presented in Figure 1, and to thank his colleagues, Richard Conway, William Maxwell, and Robert Wagner for their helpful discussions of the material.

RECEIVED APRIL, 1969; REVISED AUGUST, 1969

REFERENCES

1. ALBERGA, CYRIL, N. String similarity and misspellings. *Comm. ACM 10*, 5 (May 1967), 302–313.
2. BLAIR, CHARLES R. A program for correcting spelling errors. *Information and Control 3* (Mar. 1960), 60–67.
3. CONWAY, R. W., AND MAXWELL, W. L. CORC—the Cornell computing language. *Comm. ACM 6* (June 1963), 317–321.
4. CONWAY, R. W., AND MAXWELL, W. L. CUPL—an approach to introductory computing instruction. Tech. Rep. No. 68-4, Dept. of Computer Science, Cornell U., Ithaca, N. Y.
5. CONWAY, R. W., AND WORLEY, W. S. The Cornell—HASP system for the 360/65. Tech. Rep. No. 68-A1, Office of Computer Services, Cornell U., Ithaca, N. Y.
6. DAMERAU, F. A technique for computer detection and correction of spelling errors. *Comm. ACM 7*, 3 (Mar. 1964), 171–176.
7. DAVIDSON, L. Retrieval of misspelled names in an airlines passenger reservation system. *Comm. ACM 5*, 3 (Mar. 1962), 169–171.
8. FREEMAN, D. N. Error correction in CORC: The Cornell Computing Language. Ph.D. Th., Cornell U., Ithaca, N. Y., Sept. 1963.
9. GLANTZ, H. T. On the recognition of information with a digital computer. *J. ACM 4*, 2 (Apr. 1957), 178–188.
10. HAMMING, R. W. One Man's View of Computer Science. *J. ACM 16*, 1 (Jan. 1969), 3–12.
11. JACKSON, M. Mnemonics, *Datamation 13* (Apr. 1967), 26–29.

An Efficient Context-Free Parsing Algorithm

JAY EARLEY
University of California, Berkeley, California*

A parsing algorithm which seems to be the most efficient general context-free algorithm known is described. It is similar to both Knuth's LR(k) algorithm and the familiar top-down algorithm. It has a time bound proportional to n^3 (where n is the length of the string being parsed) in general; it has an n^2 bound for unambiguous grammars; and it runs in linear time on a large class of grammars, which seems to include most practical context-free programming language grammars. In an empirical comparison it appears to be superior to the top-down and bottom-up algorithms studied by Griffiths and Petrick.

KEY WORDS AND PHRASES: syntax analysis, parsing, context-free grammar, compilers, computational complexity
CR CATEGORIES: 4.12, 5.22, 5.23

1. Introduction

Context-free grammars (BNF grammars) have been used extensively for describing the syntax of programming languages and natural languages. Parsing algorithms for context-free grammars consequently play a large role in the implementation of compilers and interpreters for programming languages and of programs which "understand" or translate natural languages.

Numerous parsing algorithms have been developed. Some are general, in the sense that they can handle all context-free grammars, while others can handle only subclasses of grammars. The latter, restricted algorithms tend to be much more efficient. The algorithm described here seems to be the most efficient of the general algorithms, and also it can handle a larger class of grammars in linear time than most of the restricted algorithms. We back up these claims of efficiency with both a formal investigation and an empirical comparison.

This paper is based on the author's 1968 report [1] where many of the points studied here appear in much greater detail. In Section 2 the terminology used in this paper is defined. In Section 3 the algorithm is described informally and in Section 4 it is described precisely. Section 5 is a study of the formal efficiency properties of the algorithm and may be skipped by those not interested in this aspect. Section 6 has the empirical comparison and in Section 7 the practical use of the algorithm is discussed.

2. Terminology

A *language* is a set of strings over a finite set of symbols. We call these *terminal* symbols and represent them by lowercase letters: a, b, c. We use a *context-free grammar* as a formal device for specifying which strings are in the set. This grammar uses another set of symbols, the *nonterminals*, which we can think of as syntactic classes. We use capitals for nonterminals: A, B, C. Strings of either terminals or nonterminals are represented by Greek letters: α, β, γ. The empty string is λ. α^k represents

$$\overbrace{\alpha \cdots \alpha}^{k \text{ times}}.$$

$|\alpha|$ is the number of symbols in α. There is a finite set of productions or rewriting rules of the form $A \rightarrow \alpha$. The nonterminal which stands for "sentence" is called the root R

* Computer Science Department. This work was partially supported by the Office of Naval Research under Contract No. NONR3656(23) with the Computer Center, University of California, Berkeley, and by the Advanced Research Projects Agency of the Office of the Secretary of Defense (F44620-67-C-0058), monitored by the Air Force Office of Scientific Research.

of the grammar. The productions with a particular nonterminal D on their left sides are called the *alternatives* of D. Hereafter we use grammar to mean context-free grammar.

We will work with this example grammar of simple arithmetic expressions, grammar AE:

$$E \rightarrow T$$

$$E \rightarrow E + T$$

$$T \rightarrow P$$

$$T \rightarrow T * P$$

$$P \rightarrow a$$

The terminal symbols are {a,+,*}, the nonterminals are {E, T, P}, and the root is E.

Most of the rest of the definitions are understood to be with respect to a particular grammar G. We write $\alpha \Rightarrow \beta$ if $\exists \gamma, \delta, \eta$, A such that $\alpha = \gamma A \delta$ and $\beta = \gamma\eta\delta$ and $A \rightarrow \eta$ is a production. We write $\alpha \overset{*}{\Rightarrow} \beta$ (β is *derived* from α) if $\exists$ strings $\alpha_0, \alpha_1, \cdots, \alpha_m$ $(m \geq 0)$ such that

$$\alpha = \alpha_0 \Rightarrow \alpha_1 \Rightarrow \cdots \Rightarrow \alpha_m = \beta.$$

The sequence $\alpha_0, \cdots, \alpha_m$ is called a derivation (of β from α).

