AN EFFICIENT CONTEXT-FREE PARSING ALGORITHM

Jay Earley

Computer Science Department
Carnegie-Mellon University
Pittsburgh, Pennsylvania
August, 1968

Submitted to Carnegie-Mellon University
in partial fulfillment of the requirements
for the degree of Doctor of Philosophy

This work was supported by the Advanced Research Projects Agency of the Office of the Secretary of Defense (F44620-67-C-0058) and is monitored by the Air Force Office of Scientific Research. This document has been approved for public release and sale; its distribution is unlimited.

CARNEGIE-MELLON UNIVERSITY

CARNEGIE INSTITUTE OF TECHNOLOGY

COLLEGE OF ENGINEERING AND SCIENCE

THESIS

SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF Doctor of Philosophy

TITLE An Efficient Context-Free Parsing Algorithm

PRESENTED BY Jay Earley

ACCEPTED BY THE DEPARTMENT OF Computer Science

Robert W. Floyd — MAJOR PROFESSOR — Aug 14 1968 — DATE

Alan J. Perlis — DEPARTMENT HEAD — Aug. 14 1968 — DATE

APPROVED BY THE COMMITTEE ON GRADUATE DEGREES

[signature] — CHAIRMAN — Oct 15, 1968 — DATE

Jay Earley
An efficient context-free parsing algorithm

ABSTRACT

This paper describes a parsing algorithm for context-free grammars, which is of interest because of its efficiency. The algorithm runs in time proportional to n^3 (where n is the length of the input string) on all context-free grammars. It runs in time proportional to n^2 on unambiguous grammars, and we actually show that it is n^2 on a considerably larger class of grammars than this, but not on all grammars. These two results are not new, but they have been attained previously by two different algorithms, both of which require the grammar to be put into a special form before they are applicable.

The algorithm runs in linear time on a class of grammars which includes LR(K) grammars and finite unions of them (and the LR(K) grammars include those of essentially all published algorithms which run in time n), and a large number of other grammars. These time n grammars in a practical sense include almost all unambiguous grammars, many ambiguous ones, and probably all programming language grammars.

We present a method for compiling a recognizer from a time n grammar which runs much faster than our original algorithm would have, working directly with the grammar as it is recognized. We show some undecidability results about the class of grammars that are compilable by this method.

The space bound for the algorithm is n^2, and by using a garbage collector which we describe, this can be cut down to n in a large number of cases.

The algorithm can easily be converted into a parser, which has the same bounds as the recognizer except that the space bound goes up to n^3 in order to store all the parses of very ambiguous grammars.

The time results we have obtained are only valid for a random access model of a computer. The n^3 result is the only one which carries over to a Turing-machine-like model.

Our algorithm was clearly superior to the top-down and bottom-up algorithms in a practical comparison with data from the Griffiths and Petrick article.

ACKNOWLEDGEMENTS

I am deeply indebted to Professor Robert Floyd who, as my thesis advisor, guided my research. Professor Albert Meyer also showed interest and provided guidance. I am also indebted to Rudolph Krutar for discussions on the implementation of the algorithm and to Professor Alan Perlis for suggestions on the final draft of the thesis.

CONTENTS

Section	Title	Page
I.	INTRODUCTION	1
II.	TERMINOLOGY	3
III.	PREVIOUS WORK	6
IV.	INTUITIVE EXPLANATION	9
V.	THE RECOGNIZER	16
VI.	RECOGNITION TIME	26
VII.	BOUNDED DIRECT AMBIGUITY	30
VIII.	TIME n^2	33
IX.	TIME n	44
X.	THE COMPILED ALGORITHM	69
XI.	AN EXAMPLE	89
XII.	SOME UNDECIDABILITY RESULTS	97
XIII.	SPACE	103
XIV.	THE PARSER	106
XV.	TWO MODELS	112
XVI.	EMPIRICAL RESULTS	115
XVII.	THE PRACTICAL USE OF THE ALGORITHM	122
XVIII.	FURTHER RESEARCH AND CONCLUSION	135
	REFERENCES	137

I. INTRODUCTION

Context-free grammars have been used extensively for describing the syntax of programming languages and natural languages. They are often called BNF grammars when used for programming languages [Na 63]. The analysis of the syntax (parsing) of programs or sentences is a crucial part of the implementation of compilers and interpreters for programming languages and of programs which "understand" or translate natural languages. Therefore the development of automatic parsing algorithms for context-free grammars has importance in these areas of computer science.

