an arrow drawn from one vertex to another vertex (or to the same vertex). An *oriented path* in a directed graph from vertex V to vertex W is a sequence of vertices $V_0, \ldots, V_n$, such that $V=V_0$, there is an arc from V_j to V_{j+1} for $0 \leqq j < n$, $V_n = W$, and $n \geqq 1$. There obviously are algorithms to determine whether or not there is an oriented path from V to W, given two vertices V and W of a finite directed graph, since we need only consider paths which go through each vertex at most once.

Given any context-free grammar $\mathscr{G}$, we can draw its *dependency graph*. Here the vertices are the terminal and nonterminal symbols, and the arcs go from A to x if x appears on the righthand side of a production whose lefthand side is A.

For the grammar of example (5.13) we have the dependency graph



(7.1)

There is an obvious correspondence between directed graphs and binary relations on objects of an abstract set S. If we have a relation r, we can consider the

directed graph whose vertices are the elements of S and whose arcs go from V to W if and only if VrW. Conversely each directed graph defines a relation on its vertices. There is a path from V to W if and only if Vr^+W, in the notation of Section 5.

In the case of context-free grammars, let us write

$$XdY \quad (\text{"}X \text{ directly depends on } Y\text{"})$$

if and only if there is an arc from X to Y in the dependency graph; i.e., if and only if there is a production rule $X\to\alpha Y\beta$ in the grammar, for some strings α and β.

Now Xd^+Y ("X depends on Y") is easily seen to be equivalent to the statement that

$$X\to^+\alpha Y\omega$$

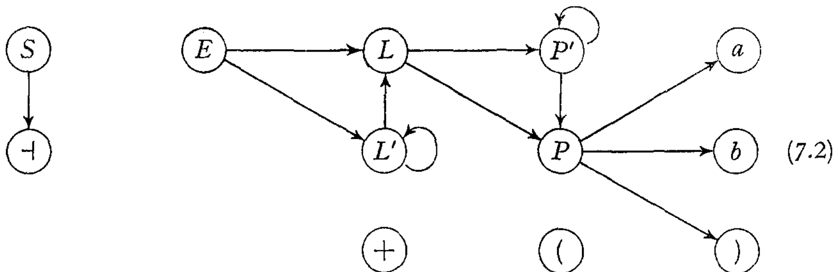
for some strings α and ω. If Xd^+X, we say X is *recursive* (it depends on itself). This means there is an oriented cycle in the dependency graph. In (7.1), we see that all nonterminals are recursive except S.

A nonterminal symbol A of $\mathscr{G}$ is called *useless* if either $L(A)=\emptyset$ (i.e., no terminal strings can be derived from A), or if S does not depend on A (i.e., the strings derivable from A have no effect on $L(\mathscr{G})$). The discussion above shows that we can determine all useless nonterminal symbols. These (and all productions involving them) can be removed from the grammar with no effect on the language, provided that S itself is not useless.

In addition to the dependency graph, we can also define the *right-dependency graph*, which is a subgraph of the dependency graph of a grammar. In this case we draw an arc from X to Y if and only if there is a production of the form

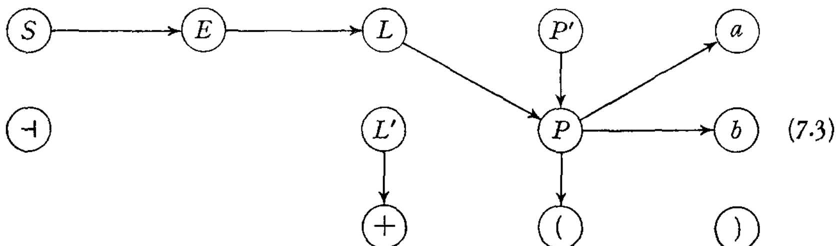
$$X\to\alpha YX_1\dots X_n$$

where $n\geqq 0$, and where each of $X_1,\dots,X_n$ can produce the null string. For the grammar (5.13), we have the right-dependency graph



(7.2)

Interchanging left and right in these definitions gives the *left-dependency graph*:



(7.3)

Let us say XlY ("X directly left-depends on Y") if there is an arc from X to Y in the left-dependency graph; Xl^+Y ("X left-depends on Y") is seen to be equivalent to saying that

$$X \to^+ Y\alpha$$

for some string α. We say that X is *left recursive* if Xl^+X. Similarly, we define XrY, Xr^+Y, etc. from the right-dependency graph. In grammar (5.13), we see from (7.3) that no nonterminals are left recursive. The graph (7.2) shows that L' and P' are right recursive.

The dependency graphs can be used to determine several quantities of interest to us. If A is a nonterminal symbol, let first(A) denote the set of all terminal symbols which can be the initial character of a string in $L(A)$. It is clear from the above discussion that

$$\text{first}(A) = \{a \in T \mid A\, l^+ a\}$$

so we can read off the first characters of any nonterminal by inspecting (7.3). Similarly, last(A) can be obtained from a consideration of the right dependency graph.

Finally, we want to define the set

$$\text{follow}(A) = \{a \in T \mid S \to^* \theta A\, a\, \varphi \text{ for some strings } \theta,\ \varphi\}.$$

It is not difficult to see that, when there are no useless nonterminals, this is equivalent to saying follow(A) is the set of all terminal a such that there is a production of the form

$$W \to \alpha X B_1 \ldots B_n Y \omega \qquad (7.4)$$

where $n \geqq 0$, B_1 through B_n can produce the null string, Y is either terminal or nonterminal, Xr^*A, and Yl^*a. This means we can compute the set follow(A).

(The reason (7.4) can be assumed is that we may consider the production in the derivation of $S \to^* \theta A\, a\, \varphi$ which "combines" A and a.)

For the example grammar (5.13), we have

A	first(A)	last(A)	follow(A)
S	$a\ b$ (	⊣	
E	$a\ b$ (	$a\ b$)	) ⊣
L	$a\ b$ (	$a\ b$)	+) ⊣
P	$a\ b$ (	$a\ b$)	+ $a\ b$ ( ) ⊣
L'	+	$a\ b$)	) ⊣
P'	$a\ b$ (	$a\ b$)	+) ⊣

8. The No-Backup Case

In Section 4 we showed that it was, in general, difficult (in fact impossible) to decide when the Parsing Machine will properly parse all strings belonging to the language defined by a context-free grammar $\mathscr{G}$. The tools developed in Sections 5, 6, and 7 now give us enough ammunition to attack the most important special case of the partial backup problem, namely when the Parsing Machine never has to back up at all; i.e., when h never is decreased. This has two practical consequences: First, we can be sure that the total time required for syntactic

analysis is bounded by a constant times the length of the input string. Second, a computer program may read the input one character at a time and need not save the characters previously read.