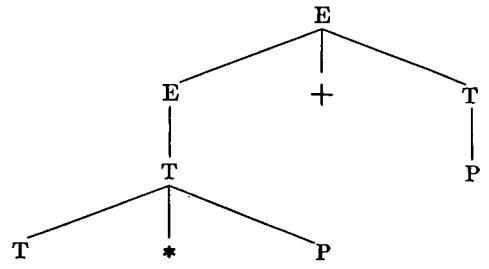
A *sentential form* is a string α such that the root $R \overset{*}{\Rightarrow} \alpha$. A *sentence* is a sentential form consisting entirely of terminal symbols. The *language defined by a grammar* $L(G)$ is the set of its sentences. We may represent any sentential form in at least one way as a *derivation tree* (or *parse tree*) reflecting the steps made in deriving it (though not the order of the steps). For example, in grammar AE, either derivation

$$E \Rightarrow E + T \Rightarrow T + T \Rightarrow T + P \Rightarrow T * P + P$$

or

$$E \Rightarrow E + T \Rightarrow E + P \Rightarrow T + P \Rightarrow T * P + P$$

is represented by



The *degree of ambiguity* of a sentence is the number of its distinct derivation trees. A sentence is *unambiguous* if it has degree 1 of ambiguity. A grammar is *unambiguous* if each of its sentences is unambiguous. A grammar has *bounded ambiguity* if there is a bound b on the degree of ambiguity of any sentence of the grammar. A grammar is *reduced* if every nonterminal appears in some derivation of some sentence.

A *recognizer* is an algorithm which takes as input a string and either *accepts* or *rejects* it depending on whether or not the string is a sentence of the grammar. A *parser* is a recognizer which also outputs the set of all legal derivation trees for the string.

3. Informal Explanation

The following is an informal description of the algorithm as a recognizer: It scans an input string $X_1 \cdots X_n$ from left to right looking ahead some fixed number k of symbols. As each symbol X_i is scanned, a set of states S_i is constructed which represents the condition of the recognition process at that point in the scan. Each state in the set represents (1) a production such that we are currently scanning a portion of the input string which is derived from its right side, (2) a point in that production which shows how much of the production's right side we have recognized so far, (3) a pointer back to the position in the input string at which we began to look for that instance of the production, and (4) a k-symbol string which is a syntactically allowed successor to that instance of the production. This quadruple is represented here as a production, with a dot in it, followed by an integer and a string.

For example, if we are recognizing a * a with respect to grammar AE and we have scanned the first a, we would be in the state set S_1 consisting of the following states (excluding the k-symbol strings):

$$P \rightarrow a. \qquad 0$$

$$T \rightarrow P. \qquad 0$$

$$T \rightarrow T.*P \qquad 0$$

$$E \rightarrow T. \qquad 0$$

$$E \rightarrow E.+T \qquad 0$$

Each state represents a possible parse for the beginning of the string, given that only the a· has been seen. All the states have 0 as a pointer, since all the productions represented must have begun at the beginning of the string.

There will be one such state set for each position in the string. To aid in recognition, we place $k + 1$ right terminators "$\dashv$" (a symbol which does not appear elsewhere in the grammar) at the right end of the input string.

To begin the algorithm, we put the single state

$$\phi \rightarrow .R \dashv \quad \dashv^k \quad 0$$

into state set S_0, where R is the root of the grammar and where ϕ is a new nonterminal.

In general, we operate on a state set S_i as follows: we process the states in the set in order, performing one of three operations on each one depending on the form of the state. These operations may add more states to S_i and may also put states in a new state set S_{i+1}. We describe these three operations by example.

In grammar AE, with $k = 1$, S_0 starts as the single state

$$\phi \rightarrow .E \dashv \quad \dashv \quad 0 \qquad (1)$$

The *predictor* operation is applicable to a state when there is a nonterminal to the right of the dot. It causes us to add one new state to S_i for each alternative of that nonterminal. We put the dot at the beginning of the production in each new state, since we have not scanned any of its symbols yet. The pointer is set to i, since the state was created in S_i. Thus the predictor adds to S_i all productions which might generate substrings beginning at X_{i+1}.

In our example, we add to S_0

$$E \rightarrow .E+T \quad \dashv \quad 0 \qquad (2)$$

$$E \rightarrow .T \quad \dashv \quad 0 \qquad (3)$$

The k-symbol look-ahead string is $\dashv$, since it is after E in the original state. We must now process these two states. The predictor is also applicable to them. Operating on (2), it produces

$$E \rightarrow .E+T \quad + \quad 0 \qquad (4)$$

$$E \rightarrow .T \quad + \quad 0 \qquad (5)$$

with a look-ahead symbol + because it appears after E in (2). Operating on (3), it produces

$$T \rightarrow .T*P \quad \dashv \quad 0$$

$$T \rightarrow .P \quad \dashv \quad 0$$

Here the look-ahead symbol is $\dashv$ because T is last in the production and $\dashv$ is its look-ahead symbol. Now, the predictor, operating on (4) produces (4) and (5) again, but they are already in S_0, so we do nothing. From (5) it produces

$$T \rightarrow .T*P \quad + \quad 0$$

$$T \rightarrow .P \quad + \quad 0$$

The rest of S_0 is

$$T \rightarrow .T*P \quad * \quad 0$$

$$T \rightarrow .P \quad * \quad 0$$

$$P \rightarrow .a \quad \dashv \quad 0$$

$$P \rightarrow .a \quad + \quad 0$$

$$P \rightarrow .a \quad * \quad 0$$

The predictor is not applicable to any of the last three states. Instead the *scanner* is, because it is applicable just in case there is a terminal to the right of the dot. The scanner compares that symbol with X_{i+1}, and if they match, it adds the state to S_{i+1}, with the dot moved over one in the state to indicate that that terminal symbol has been scanned.

If X_1 = a, then S_1 is

$$P \rightarrow a. \quad \dashv \quad 0$$

$$P \rightarrow a. \quad + \quad 0 \qquad (6)$$

$$P \rightarrow a. \quad * \quad 0$$

these states being added by the scanner.

If we finish processing S_i and S_{i+1} remains empty, an error has occurred in the input string. Otherwise, we start to process S_{i+1}.

The third operation, the *completer*, is applicable to a state if its dot is at the end of its production. Thus the completer is applicable to each of these states in S_1. It compares the look-ahead string with $X_{i+1} \cdots X_{i+k}$. If they match, it goes back to the state set indicated by the pointer, in this case S_0, and adds all states from S_0 which have P to the right of the dot. It moves the dot over P in these states. Intuitively, S_0 is the state set we were in when we went looking for that P. We have now found it, so we go back to all the states in S_0 which caused us to look for a P, and we move the dot over the P in these states to show that it has been successfully scanned.