We need some way of evaluating the numerous parsing schemes which have been developed. In order that this evaluation be unbiased by a particular computer, a particular method of implementation, or a particular set of test grammars, we will formalize our evaluation criteria and use mathematical rather than experimental methods for comparison. We are interested in how fast the algorithm runs, how much space it uses, and how widely applicable it is. To evaluate time and space, we use a formal model of an idealized computer and count the number of steps an algorithm takes and the number of registers it uses (see Section XV). To evaluate the applicability, we use a formal model of a grammar and consider what subclasses of grammars can be handled correctly by the algorithm.

These formal techniques are some of those used by researchers in complexity theory--the study of the intrinsic difficulty of computing various functions. In fact the results in this paper can be thought of as complexity results.

In particular, we are more concerned with time than space. This is because the space requirements for context-free language processing generally fall within the capabilities of most large computers, but the time requirements often are not reasonable for the application. We consider the length of the string being parsed as the most important parameter in evaluating the time. It is more critical than the grammar size because the way in which the time depends on the grammar size seems to be pretty much the same over different algorithms and different classes of grammars, but the dependence of the time on the string length varies greatly.

We are concerned with upper bounds on the time required for various algorithms on various subclasses of grammars. Specifically, if we speak of an n^2 algorithm for a subclass A of grammars, we mean that there is some number C (which may depend on the size of the grammar, but not on the length of the string), such that Cn^2 is an upper bound on the number of steps required in our model to parse any string of length n with respect to any grammar which is in class A.

Those who are not interested in the formal properties of the algorithm need read only Sections I through IV, the definition of the algorithm and its implementation (pages 16-18 and 26-27), and Sections X, XI, XIII, XIV, XVI, and XVII.

II. TERMINOLOGY

A language is a set of strings over a finite set of symbols. We call these terminal symbols and represent them by lower case letters: a, b, c. Since most interesting languages are infinite sets, we use a context-free grammar as a formal device for specifying which strings are in the set. It uses another set of symbols, the non-terminals, which we can think of as syntactic classes. We will use capitals for non-terminals: A, B, C. There is a finite set of productions or rewriting rules of the form $A \rightarrow a$. The non-terminal which stands for "sentence" is called the root R of the grammar. The productions with a particular non-terminal D on their left sides are called the alternatives of D. We will hereafter use grammar to mean context-free grammar.

Strings of either terminals or non-terminals will be represented by Greek letters: α, β, γ. The empty string is Λ. α^k represents $\underbrace{\alpha \ldots \alpha}_{k \text{ times}}$. $a^* = \Lambda \cup \{a^i \mid i \geq 1\}$. $|\alpha|$ is the number of symbols in α.

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 non-terminals 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 <u>derived</u> from α) if $\exists$ strings $\alpha_0, \alpha_1, \ldots, \alpha_m (m \geq 0)$ such that

$$\alpha = \alpha_0 \Rightarrow \alpha_1 \Rightarrow \ldots \Rightarrow \alpha_m = \beta$$

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

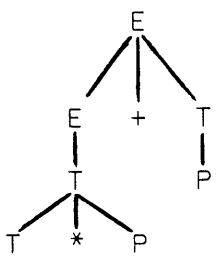
A <u>sentential form</u> is a string α such that $R \overset{*}{\Rightarrow} \alpha$. A <u>sentence</u> is a sentential form consisting entirely of terminal symbols. The <u>language defined by a grammar</u> <u>L(G)</u> is the set of its sentences. We may represent any sentential form in at least one way as a <u>derivation tree</u> (or <u>parse tree</u>) 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 <u>degree of ambiguity</u> of a sentence is the number of its distinct derivation trees. A sentence is <u>unambiguous</u> if it has degree 1 of

ambiguity. A grammar is unambiguous if each of its sentences is unambiguous.