Analysis of the general PM program (3.2) corresponding to a standard form program shows that h can back up only when a false exit occurs after $Z_2, Z_3, \ldots,$ or Z_n. This suggests that we redefine (3.2) as follows:

$$\begin{array}{llll} X & [Y_1] & T & \\ & \vdots & & \\ & [Y_m] & T & * \\ & [Z_1] & & F \\ & [Z_2] & & \text{ERROR} \\ & \vdots & & \\ & [Z_n] & T & \text{ERROR} \end{array} \tag{8.1}$$

We can now take any context-free grammar $\mathscr{G}$ with rules grouped as in (3.3); these rules can be transformed into standard form (3.1) and the corresponding "no-backup" program (8.1) can be constructed. The auxiliary rule introducing "$\dashv$" can also be added as before.

We would now like our machine to behave as follows, when it starts at location S, scanning $s_1 s_2 \ldots s_n$ where only s_n is the "$\dashv$" symbol: If $s_1 \ldots s_n$ is not in $L(\mathscr{G})$, the program should go to ERROR. If $s_1 \ldots s_n$ *is* in $L(\mathscr{G})$, and if we have any diagram such as (2.2) which corresponds to a derivation $S \to^* s_1 \ldots s_n$, then the actions of the PM should correspond precisely to that diagram (in the obvious manner).

If the no-backup PM program satisfies the conditions of the preceding paragraph, we shall say "The no-backup method works for $\mathscr{G}$." Note that this condition implies in particular that $\mathscr{G}$ is *unambiguous*, i.e. that no two different diagrams can be given for the strings of $L(\mathscr{G})$. For if there are two diagrams, the PM program is supposed to correspond to both of them, and this clearly cannot happen since the PM executes only one set of actions. Unambiguous grammars are of principal interest for programming languages.

We now wish to answer the question, "Does the no-backup method work for $\mathscr{G}$?" when $\mathscr{G}$ is given. In order to study this problem, we first want to replace a grammar with rules of the form (3.1) by another grammar whose rules are of a more simple form, and which generates the same language with only a slightly different structure.

Given a rule $X \to Y_1 | \ldots | Y_m | Z_1 Z_2 \ldots Z_n$, we can simplify it as follows: If $m \geqq 1$, change the rule to the two rules

$$X \to Y_1 | X'$$
$$X' \to Y_2 | \ldots | Y_m | Z_1 \ldots Z_n$$

This does not change the essential behavior of the no-backup method, it just introduces redundant procedure calls. We can now suppose that $m \leqq 1$, and that all rules have the form

$$X \to Z_1 \ldots Z_n$$
$$X \to Y | Z$$

If $n>1$ in the first form, replace the rule by the two rules

$$X \to Z_1 X'$$
$$X' \to Z_2 \dots Z_n$$

where X' is another new nonterminal symbol.

Again the no-backup method is unaffected. We may assume now that each rule are of one of four types

$$X \to Y|Z$$
$$X \to YZ$$
$$X \to Y$$
$$X \to \varepsilon$$

If Y or Z is a terminal (e.g., $X \to a|Z$), then introduce a new rule of the form $X' \to a$ and change the Y or Z to the X' (e.g., $X \to X'|Z$).

All rules now have one of five forms:

Type 1. $X \to Y|Z$
Type 2. $X \to YZ$
Type 3. $X \to Y$
Type 4. $X \to a$
Type 5. $X \to \varepsilon$

Here Y, Z are nonterminal symbols and a is terminal. Moreover, we may assume that no "useless" non-terminals are present. Let us say that a grammar satisfying these conditions is *simple*.

8.1 Necessary Conditions

We are now going to study four necessary conditions on grammars of the simple form we have just defined, which must be satisfied if the no-backup method works. Later we will prove that these four conditions are also sufficient, and this will solve the no-backup problem.

First Condition. *No nonterminal is left recursive.*

Otherwise it will be necessary for procedure X to call itself without advancing the input, when the machine is mimicking a derivation of $S \to^* \alpha X \omega \to^+ \alpha X \theta \omega \to^+ \tau$, where τ is terminal. (Such strings τ, α, ω, and θ exist because X is assumed to be left-recursive but not useless.)

Lemma. *Whenever a grammar has no left recursive nonterminal symbols, it is possible to order the nonterminal symbols $X_1, X_2, \dots, X_t$ in such a way that*

$$X_p l^+ X_q \text{ only if } q<p.$$

(In our example (5.13), L', P, P', L, E, S is such an ordering.)

Proof. This is a general result about the vertices of a finite directed graph which contains no oriented cycles. If there is no left recursion, the left dependency

graph has no oriented cycles. The vertices of such a graph can always be ordered $V_1, V_2, \ldots, V_t$ in such a way that there is a path from V_p to V_q only if $q<p$. (This is the problem of "topological sorting" which is analyzed in more detail in [5, volume 1].)

We can always find a vertex from which no arcs emanate, otherwise we could find an oriented cycle. Such a vertex may be placed first in the ordering and removed from the graph, and this operation may be repeated until all vertices have been removed. The lemma has therefore been proved.

Let us now restate the five types of rules under this ordering assumption:

Type 1.	$X_p \to X_q \mid X_r$	$q<p$ and $r<p$.
Type 2.	$X_p \to X_q X_r$	$q<p$; and if $X_q \to^+ \varepsilon$, also $r<p$.
Type 3.	$X_p \to X_q$	$q<p$.
Type 4.	$X_p \to a$	
Type 5.	$X_p \to \varepsilon$	

Another condition must be satisfied if the no-backup method works, as illustrated in the following grammar.

$$\begin{aligned}&X \to Y \mid Z\\ &Y \to Y_1 Y_2\\ &Z \to Z_1 Z_2\\ &Z_1 \to a\\ &Z_2 \to b\\ &Y_1 \to a\\ &Y_2 \to c \qquad L(X)=\{ab, ac\}.\end{aligned}$$

The Parsing Machine cannot parse ab as X without backing up. More generally, we can see that the following condition must be true.

Second Condition. *For every rule of Type 1,* $\text{first}(X_q) \cap \text{first}(X_r) = \emptyset$.

To prove that this is necessary suppose there is a terminal symbol a in both $\text{first}(X_q)$ and $\text{first}(X_r)$. Thus $X_q \to^+ a\theta$ and $X_r \to^+ a\varphi$ for some terminal strings θ and φ. Now if the Parsing Machine can parse $a\theta$ as X_q and then as X_p, it cannot parse $a\varphi$ as X_r and then as X_p, since the X_p procedure calls X_q first, and this must advance past the letter a.