If X_2 = +, then the completer applied to (6) causes us to add to S_1

$$T \rightarrow P. \quad \dashv \quad 0$$

$$T \rightarrow P. \quad + \quad 0$$

$$T \rightarrow P. \quad * \quad 0$$

Applying the completer to the second of these produces

$$E \rightarrow T. \quad \dashv \quad 0$$

$$E \rightarrow T. \quad + \quad 0$$

$$T \rightarrow T.*P \quad \dashv \quad 0$$

$$T \rightarrow T.*P \quad + \quad 0$$

$$T \rightarrow T.*P \quad * \quad 0$$

and finally, from the second of these, we get

$$\phi \rightarrow E. \dashv \quad \dashv \quad 0$$

$$E \rightarrow E.+T \quad \dashv \quad 0$$

$$E \rightarrow E.+T \quad + \quad 0$$

The scanner then adds to S_2

$$E \rightarrow E+.T \quad \dashv \quad 0$$

$$E \rightarrow E+.T \quad + \quad 0$$

If the algorithm ever produces an S_{i+1} consisting of the single state

$$\phi \rightarrow E \dashv. \quad \dashv \quad 0$$

then we have correctly scanned an E and the $\dashv$, so we are finished with the string, and it is a sentence of the grammar.

A complete run of the algorithm on grammar AE is given in Figure 1. In this example, we have written as one all the states in a state set which differ only in their look-ahead string. (Thus "$\dashv$+*" as a look-ahead string stands for three states, with "$\dashv$", "+", and "*" as their respective look-ahead strings.)

The technique of using state sets and the look-ahead are derived from Knuth's work on LR(k) grammars [2]. In fact our algorithm bears a close relationship to Knuth's algorithm on LR(k) grammars except for the fact that

GRAMMAR AE

root: E → T | E+T input string = a+a*a
T → P | T*P
P → a

$k = 1$

S_0	$\phi \rightarrow .E\dashv$	$\dashv$	0	S_3	$P \rightarrow a.$	$\dashv +*$	2
(X_1=a)	$E \rightarrow .E+T$	$\dashv +$	0	(X_4=*)	$T \rightarrow P.$	$\dashv +*$	2
	$E \rightarrow .T$	$\dashv +$	0		$E \rightarrow E+T.$	$\dashv +$	0
	$T \rightarrow .T*P$	$\dashv +*$	0		$T \rightarrow T.*P$	$\dashv +*$	2
	$T \rightarrow .P$	$\dashv +*$	0				
	$P \rightarrow .a$	$\dashv +*$	0	S_4	$T \rightarrow T*.P$	$\dashv +*$	2
				(X_5=a)	$P \rightarrow .a$	$\dashv +*$	4
S_1	$P \rightarrow a.$	$\dashv +*$	0				
(X_2=+)	$T \rightarrow P.$	$\dashv +*$	0	S_5	$P \rightarrow a.$	$\dashv +*$	4
	$E \rightarrow T.$	$\dashv +$	0	(X_6=$\dashv$)	$T \rightarrow T*P.$	$\dashv +*$	2
	$T \rightarrow T.*P$	$\dashv +*$	0		$E \rightarrow E+T.$	$\dashv +$	0
	$\phi \rightarrow E.\dashv$	$\dashv$	0		$T \rightarrow T.*P$	$\dashv +*$	2
	$E \rightarrow E.+T$	$\dashv +$	0		$\phi \rightarrow E.\dashv$	$\dashv$	0
					$E \rightarrow E.+T$	$\dashv +$	0
S_2	$E \rightarrow E+.T$	$\dashv +$	0				
(X_3=a)	$T \rightarrow .T*P$	$\dashv +*$	2	S_6	$\phi \rightarrow E\dashv .$	$\dashv$	0
	$T \rightarrow .P$	$\dashv +*$	2				
	$P \rightarrow .a$	$\dashv +*$	2				

FIG. 1

he uses a stack rather than pointers to keep track of what to do after a parsing decision is made.

Note also that, although it did not develop this way, our algorithm is in effect a top-down parser [3] in which we carry along all possible parses simultaneously in such a way that we can often combine like subparses. This cuts down on duplication of effort and also avoids the left-recursion problem. (A straightforward top-down parser may go into an infinite loop on grammars containing left-recursion, i.e. $A \rightarrow A\beta$.)

4. The Recognizer

The following is a precise description of the recognition algorithm for input string $X_1 \cdots X_n$ and grammar G.

NOTATION. Number the productions of grammar G arbitrarily $1, \cdots, d-1$, where each production is of the form

$$D_p \rightarrow C_{p1} \cdots C_{p\bar{p}} \qquad (1 \leqq p \leqq d-1)$$

where $\bar{p}$ is the number of symbols on the right-hand side of the pth production. Add a 0th production

$$D_0 \rightarrow R \dashv$$

where R is the root of G, and $\dashv$ is a new terminal symbol.

Definition. A *state* is a quadruple $\langle p, j, f, \alpha \rangle$ where p, j, and f are integers $(0 \leqq p \leqq d-1)$ $(0 \leqq j \leqq \bar{p})$ $(0 \leqq f \leqq n+1)$ and α is a string consisting of k terminal symbols. *state set* is an ordered set of states. A *final state* is one in which $j = \bar{p}$. We *add* a state to a state set by putting it last in the ordered set *unless it is already a member.*

Definition. $H_k(\gamma) = \{\alpha \mid \alpha$ is terminal, $|\alpha| = k$, and $\exists \beta$ such that $\gamma \overset{*}{\Rightarrow} \alpha\beta\}$.

$H_k(\gamma)$ is the set of all k-symbol terminal strings which begin some string derived from γ. This is used in forming the look-ahead string for the states.

THE RECOGNIZER. This is a function of three arguments REC(G, $X_1 \cdots X_n$, k) computed as follows:

Let $X_{n+i} = \dashv$ $\quad (1 \leqq i \leqq k+1)$.
Let S_i be empty $\quad (0 \leqq i \leqq n+1)$.
Add $\langle 0,0,0,\dashv^k \rangle$ to S_0.