The handle of a derivation tree is the set of terminal nodes which emanate from the left-most, lowest non-terminal node (the lower left 'T*P' in the above tree). That is, if we think of parsing as pruning branches from a derivation tree, it is the first prunable branch that we come to, scanning from left to right.

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.

We use // as an end-of-proof symbol.

III. PREVIOUS WORK

The previous work in developing efficient parsing algorithms can be split into two independent developments.

Time n Algorithms. These algorithms are concerned mainly with parsing in compilers; they all work in time proportional to n on various subclasses of grammars. Perhaps the best known of these is Floyd's operator precedence algorithm [FI 63]. It works by comparing the operators (terminals) immediately to the right and left of a point in a sentential form to determine whether or not it is the boundary of a handle. Wirth and Weber's precedence algorithm [WW 66] extends this to non-terminals as well as terminals and hints at extending it to strings as well.

Bounded context analysis [FI 64a] allows the algorithm to look at a bounded number of symbols to the right and left of a possible handle to determine whether or not it is a handle and what it is to be parsed as. Knuth's LR(k) algorithm [Kn 65] extends this in that it is able to consider the entire left part of the string (it scans from left to right) plus a bounded context on the right in order to determine the handle.

There are at least a dozen more algorithms which fall into this time n class. Many of these are referenced in Knuth's article. In attempting to compare these algorithms using our criteria, we discover that they all use time and space proportional to n, so the class of grammars which they can handle is the only distinguishing factor.

(This is not to say that it is the only distinguishing factor from a practical point of view.) By this standard, Knuth's algorithm is the best, because the LR(k) grammars include as a subset the grammars which the other three algorithms can handle and those of almost all other known time n algorithms. Recently there have been a couple of time n algorithms developed which seem to handle some non-LR(k) grammars [Ma 68] [Wi 68].

General Algorithms. These algorithms are general in that they can handle any context-free grammar. The oldest of these simply attempt to construct a possible parse tree from the input string in a straightforward way. If the algorithm starts this construction from the root and works toward the input string, it is called top down [Fl 64b]. If it starts from the input string and works toward the root it is called bottom up [Ir 61]. The basic idea of either of these algorithms is to make arbitrary choices about what the tree might look like and to back up and try again if the choices later turn out to be incorrect. Although some selectivity about these choices can be put into these algorithms, the fact remains that because of the backtracking involved, grammars can be constructed which cause even the selective versions to run in time proportional to C^n for any C [Fl 64b]. These backtracking algorithms are summarized in [GP 65].

The predictive analyzer [KO 63] is a version of top down analysis which works on grammars in standard form (the leftmost symbol on the right side of any production is terminal). However, even with a path elimination technique [Ku 65] to cut down on the time, it has been

shown that it can have exponential growth in some cases [Gr 66]. In [Ka 66] is described an algorithm which is exponential only when the number of parses of the sentence grows exponentially.

An algorithm without this exponential worst case was developed independently by Cocke [Ha 62] and Younger [Yo 67]. It requires that the grammar be put into a normal form such that every production is of the form $A \rightarrow a$ or $A \rightarrow BC$; this can be done effectively for any context-free grammar. It then provides a recognition algorithm for these grammars with an upper bound of n^3. Younger [Yo 66] and Kasami [Ka 67] are also able to recognize strings with respect to linear and meta-linear grammars (see Section VIII) in time n^2. And more recently Kasami [KT 68] has developed an algorithm which recognizes any unambiguous grammar in time n^2.

One disadvantage of these two separate developments is that none of the time n algorithms are also applicable to larger classes of grammars, and none of the general algorithms seem to do particularly well (time n) on interesting subclasses such as those treated by the time n algorithms. The algorithm presented in this paper seems to remedy this difficulty and unify the two developments.