Another case in which the no-backup method has difficulty is reflected in the following grammatical rules:

$$\begin{aligned}&W \to X Y\\ &X \to Y \mid Z\\ &Z \to \varepsilon\\ &Y \to a \qquad L(W)=\{aa, a\}.\end{aligned}$$

The Parsing Machine cannot parse the string "a" without backing up. Thus, we find a further condition, analogous to the second.

Third Condition. *For every rule of type 1 where X_r can produce the null string,* first $(X_q) \cap$ follow $(X_p) = \emptyset$.

Here finally is another grammar which satisfies all three of the conditions discussed so far, but which still causes the no-backup method to fail:

$$
\begin{aligned}
&S \rightarrow X \dashv \\
&X \rightarrow Y | Z \\
&Y \rightarrow W T \\
&W \rightarrow V | U \\
&T \rightarrow a \\
&Z \rightarrow b \\
&V \rightarrow c \\
&U \rightarrow \varepsilon \qquad L(S) = \{ca\dashv, a\dashv, b\dashv\}.
\end{aligned}
$$

Here the Parsing Machine program will not accept the string "$b\dashv$". The procedure U cannot ever return false, so procedure W cannot return false, and neither can Y. Therefore the procedure for X will never call Z in any circumstances! This suggests adding yet another constraint.

Fourth Condition. *For every rule of type 1, X_q must not be "nonfalse"; in other words it should not correspond to a procedure which will never return false.*

We see that a nonterminal X_p is *nonfalse* if and only if its rule is

a) of Type 5 $\quad (X_p \rightarrow \varepsilon)$;
or b) of Type 3 and X_q is nonfalse;
or c) of Type 2 and X_q is nonfalse;
or d) of Type 1 and X_q or X_r is nonfalse.

This shows how we can check condition 4, by sequentially determining which of $X_1, X_2, \ldots, X_t$ are nonfalse (in that order).

8.2 Solution to the No-Backup Problem

Theorem. *If conditions 1, 2, 3 and 4 hold in a simple grammar $\mathcal{G}$, then the no-backup method works.*

(In fact, it is possible to prove that condition 1 follows from conditions 2, 3, and 4, so that condition 1 is redundant.)

The proof uses two lemmas. Let $s_1 \ldots s_n$ be the input string, and let s_k be the current symbol on the input string. We may assume by condition 1 that the nonterminal symbols have been put into an appropriate order $X_1, X_2, \ldots$ as specified in Section 8.1.

Lemma 1. Under the assumptions of the theorem, if X_p is not nonfalse and if $s_k \notin$ first (X_p), then the Parsing Machine instruction $[X_p] L_1 L_2$ will transfer to L_2.

Proof. The proof is by induction on p, considering the ordering we have defined.

Assume that the lemma is true for all X_q, when $q < p$. (In particular when $p = 1$ we are not assuming anything; proofs by induction are often convenient to state in this way, without singling out the case $p = 1$.)

First case:

$$X_p \to X_q | X_r \quad q < p, r < p.$$

Since X_p is not nonfalse, X_q and X_r cannot be nonfalse, by the definition of that property. Also since $s_h \notin \text{first}(X_p)$, we have $s_h \notin \text{first}(X_q)$ and $s_h \notin \text{first}(X_r)$. Now the PM program for procedure X_p is

loc	op-code	AT	AF	
X_p	$[X_q]$	T		(8.1)
	$[X_r]$	T	F	

Therefore by induction the machine will go to F.

Second case:

$$X_p \to X_q X_r \quad q < p.$$

Since X_p is not nonfalse, neither is X_q. And since $s_h \notin \text{first}(X_p)$, obviously $s_h \notin \text{first}(X_q)$. Now the PM program for X_p is

X_p	$[X_q]$		F	(8.2)
	$[X_r]$	T	ERROR	

so, by induction, procedure X_p will go to F.

Third case:

$$X_p \to X_q \quad q < p.$$

This case is obvious.

Fourth case:

$$X_p \to a.$$

The PM program is

X_p	a	T	F	(8.3)

and since $s_h \neq a$ this case is also obvious.

Fifth case:

$$X_p \to \varepsilon.$$

This situation cannot occur since X_p is nonfalse.

Lemma 2. Under the assumptions of the theorem, let $X_p \to^* s_h \ldots s_{k-1}$, where $h \leqq k$. (If $h = k$, this means that $X_p \to^* \varepsilon$.) Assume also that $s_k \in \text{follow}(X_p)$. Then the Parsing Machine instruction $[X_p]\, L_1 L_2$ will transfer to L_1 with h increased to k. Furthermore the actions of the machine during this time correspond to the derivation of $X_p \to^* s_h \ldots s_{k-1}$.

Proof. The proof is by induction on $k - h = m$; and, for fixed m, on p. Thus, we assume that the lemma is true for all p and m' when $m' < m$, and for all $p' < p$ when m is given.

First case:

$$X_p \to X_q | X_r, \quad q < p, \ r < p.$$

The program for procedure X_p is (8.1).

Subcase 1a:

$$X_q \to^* s_h \ldots s_{k-1}.$$

By induction the lemma is true for X_q, thus it is also true for X_p.

Subcase 1b:

$$X_r \to^* s_h \dots s_{k-1}.$$

If $h = k$ (null string) then by condition 3 $s_h = s_k \in \text{follow}(X_p)$ so $s_h \notin \text{first}(X_q)$. On the other hand if $h > k$, $s_h \notin \text{first}(X_q)$ by condition 2. Furthermore X_q is not nonfalse, by condition 4.

Thus by Lemma 1, procedure X_p will call $[X_r]$; and since $r < p$, the lemma holds by induction.

Second case:

$$X_p \to X_q X_r, \quad \text{where} \quad X_q \to^* s_h \dots s_{j-1}, \quad X_r \to^* s_j \dots s_{k-1},$$

for some j; $h \leqq j \leqq k$.

The PM program is (8.2). If $h < j$ the lemma is true for X_q and X_r (by induction, since $k - j$ is less than $k - h$ and q is less than p). If $h = j$, then X_q can produce the null string, so $r < p$; again the procedure X_p will go to T by induction.

Third case:

$$X_p \to X_q, \quad q < p.$$

The lemma is valid for X_q so it holds for X_p.

Fourth case:

$$X_p \to a.$$

We must have $s_h = a$ and $k = h + 1$. The PM program (8.3) clearly goes to T and advances h.

Fifth case:

$$X_p \to \varepsilon.$$

Here h must equal k; the PM program for X_p always goes to T without changing h.