For $i \leftarrow 0$ step 1 until n do
Begin
Process the states of S_i in order, performing one of the following three operations on each state $s = \langle p, j, f, \alpha \rangle$.
(1) Predictor: If s is nonfinal and $C_{p(j+1)}$ is a nonterminal, then for each q such that $C_{p(j+1)} = D_q$, and for each $\beta \in H_k(C_{p(j+2)} \cdots C_{p\bar{p}}\alpha)$ add $\langle q, 0, i, \beta \rangle$ to S_i.
(2) Completer: If s is final and $\alpha = X_{i+1} \cdots X_{i+k}$, then for each $\langle q,l,g,\beta \rangle \in S_f$ (after all states have been added to S_f) such that $C_{q(l+1)} = D_p$, add $\langle q,l+1,g,\beta \rangle$ to S_i.
(3) Scanner: If s is nonfinal and $C_{p(j+1)}$ is terminal, then if $C_{p(j+1)} = X_{i+1}$, add $\langle p, j+1, f, \alpha \rangle$ to S_{i+1}.
If S_{i+1} is empty, return rejection.
If $i = n$ and $S_{i+1} = \{\langle 0,2,0,\dashv \rangle\}$, return acceptance.
End

Notice that the ordering imposed on state sets is not important to their meaning but is simply a device which allows their members to be processed correctly by the algorithm. Also note that i cannot become greater than n without either acceptance or rejection occurring because of the fact that $\dashv$ appears only in production zero. This does not really represent a complete description of the algorithm until we describe in detail how all these operations are implemented on a machine. The following description assumes a knowledge of basic list processing techniques.

Implementation

(1) For each nonterminal, we keep a linked list of its alternatives, for use in prediction.

(2) The states in a state set are kept in a linked list so they can be processed in order.

(3) In addition, as each state set S_i is constructed, we put entries into a vector of size i. The fth entry in this vector $(0 \leqq f \leqq i)$ is a pointer to a list of all states in S_i with pointer f, i.e. states of the form $\langle p, j, f, \alpha \rangle \in S_i$ for some p, j, α. Thus, to test if a state $\langle p, j, f, \alpha \rangle$ has already been added to S_i, we search through the list pointed to by the fth entry in this vector. (This takes an amount of time independent of f.) The vector and lists can be discarded after S_i is constructed.

(4) For the use of the completer, we also keep, for each state set S_i and nonterminal N, a list of all states $\langle p, j, f, \alpha \rangle \in S_i$ such that $C_{p(j+1)} = N$.

(5) If the grammar contains null productions $(A \rightarrow \lambda)$, we cannot implement the completer in a straightforward way. When performing the completer on a null state $(A \rightarrow \cdot\ \alpha\ i)$ we want to add to S_i each state in S_i with A to the right of the dot. But one of these may not have been added to S_i yet. So we must note this and check for it when we add more states to S_i.

The above implementation description is not meant to be the only way or the best way to implement the algorithm. It is merely a method which does allow the

algorithm to achieve the time and space bounds which we quote in Section 5.

The correctness of this recognizer has been proved in [1]. It requires no restrictions of any kind on the context-free grammar to be successful.

5. Time and Space Bounds

To develop some idea of the efficiency of the algorithm, we can use a formal model of a computer and measure the time as the number of primitive steps executed by this model and the space as the number of storage locations used. We use a random access model (described in the Appendix) because we feel that this model represents most accurately the properties of real computers which are relevant to syntax analysis.

We are interested in upper bounds on the time (and less important the space) as a function of n (the length of the input string) for various classes of context-free grammars. Specifically, an n^2 algorithm for a subclass A of grammars means that there is some number C (which may depend on the size of the grammar, but not on n), such that Cn^2 is an upper bound on the number of primitive steps required to parse any string of length n with respect to a grammar in class A.

THE GENERAL CASE. Our algorithm is an n^3 recognizer in general. The reasons for this are:

(a) The number of states in any state set S_i is proportional to i ($\sim i$) because the ranges of the p, j, and α components of a state are bounded, while only the f component depends on i, and it is bounded by n.

(b) The scanner and predictor operations each execute a bounded number of steps per state in any state set. So the total time for processing the states in S_i plus the scanner and predictor operations is $\sim i$.

(c) The completer executes $\sim i$ steps for each state it processes in the worst case because it may have to add $\sim f$ states for S_f, the state set pointed back to. So it takes $\sim i^2$ steps in S_i.

(d) Summing from $i = 0, \cdots, n+1$ gives $\sim n^3$ steps.

This bound holds even if the look-ahead feature is not used (by setting $k = 0$). The bound is no better than that obtained by Younger [4] for Cocke's algorithm [5], but our algorithm is better for two reasons. Ours does not require the grammar to be put into any special form (Cocke's required normal form), and ours actually does better than n^3 on most grammars, as we shall show (Cocke's always requires n^3). Furthermore, although Younger's n^3 result is obtained on a Turing machine, his algorithm is in no way made faster by putting it on a random access machine.

UNAMBIGUOUS GRAMMARS. The completer is the only operation which forces us to use i^2 steps for each state set we process, making the whole thing n^3. So the question is, in what cases does this operation involve only i steps instead of i^2? If we examine the state set S_i after the completer has been applied to it, there are at most proportional to i states in it. So unless some of them were added in more than one way (this can happen; that's why we must test for the existence of a state before we add it to a state set) then it took at most $\sim i$ steps to do the operation.

In the case that the grammar is unambiguous and reduced, we can show that each such state gets added in only one way. Assume that the state $\langle q, j+1, f, \alpha\rangle$ is added to S_i in two different ways by the completer. Then we have

$$s_1 = \langle p_1, \bar{p}_1, f_1, X_{i+1} \cdots X_{i+k}\rangle \in S_i,$$

$$s_2 = \langle p_2, \bar{p}_2, f_2, X_{i+1} \cdots X_{i+k}\rangle \in S_i,$$

$$\langle q, j, f, \alpha\rangle \in S_{f_1} \quad \text{and} \quad S_{f_2}, \qquad D_{p_1} = C_{q(j+1)} = D_{p_2}$$

and either $p_1 \neq p_2$ or $f_1 \neq f_2$, for otherwise s_1 and s_2 would be the same state.