IV. INTUITIVE EXPLANATION

The following is an informal description of the algorithm as a recognizer: It scans an input string $X_1 \ldots X_n$ 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 an instance of its right side, (2) a point in that production which shows how much of the production we have scanned, (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 can legally occur after that instance of the production. We will represent this quadruple 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:

P → a. 0

T → P. 0

T → T.*P 0

E → T. 0

E → E.+T 0

each with various k-symbol strings. Each state represents a possible parse for the beginning of the string, given that we have seen only the a. 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 '⊣' (a symbol which doesn't appear elsewhere in the grammar) at the right end of the input string.

To begin the algorithm we put the single state

$$\phi \rightarrow .R \dashv \quad \dashv^{k} \qquad 0$$

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

In general, we operate on a state set S_i as follows: we scan 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 will 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 <u>predictor</u> operation is applicable to this state because there is a non-terminal E to the right of the dot. It causes us to add one new state to S_i for each alternative of E. We put the dot at the beginning of the production in these new states since we haven't scanned any of its symbols yet. The pointer is set to i, since the state was created in S_i. The k-symbol look-ahead string in this case is ⊣, since it is after E in the original state. Thus the predictor adds to S_i all productions which we might begin to look for at X_{i+1}.

In our example, we add to S_0

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

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

We must now scan these two states. The predictor is also applicable to them. Operating on (2), it produces.

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

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

The difference is only in the look-ahead symbol. Operating on (3), it produces

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

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

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

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

$$T \rightarrow .P + 0$$

The rest of S_0 is

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

$$T \rightarrow .P * 0$$

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

$$P \rightarrow .a + 0$$

$$P \rightarrow .a * 0$$

The predictor is not applicable to the last three states. Instead the scanner is, because they have 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 we have scanned that terminal symbol.

If $X_1 = a$, then S_1 is

$$\begin{aligned} P &\rightarrow a. \dashv 0 \\ P &\rightarrow a. + 0 \\ P &\rightarrow a. * 0 \end{aligned} \qquad (6)$$

these states being added by the scanner.

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

The third operation, the completer, is applicable to these states in S_1 because the dot is at the end of the production. It compares the look-ahead string with $X_{i+1} \ldots X_{i+k}$. 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 to show that we have successfully scanned it.

If X_2 = +, then the completer is applicable to (6), and we 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 ends up with 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 on page 15. 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 [Kn 65]. In fact our algorithm approximately reduces to Knuth's algorithm on LR(k) grammars.

Grammar AE

root: E → T|E+T

T → P|T*P

P → a

input string = a+a*a

k = 1

S_0

φ → .E⊣	⊣	0
E → .E+T	⊣+	0
E → .T	⊣+	0
T → .T*P	⊣+*	0
T → .P	⊣+*	0
P → .a	⊣+*	0

S_1

P → a.	⊣+*	0
T → P.	⊣+*	0
E → T.	⊣+	0
T → T.*P	⊣+*	0
φ → E.⊣	⊣	0
E → E.+T	⊣+	0

S_2

E → E+.T	⊣+	0
T → .T*P	⊣+*	2
T → .P	⊣+*	2
P → .a	⊣+*	2

S_3

P → a.	⊣+*	2
T → P.	⊣+*	2
E → E+T.	⊣+	0
T → T.*P	⊣+*	2

S_4

T → T*.P	⊣+*	2
P → .a	⊣+*	4

S_5

P → a.	⊣+*	4
T → T*P.	⊣+*	2
E → E+T.	⊣+	0
T → T.*P	⊣+*	2
φ → E.⊣	⊣	0
E → E.+T	⊣+*	0

S_6

φ → E⊣.	⊣	0

V. THE RECOGNIZER

The following is a precise description of the recognition algorithm:

Notation: Number the productions of grammar G arbitrarily 1,...,d-1, where each production is of the form

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

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, $\dashv$ is a new terminal symbol, and k is a non-negative integer called the look-ahead parameter.