This completes the proof of Lemma 2.

Now the theorem can be proved as follows. If $s_1 \dots s_n$ is in $L(\mathcal{G})$, the machine goes to "OK", and its actions correspond to a given derivation, by applying Lemma 2 to the program for S.

If $s_1 \dots s_n$ is not in $L(\mathcal{G})$, the machine cannot go to "OK", for each time the machine goes to "OK" its actions obviously correspond to a derivation in the grammar; this derivation must be of the entire string, since only $s_n = \dashv$.

The remaining possibility is that $s_1 \dots s_n$ is not in $L(\mathcal{G})$ but the PM program never terminates. This is impossible, since the grammar is not left-recursive; we can prove by induction on $n - h$ and (for fixed h) on p that no call of $[X_p]$ can result in an infinite loop. The latter proof is straightforward as in Lemmas 1 and 2; but it is not completely trivial, since the PM can go rather slowly as in the grammar

$$\begin{aligned} &X_1 \to \varepsilon \\ &X_2 \to X_1 X_1 \\ &X_3 \to X_2 X_2 \\ &X_4 \to X_3 X_3 \\ &X_5 \to a \\ &X_6 \to X_4 X_5 \end{aligned}$$

8.3 *Summary*

The four conditions of Section 8.1, derived for the special case of simple grammars, can now be translated back into the general situation in which all rules are of the standard form

$$X \to Y_1 | \ldots | Y_m | Z_1 \ldots Z_n \tag{8.4}$$

The theorem of Section 8.2 and the simplification procedure discussed just before Section 8.1 imply that the following four conditions are necessary and sufficient for the validity of the no-backup method, provided there are no useless nonterminal symbols:

1. The grammar contains no left-recursive nonterminals.
2. The sets first $(Y_1), \ldots,$ first (Y_m), first $(Z_1 \ldots Z_n)$ are mutually disjoint, i.e. they have no letters in common.
3. If $Z_1 \ldots Z_n \to^* \varepsilon$ then first (Y_j) contains no letters in common with follow (X).
4. $Y_1, \ldots, Y_m$ are not nonfalse.

In this case a nonterminal symbol X corresponding to the rule (8.4) is defined to be *nonfalse* if and only if either

(a) Y_j is nonfalse, for some j $(1 \leqq j \leqq m)$
or (b) $n = 0$
or (c) $n > 0$ and Z_1 is nonfalse.

As remarked earlier, condition 1 is redundant since it can be deduced from the other conditions.

It is possible to design a rather efficient method for checking these conditions: First do a "topological sort" of the nonterminal symbols, based on the left-dependency graph, and at the same time determine the sets first (X) and check conditions 2 and 4 for each X as it is emitted by the topological sorting algorithm. Then compute follow (X) for each X and test condition 3.

Exercise 2. Does the no-backup method work on the grammar (5.13)? Prove your answer, by testing the four conditions above.

9. An Overview of Top-Down and Bottom-Up Analysis

In addition to the relation $\to$ which has been defined in connection with a context-free grammar $\mathcal{G}$ in Section 5, we can also define the restricted relation

$$\alpha A \omega \overset{L}{\to} \alpha \Theta \omega \tag{9.1}$$

if $A \to \Theta$ is a production and if $\alpha \in T^*$. This means A is the leftmost nonterminal in $\alpha A \omega$. Similarly we define

$$\alpha A \omega \overset{R}{\to} \alpha \Theta \omega \tag{9.2}$$

if $\omega \in T^*$. These relations can be extended as before to $\overset{L}{\to}{}^*$, $\overset{R}{\to}{}^+$, $\overset{L}{\to}{}^*$, $\overset{R}{\to}{}^+$; the relation

$$\varphi \overset{L}{\to}{}^+ \psi$$

may be read, "φ left-produces ψ".

There are in general many sequences of strings $\sigma_1, \sigma_2, \ldots, \sigma_m$ such that

$$S \to \sigma_1 \to \cdots \to \sigma_m \to \tau$$

is a derivation of the terminal string τ in a contextfree grammar; whenever σ_j contains at least two nonterminal symbols, we have a choice as to which nonterminal to replace first. The importance of the $\overset{L}{\to}$ and $\overset{R}{\to}$ relations is that, for every diagram such as (2.2), there is exactly one corresponding $\overset{L}{\to}$ derivation, and exactly one corresponding $\overset{R}{\to}$ derivation. Thus, using the example (2.2) and the grammar (2.1), we have.

$$\begin{aligned} S &\overset{L}{\to} B\dashv \overset{L}{\to} (B)\dashv \overset{L}{\to} (R)\dashv \\ &\overset{L}{\to} (E=E)\dashv \overset{L}{\to} (a=E)\dashv \\ &\overset{L}{\to} (a=(E+E))\dashv \\ &\overset{L}{\to} (a=(b+E))\dashv \\ &\overset{L}{\to} (a=(b+a))\dashv \end{aligned} \tag{9.3}$$

and

$$\begin{aligned} S &\overset{R}{\to} B\dashv \overset{R}{\to} (B)\dashv \overset{R}{\to} (R)\dashv \\ &\overset{R}{\to} (E=E)\dashv \overset{R}{\to} (E=(E+E))\dashv \\ &\overset{R}{\to} (E=(E+a))\dashv \\ &\overset{R}{\to} (E=(b+a))\dashv \\ &\overset{R}{\to} (a=(b+a))\dashv \end{aligned} \tag{9.4}$$

as the corresponding left and right derivations. A context-free grammar $\mathscr{G}$ is *unambiguous* if each string in $L(\mathscr{G})$ has exactly one left derivation (or equivalently, exactly one right derivation).

The general problem of syntactic analysis is to start with a string τ of terminal symbols, e.g., $(a=(b+a))\dashv$, and to find (when possible) a sequence of productions such that $S \to^* \tau$.

The *bottom-up method* (proceeding from left to right) attacks this problem by first "reducing" the above string to

$$(E=(b+a))\dashv$$

then reducing this to

$$(E=(E+a))\dashv$$

and then

$$(E=(E+E))\dashv$$

and

$$(E=E)\dashv$$

etc. (A *reduction* is the opposite of a production.) The leftmost possible reduction is applied at each step. This process continues until we reduce everything to S, or show that this would be impossible. Note that the sequence of intermediate steps is exactly the reverse of the right production sequence (9.4). In general, *bottom-up analysis (from left to right) proceeds by right reductions,* i.e. by reversing the right production sequence.