So we have

$$X_1 \cdots X_f C_{q1} \cdots C_{q(j+1)} \overset{*}{\Rightarrow} X_1 \cdots X_{f_1} C_{p_1 1} \cdots C_{p_1 \bar{p}_1} \overset{*}{\Rightarrow} X_1 \cdots X_i$$

and

$$X_1 \cdots X_f C_{q1} \cdots C_{q(j+1)} \overset{*}{\Rightarrow} X_1 \cdots X_{f_2} C_{p_2 1} \cdots C_{p_2 \bar{p}_2} \overset{*}{\Rightarrow} X_1 \cdots X_i$$

and since $p_1 = p_2$ and $f_1 = f_2$ cannot both be true, the above two derivations of $X_1 \cdots X_i$ are represented by different derivation trees. Therefore since the grammar is reduced, every nonterminal generates some terminal string, and so there exists an ambiguous sentence $X_1 \cdots X_i\alpha$ for some α.

So if the grammar is unambiguous, the completer executes $\sim$i steps per state set and the time is bounded by n^2. Notice that the time is also n^2 for grammars with bounded ambiguity since each state can then be added by the completer only a bounded number of times. In [1] we show that the time is n^2 for an even larger class of grammars, and thereby also obtain Younger's n^2 results for linear and metalinear grammars [6].

Kasami [7] has also obtained independently the result for unambiguous grammars, but his algorithm (which is a modification of Cocke's) has the disadvantage that it requires the grammar in normal form. His algorithm, like ours, achieves its time bound on a random access machine only.

LINEAR TIME. We now characterize the class of grammars which the algorithm will do in time n. We notice that for some grammars the number of states in a state set can grow indefinitely with the length of the string being recognized. For some others there is a fixed bound on the size of any state set. We call the latter grammars *bounded state* grammars. They can be done by the algorithm in time n for the following reason. Let b be the bound on the number of states in any state set. Then the processing of the states in a state set together with the scanner and predictor requires $\sim b$ steps, and the completer requires $\sim b^2$ steps. Summing over all the state sets gives us $\sim b^2 n$ or $\sim n$ steps.

So the class of time n grammars for our algorithm in-

cludes the bounded state grammars, but it actually includes more. We now examine how this class of grammars compares with those that can be done in linear time by other algorithms. Most of the previously mentioned "restricted" algorithms work on some subclass of grammars which they can do in time n, and we will henceforth call them *time n* algorithms. Knuth's LR(k) algorithm [2] works on a class of grammars which includes those of just about all the others, so his will be a good one for comparison if we expect to do well.

It turns out that almost all LR(k) grammars are bounded state (except for certain right recursive grammars). And even though some LR(k) grammars may not be bounded state, all of them can be done in time n by our algorithm if a look-ahead of k or greater is used. In fact any finite union of LR(k) grammars (obtained by combining the grammars in a straightforward way in order to generate the union of the languages) is a time n grammar for our algorithm given the proper look-ahead. (This is proved in [1].)

It is here that the look-ahead feature of the algorithm is most obviously useful. We can obtain the n^3 and n^2 results without it, but we cannot do all LR(k) grammars in time n without it. In addition, a look-ahead of $k = 1$ is a good practical device for cutting down on a lot of extraneous processing with many common grammars.

The time n grammars for our algorithm, then, include bounded state grammars, finite unions of LR(k) grammars, and others. They include many grammars which are ambiguous, and some with unbounded degree of ambiguity, but unfortunately there are also unambiguous grammars which require time n^2.

GRAMMAR UBDA

root: $A \rightarrow x \mid AA$ sentences: x^n ($n \geqq 1$)

REC(UBDA, x^4, 1)

S_0	$\phi \rightarrow .A\dashv$	$\dashv$	0
	$A \rightarrow .x$	$\dashv x$	0
	$A \rightarrow .AA$	$\dashv x$	0
S_1	$A \rightarrow x.$	$\dashv x$	0
	$\phi \rightarrow A.\dashv$	$\dashv$	0
	$A \rightarrow A.A$	$\dashv x$	0
	$A \rightarrow .x$	$\dashv x$	1
	$A \rightarrow .AA$	$\dashv x$	1
S_2	$A \rightarrow x.$	$\dashv x$	1
	$A \rightarrow AA.$	$\dashv x$	0
	$A \rightarrow A.A$	$\dashv x$	1
	$\phi \rightarrow A.\dashv$	$\dashv$	0
	$A \rightarrow A.A$	$\dashv x$	0
	$A \rightarrow .x$	$\dashv x$	2
	$A \rightarrow .AA$	$\dashv x$	2
S_3	$A \rightarrow x.$	$\dashv x$	2
	$A \rightarrow AA.$	$\dashv x$	1
	$A \rightarrow AA.$	$\dashv x$	0^2
	$A \rightarrow A.A$	$\dashv x$	2
	$A \rightarrow A.A$	$\dashv x$	1
	$\phi \rightarrow A.\dashv$	$\dashv$	0
	$A \rightarrow A.A$	$\dashv x$	0
	$A \rightarrow .x$	$\dashv x$	3
	$A \rightarrow .AA$	$\dashv x$	3
S_4	$A \rightarrow x.$	$\dashv x$	3
	$A \rightarrow AA.$	$\dashv x$	2
	$A \rightarrow AA.$	$\dashv x$	1^2
	$A \rightarrow AA.$	$\dashv x$	0^3
	$A \rightarrow A.A$	$\dashv x$	3
	$A \rightarrow A.A$	$\dashv x$	2
	$A \rightarrow A.A$	$\dashv x$	1
	$\phi \rightarrow A.\dashv$	$\dashv$	0
	$A \rightarrow A.A$	$\dashv x$	0
	$A \rightarrow .x$	$\dashv x$	4
	$A \rightarrow .AA$	$\dashv x$	4
S_5	$\phi \rightarrow A\dashv.$	$\dashv$	0

FIG. 2

The following examples illustrate some of the ideas in this section. Grammar UBDA (Figure 2) actually requires time proportional to n^3. Notice that state

$$A \rightarrow AA. \quad \dashv x \quad 0^2$$

gets added twice by the completer. This is signified by the superscript on the 0. One can tell by the looks of the superscripts on states

$$A \rightarrow AA. \quad \dashv x \quad 2$$

$$A \rightarrow AA. \quad \dashv x \quad 1^2$$

$$A \rightarrow AA. \quad \dashv x \quad 0^3$$

in S_4 that there are $\sim i$ states, each of which is added $\sim i$ times, in $\sim n$ state sets, producing the n^3 behavior.