Definition. A state is a quadruple $\langle p, j, f, \alpha \rangle$ where p, j, and f are integers $(0 \leq p \leq d-1)$ $(0 \leq j \leq \bar{p})$ $(0 \leq f \leq n+1)$ and α is string consisting of k terminal symbols. A 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. $H_k(\gamma) = \{\alpha | \alpha$ 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 3 arguments $REC(G, X_1 \ldots X_n, k)$ computed as follows:

Let $X_{n+i} = \dashv$ $(1 \leq i \leq k+1)$.

Let S_i be empty $(0 \leq i \leq n+1)$.

Add $\langle 0,0,0,\dashv^k \rangle$ to S_0.

Set $i \leftarrow 0$ and go to A.

A: 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 non-final and $C_{p(j+1)}$ is non-terminal, then for each q such that $C_{p(j+1)} = D_q$, and for each $\beta \in H_k(C_{p(j+2)} \ldots C_{pp} \alpha)$ add $\langle q, 0, i, \beta \rangle$ to S_i.

(2) Completer: If s is final and $\alpha = X_{i+1} \ldots X_{i+k}$, then for each $\langle q, \ell, g, \beta \rangle \in S_f$ (after all states have been added to S_f) such that $C_{q(\ell+1)} = D_p$, add $\langle q, \ell+1, g, \beta \rangle$ to S_i.

(3) Scanner: If s is non-final 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, reject $X_1 \ldots X_n$.

If $i = n$ and $S_{i+1} = \{\langle 0, 2, 0, \dashv^k \rangle\}$, accept $X_1 \ldots X_n$.

Otherwise set $i \leftarrow i+1$ and go to A.

This is not really a complete description of the algorithm until we describe in detail how all these operations are implemented in our

model. However, we will defer that description until it is required to obtain the time bounds.

We will now prove that this algorithm is indeed a recognizer. First we introduce the idea of i-state, which is a way of stating, in an algorithm-independent way, the properties of a state constructed by the algorithm. We will use this concept in later proofs. Theorem 1 establishes the relationship between i-states and states constructed by the algorithm. Theorems 2, 3, and 4 then complete the recognizer proof.

Definition. The extensions of a string α are the strings $\alpha\beta$ for all β

Definition. The i-states of a derivation of a string $X_1 \ldots X_n$ $(1 \leq i \leq n)$ (let $X_{n+1} = \dashv$) are the triples (p, j, f), $(0 \leq p \leq d-1)$, $(0 \leq j \leq \bar{p})$, $(1 \leq f \leq i)$ such that $\exists\, \ell$ $(i \leq \ell \leq n+1)$ such that

(a) if $p \neq 0$, $R \overset{*}{\Rightarrow} X_1 \ldots X_f D_p X_{\ell+1} \ldots X_n$

(b) if $j > 0$, $C_{p1} \ldots C_{pj} \overset{*}{\Rightarrow} X_{f+1} \ldots X_i$

(c) if $j < \bar{p}$, $C_{p(j+1)} \ldots C_{p\bar{p}} \overset{*}{\Rightarrow} X_{i+1} \ldots X_\ell$

in that derivation.

Definition. The i-states of a string $X_1 \ldots X_m$ $(m \geq i)$ are the i-states of the derivations of the extensions of $X_1 \ldots X_m$.

The i-states of a string $X_1 \ldots X_m$ are roughly those states constructed by the algorithm on $X_1 \ldots X_i$ which are consistent with $X_{i+1} \ldots X_m$ being next in the input string.

Note: Unless otherwise specified, the algorithm is understood to be REC(G, $X_1 \ldots X_n$,k) in our proofs.

Theorem 1. If $\langle p, j, f, \alpha \rangle \in S_i$ and

$$X_{i+1} \ldots X_{i+k} \in H_k(C_{p(j+1)} \ldots C_{p\bar{p}} \alpha)$$

then (p, j, f) is an i-state of $X_1 \ldots X_{i+k}$.

Proof: By induction on the number of states added to any state set before $\langle p, j, f, \alpha \rangle$ is added to S_i.