The *top-down method* (proceeding from left to right) attacks this problem somewhat differently. It starts with S, and attempts to reach the given terminal string τ by a sequence of left productions, as in (9.3). At each step we must decide which production to apply to the leftmost nonterminal symbol. In general, *top-down analysis (from left to right) proceeds by left productions.*

Similarly we can describe top-down and bottom-up methods which go from right to left, by interchanging the rôles of left and right in the above discussion. (Currently some extended methods which are more symmetrical between left and right are being explored by several people.)

Various "backup" procedures can be given for reconsidering some alternatives of the derivation sequence that later prove to be incorrect. Principal interest, however, centers on the cases where the syntactic analysis proceeds *without* backing up: then each step of the derivation sequence is known to be the only possible step. Such procedures are usually called *deterministic* analysis methods.

In a previous paper [7], the author has described the general conditions under which strings of a grammar can be analyzed deterministically, bottom-up, from left to right, looking k characters ahead of where the next reduction step will be made. Such grammars are called $LR(k)$ grammars. The analogous property for top-down analysis, first suggested by Lewis and Stearns [8], will be called here the "$LL(k)$" property; it means a top-down analysis is to be performed from left to right, looking k characters ahead of the terminal symbols which have already been matched. Formal definitions of these concepts appear in the next section; for the present, let us quickly review the significance of four properties a context-free grammar may have;

$LL(k)$: scan from the left, using left productions

$LR(k)$: scan from the left, using right reductions

$RL(k)$: scan from the right, using left reductions

$RR(k)$: scan from the right, using right productions.

(In each case, k represents the number of symbols of "lookahead," used to decide what production or reduction to perform next. A "right reduction" is the opposite of a "right production" (9.2).) $LL(k)$ and $RR(k)$ correspond to top-down analysis.

10. Definitions of $LL(k)$ and $LR(k)$

Let us now introduce some useful notations. If k is a nonnegative integer and α is a string, we define

$$k:\alpha = \begin{cases} \textbf{if } |\alpha| \geqq k \textbf{ then} \text{ the first } k \text{ characters of } \alpha \\ \textbf{else } \alpha \end{cases} \tag{10.1}$$

$$\alpha:k = \begin{cases} \textbf{if } |\alpha| \geqq k \textbf{ then} \text{ the last } k \text{ characters of } \alpha \\ \textbf{else } \alpha \end{cases} \tag{10.2}$$

Furthermore if A is a set of strings, we write

$$k:A = \{k:\alpha \mid \alpha \in A\}, \tag{10.3}$$

$$A:k = \{\alpha:k \mid \alpha \in A\}. \tag{10.4}$$

Note that $\text{first}(A) = (1:L(A)) \cap T = (1:L(A)) - \{\varepsilon\}$; also if $k:\beta = k:\gamma$ then $k:(\alpha\beta) = k:(\alpha\gamma)$.

The informal discussion of Section 9 can now be formulated precisely in terms of this notation.

Definition. A context-free grammar is $LR(k)$ if the following condition holds for all α_1, α_1', α_2, and α_2' in $(N \cup T)^*$, all α_3 and α_3' in T^*, and all A, A' in N:

$$S \overset{R}{\rightarrow}{}^* \alpha_1 A \alpha_3 \overset{R}{\rightarrow} \alpha_1\alpha_2\alpha_3$$

and

$$S \overset{R}{\rightarrow}{}^* \alpha_1' A' \alpha_3' \overset{R}{\rightarrow} \alpha_1'\alpha_2'\alpha_3'$$

and

$$(|\alpha_1\alpha_2| + k):\alpha_1\alpha_2\alpha_3 = (|\alpha_1\alpha_2| + k):\alpha_1'\alpha_2'\alpha_3'$$

implies that

$$\alpha_1 = \alpha_1', \quad A = A', \quad \text{and} \quad \alpha_2 = \alpha_2'.$$

Definition. A context-free grammar is $LL(k)$ if the following condition holds for all α_1, α_4, α_4' in T^* and all α_2, α_3, α_2', in $(N \cap T)^*$:

$$S \overset{L}{\rightarrow}{}^* \alpha_1 A \alpha_3 \overset{L}{\rightarrow} \alpha_1\alpha_2\alpha_3 \overset{L}{\rightarrow}{}^* \alpha_1\alpha_4$$

and

$$S \overset{L}{\rightarrow}{}^* \alpha_1 A \alpha_3 \overset{L}{\rightarrow} \alpha_1\alpha_2'\alpha_3 \overset{L}{\rightarrow}{}^* \alpha_1\alpha_4'$$

and

$$k:\alpha_4 = k:\alpha_4'$$

implies that

$$\alpha_2 = \alpha_2'.$$

Let us now define a generalization of the "follow" function that allows us to derive further information about the $LL(k)$ property: If $\beta \in T^*$, $A \in N$, and k is a nonnegative integer, let

$$F_k(A, \beta) = k:\{\theta \mid \theta \in T^* \text{ and } S \rightarrow^* \beta A \theta\}. \qquad (10.5)$$

Thus in our earlier notation,

$$\text{follow}(A) = \bigcup_{\beta \in T^*} F_1(A, \beta)$$

if $A \neq S$ and if we regard $\dashv$ as an element of T.

Theorem (Lewis and Stearns). *If $A \rightarrow \alpha_1$ and $A \rightarrow \alpha_2$ are two distinct productions of an $LL(k)$ grammar, and if β is in T^*, then*

$$[k:(L(\alpha_1)\,\theta_1)] \cap [k:(L(\alpha_2)\,\theta_2)] = \emptyset \qquad (10.6)$$

for all θ_1, $\theta_2 \in F_k(A, \beta)$.

(In fact this condition can be proved equivalent to the $LL(k)$ property; see [8]. The condition leads to a fairly simple algorithm to test whether a context-free grammar is $LL(k)$ for a given k.)

Proof. Suppose there are strings ω, θ_1, θ_2, such that

$$\omega \in k: (L(\alpha_1)\,\theta_1)$$
$$\omega \in k: (L(\alpha_2)\,\theta_2)$$
$$\theta_1, \theta_2 \in F_k(A, \beta).$$

We will show that this implies $\alpha_1 = \alpha_2$, which contradicts the hypothesis that $A \to \alpha_1$ and $A \to \alpha_2$ are distinct production rules.