Grammar BK (Figure 3) has unbounded ambiguity, but it is a time n grammar, and in fact it is bounded state. This can be seen because all the state sets after S_1 are the same size. Grammar PAL (Figure 4) is an unambiguous grammar which requires time n^2. S_i and S_{i+1} each have $i + 4$ states in them, up to and including S_n, so the total number of states is $\sim n^2$.

SPACE. Since the space is taken up by $\sim n$ state sets, each containing $\sim n$ states, the space bound is n^2 in general. This is comparable to Cocke's and Kasami's algorithms, which also require n^2. However, it has the advantage over Cocke's in that the n^2 is only an upper bound for ours, while his requires n^2 all the time.

GRAMMAR BK

root: $K \rightarrow \mid KJ$ sentences: x^n ($n \geqq 0$)
$J \rightarrow F \mid I$
$F \rightarrow x$
$I \rightarrow x$

REC(BK, x^n, 1)

S_0	$\phi \rightarrow .K\dashv$	$\dashv$	0
	$K \rightarrow .$	$\dashv x$	0
	$K \rightarrow .KJ$	$\dashv x$	0
	$\phi \rightarrow K.\dashv$	$\dashv$	0
	$K \rightarrow K.J$	$\dashv x$	0
	$J \rightarrow .F$	$\dashv x$	0
	$J \rightarrow .I$	$\dashv x$	0
	$F \rightarrow .x$	$\dashv x$	0
	$I \rightarrow .x$	$\dashv x$	0
S_1	$F \rightarrow x.$	$\dashv x$	0
	$I \rightarrow x.$	$\dashv x$	0
	$J \rightarrow F.$	$\dashv x$	0
	$J \rightarrow I.$	$\dashv x$	0
	$K \rightarrow KJ.$	$\dashv x$	0^2
	$K \rightarrow K.J$	$\dashv x$	0
	$\phi \rightarrow K.\dashv$	$\dashv$	0
	$J \rightarrow .F$	$\dashv x$	1
	$J \rightarrow .I$	$\dashv x$	1
	$F \rightarrow .x$	$\dashv x$	1
	$I \rightarrow .x$	$\dashv x$	1

S_i ($2 \leqq i \leqq n$)

$F \rightarrow x.$	$\dashv x$	$i-1$
$F \rightarrow x.$	$\dashv x$	$i-1$
$J \rightarrow F.$	$\dashv x$	$i-1$
$J \rightarrow I.$	$\dashv x$	$i-1$
$K \rightarrow KJ.$	$\dashv x$	0^2
$K \rightarrow K.J$	$\dashv x$	0
$\phi \rightarrow K.\dashv$		0
$J \rightarrow .F$	$\dashv x$	i
$J \rightarrow .I$	$\dashv x$	i
$F \rightarrow .x$	$\dashv x$	i
$I \rightarrow .x$	$\dashv x$	i

S_{n+1}

$\phi \rightarrow K\dashv.$	$\dashv$	0

FIG. 3

GRAMMAR PAL

$A \rightarrow x \mid xAx$ sentences: x^n ($n \geqq 1$, n odd)

REC(PAL, x^5, 0):

S_0	$\phi \rightarrow .A\dashv$	0	S_4	$A \rightarrow xAx.$	1
	$A \rightarrow .x$	0		$A \rightarrow x.$	3
	$A \rightarrow .xAx$	0		$A \rightarrow x.Ax$	3
S_1	$A \rightarrow x.$	0		$A \rightarrow xA.x$	0
	$A \rightarrow x.Ax$	0		$A \rightarrow xA.x$	2
	$\phi \rightarrow A.\dashv$	0		$A \rightarrow .x$	4
	$A \rightarrow .x$	1		$A \rightarrow .xAx$	4
	$A \rightarrow .xAx$	1	S_5	$A \rightarrow xAx.$	0
S_2	$A \rightarrow x.$	1		$A \rightarrow xAx.$	2
	$A \rightarrow x.Ax$	1		$A \rightarrow x.$	4
	$A \rightarrow xA.x$	0		$A \rightarrow x.Ax$	4
	$A \rightarrow .x$	2		$\phi \rightarrow A.\dashv$	0
	$A \rightarrow .xAx$	2		$A \rightarrow xA.x$	1
S_3	$A \rightarrow xAx.$	0		$A \rightarrow xA.x$	3
	$A \rightarrow x.$	2		$A \rightarrow .x$	5
	$A \rightarrow x.Ax$	2		$A \rightarrow xAx.$	5
	$\phi \rightarrow A.\dashv$	0			
	$A \rightarrow xA.x$	1	S_6	$\phi \rightarrow A\dashv.$	0
	$A \rightarrow .x$	3			
	$A \rightarrow .xAx$	3			

FIG. 4

6. Empirical Results

We have programmed the algorithm and tested it against the top-down and bottom-up parsers evaluated by Griffiths and Petrick [8]. These are the oldest of the context-free parsers, and they depend heavily on backtracking. Perhaps because of this, their upper bounds for time are exponential (C^n for some constant C). However, they also can do well on some grammars, and both have been used in numerous compiler-compilers, so it will be interesting to compare our algorithm with them.

The Griffiths and Petrick data is not in terms of actual running times but in terms of "primitive operations." They have expressed their algorithms as sets of nondeterministic rewriting rules for a Turing-machine-like device. Each application of one of these is a primitive operation. We have chosen as our primitive operation the act of adding a state to a state set (or attempting to add one which is already there). We feel that this is comparable to their primitive operation because both are in some sense the most complex operation performed by the algorithm whose complexity is independent of the size of the grammar or input string.