Basis: $\langle 0, 0, 0, \Lambda \rangle$ is the first state added to S_0. If $X_1 \ldots X_k \in H_k(R^{k+1})$, then (0, 0, 0) is an i-state of $X_1 \ldots X_k$ as follows:

(a) p = 0, so not applicable

(b) j = 0, so not applicable

(c) $\exists\ \beta$ such that $R \overset{*}{\Rightarrow} X_1 \ldots X_k \beta$ since $X_1 \ldots X_k \in H_k(R^{k+1})$.

Induction Step:

(1) $\langle q, 0, i, \beta \rangle$ is added to S_i by the predictor from state $\langle p, j, f, \alpha \rangle$ and

$$X_{i+1} \ldots X_{i+k} \in H_k(C_{q1} \ldots C_{q\bar{q}} \beta) = H_k(D_q \beta)$$

Then

$$X_{i+1} \ldots X_{i+k} \in H_k(C_{p(j+1)} C_{p(j+2)} \ldots C_{p\bar{p}} \alpha)$$

since $D_q = C_{p(j+1)}$ and $\beta \in H_k(C_{p(j+2)} \ldots C_{p\bar{p}} \alpha)$. So by inductive

hypothesis (p, j, f) is an i-state of $X_1 \ldots X_{i+k}$. Therefore $\exists$ an extension $X_1 \ldots X_n$ and an ℓ such that

$$R \overset{*}{\Rightarrow} X_1 \ldots X_f D_p X_{\ell+1} \ldots X_n$$

$$C_{p1} \ldots C_{pj} \overset{*}{\Rightarrow} X_{f+1} \ldots X_i$$

and $$C_{p(j+1)} \ldots C_{p\bar{p}} \overset{*}{\Rightarrow} X_{i+1} \ldots X_\ell$$

So there is an ℓ' ($i \leq \ell' \leq \ell$) such that

$$C_{p(j+2)} \ldots C_{p\bar{p}} \overset{*}{\Rightarrow} X_{\ell'+1} \ldots X_\ell$$

If $\exists$ γ such that

$$C_{q1} \ldots C_{q\bar{q}} \overset{*}{\Rightarrow} X_{i+1} \ldots X_{i+k}\gamma$$

(q, 0, i) is an i-state of a derivation of $X_i \ldots X_{i+k}\gamma X_{\ell'+1} \ldots X_n$ as follows:

(a) $D_p \overset{*}{\Rightarrow} X_{f+1} \ldots X_i D_q X_{\ell'+1} \ldots X_\ell$, so

$$R \overset{*}{\Rightarrow} X_1 \ldots X_i D_q X_{\ell'+1} \ldots X_n$$

(b) $j = 0$, so not applicable

(c) $C_{q1} \ldots C_{q\bar{q}} \overset{*}{\Rightarrow} X_{i+1} \ldots X_{i+k}\gamma$

If $\exists\ k' < k$ such that $C_{q1}\ldots C_{q\bar{q}} \overset{*}{\Rightarrow} X_{i+1}\ldots X_{i+k'}$, then $(q, 0, i)$ is an i-state of a derivation of $X_1\ldots X_{i+k'}X_{\ell'+1}\ldots X_n$ $(i+k' = \ell')$ as follows:

(a) and (b) are the same as above.

(c) $C_{q1}\ldots C_{q\bar{q}} \overset{*}{\Rightarrow} X_{i+1}\ldots X_{i+k'}$

(2) $\langle q, j+1, g, \beta\rangle$ is added to S_i by the completer from $\langle p, \bar{p}, f, \alpha\rangle \in S_i$ and $\langle q, j, g, \beta\rangle \in S_f$. Then $\alpha = X_{i+1}\ldots X_{i+k}$, so by inductive hypothesis, $(p, \bar{p}, f)$ is an i-state of $X_1\ldots X_{i+k}$. So