The proof is mostly a matter of translating between notations. By hypothesis, there are θ_1' and θ_2' in T^* such that

$$S \to^* \beta A \theta_1', \quad S \to^* \beta A \theta_2', \quad k:\theta_1' = \theta_1, \quad k:\theta_2' = \theta_2.$$

Hence by considering the corresponding left-derivations, there are strings γ_1, γ_2 in $(N \cup T)^*$ such that

$$S \overset{L}{\to}{}^* \beta A \gamma_1, \quad S \overset{L}{\to}{}^* \beta A \gamma_2, \quad \gamma_1 \to^* \theta_1', \quad \gamma_2 \to^* \theta_2'.$$

There are also τ_1, τ_2 in $L(\alpha_1)$, $L(\alpha_2)$ respectively such that

$$k:\tau_1\theta_1' = k:\tau_1\theta_1 = \omega = k:\tau_2\theta_2 = k:\tau_2\theta_2'.$$

Therefore $\gamma_1 = \gamma_2$ (see Exercise 4).

Now we have

$$S \overset{L}{\to}{}^* \beta A \gamma_1 \overset{L}{\to} \beta \alpha_1 \gamma_1 \overset{L}{\to}{}^* \beta \tau_1 \theta_1'$$
$$S \overset{L}{\to}{}^* \beta A \gamma_2 \overset{L}{\to} \beta \alpha_2 \gamma_2 \overset{L}{\to}{}^* \beta \tau_2 \theta_2'$$

so $\alpha_1 = \alpha_2$ by the definition of $LL(k)$. Q.E.D.

Exercise 3. Show that an $LL(k)$ grammar with no useless nonterminals has no left-recursive nonterminals.

Exercise 4. Show that every $LL(k)$ grammar satisfies the following condition: For all α_1, α_4, α_4' in T^* and all α_3, α_3' in $(N \cup T)^*$, if

$$S \overset{L}{\to}{}^* \alpha_1 A \alpha_3' \overset{L}{\to}{}^* \alpha_1 \alpha_4$$

and

$$S \overset{L}{\to}{}^* \alpha_1 A \alpha_3' \overset{L}{\to}{}^* \alpha_1 \alpha_4'$$

and

$$k:\alpha_4 = k:\alpha_4'$$

then $\alpha_3 = \alpha_3'$.

11. *LL*(1) Grammars

The special case $k = 1$ in the preceding discussion of $LL(k)$ seems to be quite a bit simpler than the case of higher values of k, and it is directly related to our previous discussion of the "no-backup method," so we shall now study it in detail.

Suppose G is a grammar with no useless nonterminals; let us test the condition of the last theorem for the case $k = 1$. Let A be a nonterminal symbol whose

rule is

$$A \to \alpha_1 | \alpha_2 | \ldots | \alpha_m, \quad m \geq 1. \tag{11.1}$$

If $p \neq q$, and if $a \in \text{first}(\alpha_p)$, $b \in \text{first}(\alpha_q)$, condition (10.6) says that $a \neq b$. Furthermore we cannot have both $\alpha_p \to^* \varepsilon$ and $\alpha_q \to^* \varepsilon$, since (10.6) would be violated if we take $\theta_1 = \theta_2$. And if $\alpha_p \to^* \varepsilon$ and $b \in \text{first}(\alpha_q)$, condition (10.6) says that $b \notin F_1(A, \beta)$ for any β, i.e. $b \notin \text{follow}(A)$.

Thus we have three conditions which are necessary for each rule (11.1) of an $LL(1)$ grammar:

1. $\text{first}(\alpha_1), \ldots, \text{first}(\alpha_m)$ are mutually disjoint, i.e. contain no common elements.

2. At most one of $\alpha_1, \ldots, \alpha_m$ can produce a null string.

3. If $\alpha_p \to^* \varepsilon$, then $\text{first}(\alpha_q)$ has no elements in common with $\text{follow}(A)$, for $1 \leq q \leq m$, $q \neq p$.

(Note: Conditions 1 and 2 can be combined by saying that

$$(1 : L(\alpha_p)) \cap (1 : L(\alpha_q)) = \emptyset \quad \text{when} \quad p \neq q.)$$

Now these three conditions are obviously *sufficient* to show that $\mathscr{G}$ is $LL(1)$: For if we are to replace A by one of $\alpha_1, \ldots, \alpha_m$, the choice of which α_j to use is uniquely determined by examining the first character of the terminal string which ultimately is to be produced by A.

Note that grammars satisfying the no-backup condition stated at the end of Section 8 also satisfy the three conditions above, so they are $LL(1)$ grammars. But a grammar such as

$$\begin{aligned} &B \to A\,b \,|\, C\,d \\ &A \to a\,A \,|\, \varepsilon \\ &C \to c\,C \,|\, \varepsilon \\ &S \to B\dashv \end{aligned} \tag{11.2}$$

which is $LL(1)$ does not satisfy the non-backup condition. Both A and C are nonfalse, so the Parsing Machine will not be able to work with this grammar, regardless of whether we write

$$B \to A\,b \,|\, C\,d \quad \text{or} \quad B \to C\,d \,|\, A\,b$$

before converting to standard form.

However, it is possible to put any $LL(1)$ grammar into a fairly simple standard form, which can be treated by the Parsing Machine.

Let us say an $LL(k)$ *language* is a set of strings which can be defined by an $LL(k)$ *grammar.*

Theorem. *Any $LL(1)$ language can be given a grammar for which all rules have one of the two forms*

Type 1. $A \to a_1\alpha_1 | \ldots | a_m\alpha_m$

Type 2. $A \to a_1\alpha_1 | \ldots | a_m\alpha_m | \varepsilon$

Here $a_1, \ldots a_m$ are distinct terminal characters; and in rules of Type 2, no a_j is in follow(A).

(Conversely, such a grammar is obviously $LL(1)$, and it obviously also satisfies the conditions of the no-backup method. So the $LL(1)$ languages are precisely those analyzable by some no-backup PM program.)

Proof. Given a grammar $\mathcal{G}$ satisfying conditions 1, 2, 3 above and with no useless nonterminals, we can prove (see for example exercise 3) that $\mathcal{G}$ has no left-recursive nonterminals. Thus we can order the nonterminal symbols A_1, $A_2, \ldots, A_t$ where $A_p l^+ A_q$ only if $q < p$.

It follows that the rule for A_1 must be of either Type 1 or Type 2 as in the theorem.

Suppose that the rules for nonterminals A_q have the form of Type 1 or Type 2, for all $q < p$. Suppose further that, if

$$A_q \rightarrow a_{q1}\,\alpha_{q1} | \ldots | a_{q m_q} \alpha_{q m_q} | \varepsilon \quad \text{and} \quad q < p, \tag{11.3}$$

is a rule of Type 2, we also have added the additional rule

$$A'_q \rightarrow a_{q1}\,\alpha_{q1} | \ldots | a_{q m_q} \alpha_{q m_q} \tag{11.4}$$

to the grammar, where A'_q is a new nonterminal symbol. We will show how to change the rule of A_p so that it has the form of either Type 1 or Type 2.