We compare the algorithms on seven different grammars. Two of their examples were not used because the exact grammar was not given. For the first four, Griffiths and Petrick were able to find closed-form expressions for their results, so we did also (Figure 5). BU and TD are the bottom-up and top-down algorithms respectively and SBU and STD are their selective versions. It is obvious from these results that SBU is by far the best of the other algorithms, and the rest of their data bears this out. There-

	G1	G2	G3	G4
root:	$S \rightarrow Ab$ $A \rightarrow a \mid Ab$	$S \rightarrow aB$ $B \rightarrow aB \mid b$	$S \rightarrow ab \mid aSb$	$S \rightarrow AB$ $A \rightarrow a \mid Ab$ $B \rightarrow bc \mid bB \mid Bd$

Grammar	Sentence	TD	STD	BU	SBU	Ours
G1	ab^n	$(n^2+7n+2)/2$	$(n^2+7n+2)/2$	$9n+5$	$9n+5$	$4n+7$
G2	a^nb	$3n+2$	$2n+2$	$11 \cdot 2^n + 7$	$4n+4$	$4n+4$
G3	a^nb^n	$5n-1$	$5n-1$	$11 \cdot 2^{n-1} - 5$	$6n$	$6n+4$
G4	ab^ncd	$\sim 2^{n+6}$	$\sim 2^{n+2}$	$\sim 2^{n+5}$	$(n^3+21n^2+46n+15)/3$	$18n+8$

FIG. 5

PROPOSITIONAL CALCULUS GRAMMAR

root: $F \rightarrow C \mid S \mid P \mid U$
$C \rightarrow U \supset U$
$U \rightarrow (F) \mid \sim U \mid L$
$L \rightarrow L' \mid p \mid q \mid r$
$S \rightarrow U \vee S \mid U \vee U$
$P \rightarrow U \wedge P \mid U \wedge U$

Sentence	Length	PA	SBU	Ours
p	1	14	18	28
$(p \wedge q)$	5	89	56	68
$(p' \wedge q) \vee r \vee p \vee q'$	13	232	185	148
$p \supset ((q \supset \sim (r' \vee (p \wedge q))) \supset (q' \vee r)$	26	712	277	277
$\sim(\sim p' \wedge (q \vee r) \wedge p'))$	17	1955	223	141
$((p \wedge q) \vee (q \wedge r) \vee (r \wedge p')) \supset \sim((p' \vee q') \wedge (r' \vee p))$	38	2040	562	399

FIG. 6

GRAMMAR GRE

root: $X \rightarrow a \mid Xb \mid Ya$
$Y \rightarrow e \mid YdY$

Sentence	Length	PA	SBU	Ours
ededea	6	35	52	33
$ededeab^4$	10	75	92	45
$ededeab^{10}$	16	99	152	63
$ededeab^{200}$	206	859	2052	633
$(ed)^4eabb$	12	617	526	79
$(ed)^7eabb$	18	24352	16336	194
$(ed)^8eabb$	20	86139	54660	251

FIG. 7

GRAMMAR NSE

root: $S \rightarrow AB$
$A \rightarrow a \mid SC$
$B \rightarrow b \mid DB$
$C \rightarrow c$
$D \rightarrow d$

Sentence	Length	SBU	Ours
adbcddb	7	43	44
$ad^3bcbcd^3bcd^4b$	18	111	108
$adbcd^2bcd^5bcd^3b$	19	117	114
$ad^{18}b$	20	120	123
$a(bc)^3d^3(bcd)^2dbcd^4b$	24	150	141
$a(bcd)^2dbcd^3bcb$	16	100	95

FIG. 8

fore we compare our algorithm with SBU only. We used our algorithm with $k = 0$. The two are comparable on G1, G2, G3, the simple grammars, but on G4, which is very ambiguous, ours is clearly superior—n to n^3.

For the next three grammars we present only the raw data (Figures 6–8). The data for our algorithm was obtained by programming it and having the program compute the number of primitive operations it performed. We have also included the data from [8] on PA, the predictive analyzer, which is a modified top-down algorithm. On the propositional calculus grammar, PA seems to be running in time n^2, while both SBU and ours run in time n, with ours a little faster. Grammar GRE produces two kinds of behavior. All three algorithms go up linearly with the number of "b" 's, with SBU using a considerably higher constant coefficient. However, PA and SBU go up exponentially with the number of "ed" 's, while ours goes up as the square. Grammar NSE is quite simple, and each algorithm takes time n with the same coefficient.

So we conclude that our algorithm is clearly superior to the backtracking algorithms. It performs as well as the best of them on all seven grammars and is substantially faster on some.

There are at least four distinct general context-free algorithms besides ours—TD, BU, Kasami's n^2, and Cocke's n^3. We have shown so far that our algorithm achieves time bounds which are as good as those of any of these algorithms, or better. However, we are also interested in how our algorithm compares with these algorithms in a practical sense, not just at an upper bound.

We have just presented some empirical results in this section which indicate that our algorithm is better than TD and BU. Furthermore, our algorithm must be superior to Cocke's since his always achieves its upper bound of n^3. This leaves Kasami's. His algorithm [7] is actually described as an algorithm for unambiguous grammars, but it can easily be extended to a general algorithm. In this form we suspect that it will have an n^3 bound in general and will be n^2 as often as ours. We are aware of no results about the class of grammars that it can parse in time n.

7. The Practical Use of the Algorithm

In this section we discuss the question, in what areas and in what form can the algorithm best be put to use?

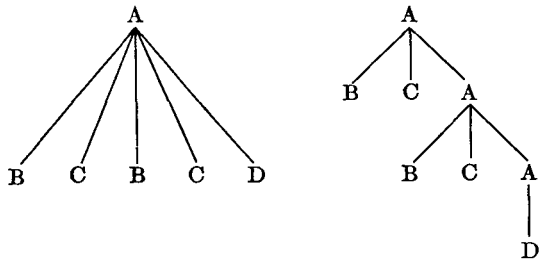
THE FORM. Before we can do much with it, we must make the recognizer into a parser. This is done by altering the recognizer so that it builds a parse tree as it does the recognition process. Each time we perform the completer operation adding a state $E \rightarrow \alpha D.\beta$ g (ignoring look-ahead) we construct a pointer from the instance of D in that state to the state $D \rightarrow \gamma.$ f which caused us to do the operation. This indicates that D was parsed as γ. In case D is ambiguous there will be a set of pointers from it, one for each completer operation which caused $E \rightarrow \alpha D \cdot \beta$ g to be added to the particular state set. Each symbol in γ will also have pointers from it (unless it is terminal), and so on, thus representing the derivation tree for D.

In this way, when we reach the terminating state $\phi \rightarrow R \dashv .$ 0 we will have the parse tree for the sentence hanging from R if it is unambiguous, and otherwise we will have a factored representation of all possible parse trees. In [1] a precise description of this process is given.

The time bounds for the parser are the same as those of the recognizer, while the space bound goes up to n^3 in general in order to store the parse trees.

We recommend using consistently a look-ahead of $k = 1$. In fact it would probably be most efficient to implement the algorithm to do just a look-ahead of 1. To implement the full look-ahead for any k would be more costly in programming effort and less efficient overall since so few programming languages need the extra look-ahead. Most programming languages use only a one character context to disambiguate their constructs, and if two characters are needed in some cases, our algorithm has the nice property that it will not fail, it may just take a little longer.

Our algorithm has the useful property that it can be modified to handle an extension of context-free grammars which makes use of the Kleene star notation. In this notation: $A \rightarrow \{BC\} * D$ means A may be rewritten as an arbitrary number (including 0) of BC's followed by a D. It generates a language equivalent to that generated by $A \rightarrow D \mid BCA$. However, the parse structure given to the language is different in the two grammars:



Structures like that on the left cannot be obtained using context-free grammars at all, so this extension is useful. The modification to our algorithm which implements it involves two additional operations:

(1) Any state of the form

$$A \rightarrow \alpha.\{\beta\}^*\gamma \quad f$$

is replaced by

$$A \rightarrow \alpha\{.\beta\}^*\gamma \quad f$$

$$A \rightarrow \alpha\{\beta\}^*.\gamma \quad f$$

indicating that β may be present or absent.

(2) Any state of the form

$$A \rightarrow \alpha\{\beta.\}^*\gamma \quad f$$

is replaced by

$$A \rightarrow \alpha\{.\beta\}^*\gamma \quad f$$

$$A \rightarrow \alpha\{\beta\}^*.\gamma \quad f$$

indicating the β may be repeated or not.

THE USE. The algorithm will probably be most useful in natural language processing systems where the full power of context-free grammars is used. It should also be useful in compiler writing systems and extendible languages. In most compiler writing systems and extendible languages, the programmer is allowed to express the syntax (or the syntax extension) of his language in something like BNF, and the system uses a parser to analyze subsequent programs in this language. Programming language grammars tend to lie in a restricted subset of context-free grammars which can be processed efficiently, yet some compiler writing systems in fact use general parsers, so ours may be of use here. In addition to its efficiency properties, ours has the advantage that it accepts the grammar in the form in which it is written, so that semantic routines can be associated with productions without fear that the parser will not reflect the original structure of the grammar.

Our algorithm will not compete so favorably with the time n algorithms, however. Certainly ours will do in time n any grammar that a time n parser can do at all, but this does not take into account the constant coefficient of n. Most of the time n algorithms really consist of a two-fold process. First they compile from an acceptable grammar a parser for that particular grammar, and then the grammar may be discarded and the compiled parser used directly to analyze strings. This allows the time n algorithms to incorporate much specialized information into the compiled parser, thus reducing the coefficient of n to something quite small—probably an order of magnitude less than that of our algorithm.

Consequently we have developed a compilation process for our algorithm which works only on time n grammars and reduces our coefficient to approximately the same order of magnitude as those of the time n parsers. This may make our algorithm competitive with them, but we have not implemented and tested it, so this is speculation. Some sort of efficient time n parser for a larger class of grammars is needed, however, because most restricted parsers suffer from the problem that the grammar one naturally writes for many programming languages is not acceptable to them, and much fiddling must be done with the grammar to get it accepted. Knuth's algorithm is an exception to this, but it has the problem that the size of the compiled parser is much too great for reasonable programming language grammars (see [1, p. 129]). Unfortunately, our compiled algorithm, since it is similar to Knuth's, may also have these problems.

8. Conclusion

In conclusion let us emphasize that our algorithm not only matches or surpasses the best previous results for times n^3 (Younger), n^2 (Kasami) and n (Knuth), but it does this with one single algorithm which does not have specified to it the class of grammars it is operating on and does not require the grammar in any special form. In other words, Knuth's algorithm works only on LR(k) grammars and Kasami's (at least in his paper) only on unambiguous ones, but ours works on them all and seems to do about as well as other algorithms automatically.

Appendix

RANDOM ACCESS MACHINE. This model has an unbounded number of registers (counters), each of which may contain any nonnegative integer. These registers are named (addressed) by successive nonnegative integers. The primitive operations which are allowed on these registers are as follows:

(1) Store 0 or the contents of one register into another.
(2) Test the contents of one register against 0 or against the contents of another register for equality.
(3) Add 1 or subtract 1 from the contents of a register (taking $0 - 1 = 0$).
(4) Add the contents of one register to another.

The control for this model is a normal finite state device. The most important property of this machine is that in the above four operations, the register R to be operated on may be specified in two ways:

(1) R is the register whose address is n (register n).
(2) R is the register whose address is the contents of register n.

This second mode (sometimes called indirect addressing) plus primitive operation 4 (used for array accessing) gives our model the random access property. The time is measured by the number of primitive operations performed, and the space is measured by the number of registers used in any of these operations.

Acknowledgments. I am deeply indebted to Robert Floyd for his guidance in this research. I also benefited from discussions with Albert Meyer, Rudolph Krutar, and James Gray, and from detailed criticisms by the referees.

RECEIVED FEBRUARY, 1969; REVISED JUNE, 1969

REFERENCES

1. EARLEY, J. An efficient context-free parsing algorithm. Ph.D. Thesis, Comput. Sci. Dept., Carnegie-Mellon U., Pittsburgh, Pa., 1968.
2. KNUTH, D. E. On the translation of languages from left to right. *Information and Control 8* (1965), 607–639.
3. FLOYD, R. W. The syntax of programming languages—a survey. *IEEE Trans. EC-13*, 4 (Aug. 1964).
4. YOUNGER, D. H. Recognition and parsing of context-free languages in time n^3. *Information and Control 10* (1967), 189–208.
5. HAYS, D. Automatic language-data processing. In *Computer Applications in the Behavioral Sciences*, H. Borko (Ed.) Prentice Hall, Englewood Cliffs, N.J., 1962.
6. YOUNGER, D. H. Context-free language processing in time n^3. General Electric R & D Center, Schenectady, N.Y., 1966.
7. KASAMI, T., AND TORII, K. A syntax-analysis procedure for unambiguous context-free grammars. *J. ACM 16*, 3 (July 1969), 423–431.
8. GRIFFITHS, T., AND PETRICK, S. On the relative efficiencies of context-free grammar recognizers. *Comm. ACM 8*, 5 (May 1965), 289–300.
9. FELDMAN, J., AND GRIES, D. Translator writing systems. *Comm. ACM 11*, 2 (Feb. 1968), 77–113.