Let the rule for A_p be

$$A_p \rightarrow \alpha_1 | \alpha_2 | \ldots | \alpha_n.$$

If α_j begins with a nonterminal symbol, so that α_j has the form $A_q \beta$, we must have $q < p$. If the rule for A_q is of Type 2, replace α_j by two alternatives $A'_q \beta | \beta$, where A'_q is given by (11.4). This leaves us with an $LL(1)$ grammar: for first$(A_q) \cap$ first$(\beta) \subseteq$ first$(A'_q) \cap$ follow$(A_q) = \emptyset$; and first$(A'_q) \cup$ first$(\beta) =$ first(α_j) has no letters in common with first(α_i) for $i \neq j$. Furthermore if $\beta \rightarrow^* \varepsilon$, we must verify that first(A'_q) has no letters in common with follow(A_p), and this is true because follow$(A_p) \subseteq$ follow(A_q) when $\beta \rightarrow^* \varepsilon$.

By repeating this process, we may assume that A_p has a rule

$$A_p \rightarrow \alpha_1 | \alpha_2 | \ldots | \alpha_n$$

where none of the α_j begins with a nonterminal whose rule is of Type 2. Now if α_j begins with a nonterminal A_q, we may write $\alpha_j = A_q \beta$, where $q < p$, and the rule for A_q is

$$A_q \rightarrow a_{q1} \alpha_{q1} | \ldots | a_{q m_q} \alpha_{q m_q}$$

Therefore we may replace the alternative α_j by

$$a_{q1} \alpha_{q1} \beta | \ldots | a_{q m_q} \alpha_{q m_q} \beta$$

and we still clearly have an $LL(1)$ grammar. Thus the rule for A_p can ultimately be reduced to either Type 1 or Type 2, and the proof is complete. (The auxiliary rules (11.4) are not needed after the construction has terminated.)

As an example, the grammar (11.2) above would be changed to

$$\begin{aligned} &A \to aA \mid \varepsilon \\ &C \to cC \mid \varepsilon \\ &B \to aAb \mid b \mid cCd \mid d \\ &S \to aAb\dashv \mid b\dashv \mid cCd\dashv \mid d\dashv \end{aligned} \tag{11.5}$$

(Here B has become useless.)

Grammars in while all rules are of Type 1 are obviously $LL(1)$. They have been called "s-grammars" by Korenjak and Hopcroft [9], who proved (among other things) that the *equivalence problem* is solvable for s-languages. In other words, if $\mathscr{G}_1$ and $\mathscr{G}_2$ are s-grammars, we can decide if $L(\mathscr{G}_1) = L(\mathscr{G}_2)$.

The grammar (11.5) can be transformed into an s-grammar for the same language if we introduce new nonterminal symbols $[Ab]$ and $[Cd]$ standing respectively for $L(Ab)$ and $L(Cd)$. The s-grammar is

$$\begin{aligned} [Ab] &\to a[Ab] \mid b \\ [Cd] &\to c[Cd] \mid d \\ S &\to a[Ab]\dashv \mid b\dashv \mid c[Cd]\dashv \mid d\dashv \end{aligned}$$

While preparing these lectures, the author found that many other $LL(1)$ grammars are amenable to similar transformations; so it seemed reasonable to make the conjecture that, whenever L is an $LL(1)$ language, then $L\dashv$ is an s-language. If this conjecture were true, it would provide a solution to the equivalence problem for $LL(1)$ languages.

One grammar which seemed to provide a counterexample to this conjecture was the following:

$$\begin{aligned} &P \to +PP \mid xA \\ &A \to aA \mid \varepsilon \\ &S \to P\dashv \end{aligned} \tag{11.6}$$

This represents Polish prefix notation, with the binary operator $+$ and the variables x, xa, xaa, ... The author made several fruitless attempts to find an s-grammar for this language, before finally hitting on the following trick: We can write $P = P'A$, where P' represents those strings of P not ending in a; and we can therefore give the following s-grammar for the language (11.6):

$$\begin{aligned} P' &\to +P'[AP'] \mid x \\ [AP'] &\to a[AP'] \mid +P'[AP'] \mid x \\ S &\to P'[A\dashv] \\ [A\dashv] &\to a[A\dashv] \mid \dashv \end{aligned}$$

This example gave added weight to the conjecture.

However, R. Kürki-Suonio found a simple counterexample [10], which shows that ε cannot in general be removed from an $LL(1)$ grammar followed by $\dashv$. Thus, *the null string plays an important role in top-down analysis.*

Exercise 5 (suggested by Karel Čulik II). Show that the following grammar is $LL(2)$. Is there an $LL(1)$ grammar for the same language?

$$S \to aaSbb \mid a \mid \varepsilon$$

12. Concluding Remarks

It is natural to inquire which is better, top-down or bottom-up analysis? It is difficult to give a complete answer to this question, but some comments can be made.

First, every $LL(k)$ grammar is an $LR(k)$ grammar. (This can be proved using a rather involved but intuitively clear argument. A careful and clear proof of this fact would make a suitable master's thesis and would be an instructive project.)

On the other hand, the "Boolean expression" grammar (2.1) is $LR(0)$ but not $LL(k)$ for any k: The string $(((((((((((a=b)))))))))))$ needs to be parsed quite differently from the string

$$(((((((((((a+b)+b)+b)+b)+b)+b)+b)+b)+b)+b)+b)=b$$

which begins with the same 12 symbols. It can in fact be shown that this is not even an $LL(k)$ *language*, i.e. there is no equivalent $LL(k)$ grammar. Thus, bottom-up analysis can deterministically parse more general languages than top-down analysis can.

On the other hand when we are fortunate enough to have an $LL(1)$ grammar, we have more flexibility in applying semantic rules, since we know what production is being used *before* we actually process its components. This foreknowledge can be extremely important in practice. The bottom-up procedure only discovers what is present after it has scanned the text. A theoretical model which demonstrates this advantage of top-down analysis has been discussed at length by Lewis and Stearns [8].

A simple model for bottom-up analysis which might be considered the bottom-up analogue of the *no-backup* Parsing Machine is the technique of "simple precedence grammars" developed by Wirth and Weber [11]. The bottom-up analog of the *partial backup* method is the original method of Irons [12], which has not yet been given a theoretical treatment.

In conclusion, here are some research problems which are (perhaps) listed in increasing order of difficulty:

1. Is every $LL(k)$ language, $k \geqq 1$, an $LL(1)$ language? (The corresponding statement is true for $LR(k)$ and $LR(1)$ languages.)

2. Is the equivalence problem solvable for $LL(1)$ languages? (The work of Korenjak and Hopcroft [9] makes this likely.)

3. Can the method of Irons [12] be analyzed theoretically in a fashion analogous to the partial-backup method discussed above?

4. What class of languages can the Parsing Machine accept? (See Exercise 1.)

5. Is the equivalence problem decidable for $LR(1)$ languages? (This seems to be the most important unsolved problem at the present time; it is also desirable to answer the question for very special cases, such as Wirth's precedence grammars [11].)

Postscript, Added May 1971. Research problems 1 and 2 have now been resolved. R. Kürki-Suonio proved that the $LL(k+1)$ grammar

$$S \to aSA \mid \varepsilon$$
$$A \to a^k bS \mid c$$

defines a non-$LL(k)$ language [10]. D. J. Rosenkrantz and R. E. Stearns showed that the equivalence problem for $LL(k)$ grammars is solvable [13]. Both of these papers establish a number of other important facts about $LL(k)$ grammars and languages.

Substantial progress has also been made on the theory related to research problem 3. The author (and others) believed in 1967 that Irons's original algorithm was "bottom up" as stated above, but it has subsequently become clear that the true bottom-up methods are those of Cocke, Younger, Kasami *et al.* (see [14]). The method of Irons is neither top-down nor bottom-up, and it has become known as "left-corner parsing"; the corresponding grammars, called $LC(k)$, have been studied by Rosenkrantz and Lewis [15], who showed that all $LC(k)$ languages are $LL(k)$ languages and conversely.

Appendix. Answers to the Exercises

Exercise 1. $\{a^n b^n c^n \dashv | n=0, 1, 2, \ldots\}$. Hence the PM can accept languages which are not context-free.

Exercise 2. Condition 1 has been verified in the text. Condition 2 needs only be verified for the rule $P \to a|b|(E)$, where first$(a)=a$, first$(b)=b$, first$((E))=($ are obviously disjoint. (In the other rules, $m=0$ or $n=0$.) Condition 3 is true because follow$(L')=\{), \dashv\}$ has no letters in common with first$(+LL')=\{+\}$; and follow$(P')=\{), +, \dashv\}$ has no letters in common with first$(PP')=\{a, b, (\}$. Finally, the nonfalse nonterminals are L' and P', so condition 4 holds.

Exercise 3. If $A \to^+ A$ the grammar is ambiguous and therefore not $LL(k)$. If $A \to^+ A\alpha$ for $\alpha \neq \varepsilon$, and if $A \to^+ \beta$, where β and α are in T^*, then $A \to^+ \beta\alpha^n$ for arbitrarily large n. So examining the first k characters of $\beta\alpha^n$ will not tell us how many times to do the left production sequence corresponding to $A \to^+ A\alpha$.

Exercise 4. Consider the initial steps of the derivation; if

$$S \overset{L}{\to}{}^* \beta_1 B \beta_3 \overset{L}{\to} \beta_1 \beta_2 \beta_3 \overset{L}{\to}{}^* \alpha_1 A \alpha_3$$
$$S \overset{L}{\to}{}^* \beta_1 B \beta_3 \overset{L}{\to} \beta_1 \beta_2' \beta_3 \overset{L}{\to}{}^* \alpha_1 A \alpha_3'$$

then $\alpha_1 = \beta_1 \gamma$ for some γ in T^*. Since $k{:}\gamma\alpha_4 = k{:}\gamma\alpha_4'$, the definition of $LL(k)$ implies that $\beta_2 = \beta_2'$. Thus if $S \overset{L}{\to}{}^* \alpha_1 A \alpha_3$ in m steps and if $S \overset{L}{\to}{}^* \alpha_1 A \alpha_3'$ in $n \geqq m$ steps, the first m steps must be identical. It follows that $\alpha_1 A \alpha_3 \overset{L}{\to}{}^* \alpha_1 A \alpha_3'$ in $n-m$ steps. By exercise 3, $m=n$.

Exercise 5. The strings are $\{a^n b^{2[n/2]} | n \geqq 0\}$. By looking 2 characters ahead, we apply $S \to aaSbb$ if there are two a's, $S \to a$ if one a but not two, $S \to \varepsilon$ otherwise.

An equivalent $LL(1)$ grammar is

$$S \to aT \mid \varepsilon$$
$$T \to aUb \mid \varepsilon$$
$$U \to aTb \mid b$$

References

1. Barnett, M. P., Futrelle, R. P.: Syntactic analysis by digital computer. Comm. ACM **5**, 515–526 (1962).
2. Brooker, R. A., Morris, D.: A description of Mercury Autocode in terms of a phrase structure language. Ann. Review Auto. Programming **2**, 29–65 (1961). See also S. Rosen, Comm. ACM **7**, 403–414 (1964).
3. Conway, M. E.: Design of a separable transition-diagram compiler. Comm. ACM **6**, 396–408 (1963).
4. Schorre, D. V.: META II, a syntax-oriented compiler writing language. Proc. ACM National Conf. **19**, D1.3.1–D1.3.11 (1964).
5. Knuth, D. E.: The Art of Computer Programming (to be published in seven volumes).
6. Floyd, R. W.: The syntax of programming languages—a survey. IEEE Transactions EC-**13**, 346–353 (1964).
7. Knuth, D. E.: On the translation of languages from left to right. Information and Control **8**, 607–639 (1965).
8. Lewis II, P. M., Stearns, R. E.: Syntax-directed transduction. J. ACM **15**, 464–488 (1968).
9. Korenjak, A. J., Hopcroft, J. E.: Simple deterministic languages. Proc. IEEE Symp. Switching and Automata Theory **7**, 36–46 (1966).
10. Kürki-Suoni, R.: Notes on top-down languages. BIT **9**, 225–238 (1969).
11. Wirth, N., Weber, H.: Euler, a generalization of Algol, and its formal definition. Comm. ACM **9**, 13–23, 25, 89–99, 878 (1966).
12. Irons, E. T.: A syntax-directed compiler for Algol 60. Comm. ACM **4**, 51–55 (1961).
13. Rosenkrantz, D. J., Stearns, R. E.: Properties of deterministic top-down grammars. Information and Control **17**, 226–256 (1970).
14. Earley, J.: An efficient context-free parsing algorithm. Comm. ACM **13**, 94–102 (1970).
15. Rosenkrantz, D. J., Lewis II, P. M.: Deterministic left corner parsing. Proc. IEEE Symp. Switching and Automata Theory **11**, 139–152 (1970).

Prof. Donald E. Knuth
Computer Science Department
Stanford University
Stanford, California 94305
U.S.A.