$$C_{p1}\ldots C_{p\bar{p}} \overset{*}{\Rightarrow} X_{f+1}\ldots X_i$$

Also, since

$$X_{i+1}\ldots X_{i+k} \in H_k(C_{q(j+2)}\ldots C_{q\bar{q}}\ \beta)$$

$$X_{f+1}\ldots X_i X_{i+1}\ldots X_{i+k} \in H_{i-f+k}\ (C_{p1}\ldots C_{p\bar{p}}\ C_{q(j+2)}\ldots C_{q\bar{q}}\ \beta)$$

which equals $H_{i-f+k}\ (C_{q(j+1)}\ldots C_{q\bar{q}}\ \beta)$ since $D_p = C_{q(j+1)}$. So $X_{f+1}\ldots X_{f+k} \in H_k(C_{q(j+1)}\ldots C_{q\bar{q}}\ \beta)$. So by inductive hypothesis (q, j, g) is an f-state of $X_1\ldots X_{f+k}$. Therefore $\exists$ an extension $X_1\ldots X_n$ and an ℓ such that

$$R \overset{*}{\Rightarrow} X_1\ldots X_g D_q X_{\ell+1}\ldots X_n$$

$$C_{q1}\ldots C_{qj} \overset{*}{\Rightarrow} X_{g+1}\ldots X_f$$

and

$$C_{p(j+1)}\cdots C_{p\bar{p}} \overset{*}{\Rightarrow} X_{f+1}\cdots X_{\ell}$$

If $\exists\ \gamma$ such that

$$C_{p(j+2)}\cdots C_{p\bar{p}} \overset{*}{\Rightarrow} X_{i+1}\cdots X_{i+k}\ \gamma$$

$(q, j+1, g)$ is an i-state of a derivation of $X_1\cdots X_{i+k}\ \gamma\ X_{\ell+1}\cdots X_n$ as follows:

(a) $R \overset{*}{\Rightarrow} X_1\cdots X_g D_q X_{\ell+1}\cdots X_n$

(b) $C_{q1}\cdots C_{qj}\ C_{p1}\cdots C_{p\bar{p}} \overset{*}{\Rightarrow} X_{g+1}\cdots X_f X_{f+1}\cdots X_i$ so

$C_{q1}\cdots C_{q(j+1)} \overset{*}{\Rightarrow} X_{g+1}\cdots X_i$, since $C_{q(j+1)} = D_p$

(c) $C_{p(j+2)}\cdots C_{p\bar{p}} \overset{*}{\Rightarrow} X_{i+1}\cdots X_{i+k}\ \gamma$

If $\exists\ k' < k$ such that $C_{p(j+2)}\cdots C_{p\bar{p}} \overset{*}{\Rightarrow} X_{i+1}\cdots X_{i+k'}$, then $(q, j+1, g)$ is an i-state of a derivation of $X_1\cdots X_{i+k'}X_{\ell+1}\cdots X_n$ $(i+k' = \ell)$ as follows:

(a) and (b) are the same as above.

(c) $C_{p(j+2)}\cdots C_{p\bar{p}} \overset{*}{\Rightarrow} X_{i+1}\cdots X_{i+k'}$

(3) $\langle p, j+1, f, \beta\rangle$ is added to S_i by the scanner from $\langle p, j, f, \beta\rangle \in S_{i-1}$.

$$X_{i+1}\cdots X_{i+k} \in H_k(C_{p(j+2)}\cdots C_{p\bar{p}}\ \beta)$$

and $C_{p(j+1)} = X_i$, so

$$X_i \ldots X_{i+k} \in H_{k+1}(C_{p(j+1)} \ldots C_{p\bar{p}}\ \beta)$$

so

$$X_i \ldots X_{i+k-1} \in H_k(C_{p(j+1)} \ldots C_{p\bar{p}}\ \beta)$$

Therefore, by inductive hypothesis, (p, j, f) is an (i-1)-state of $X_1 \ldots X_{i+k-1}$. Therefore $\exists$ an extension $X_1 \ldots X_n$ and an ℓ such that

$$R \overset{*}{\Rightarrow} X_1 \ldots X_f D_p X_{\ell+1} \ldots X_n$$

$$C_{p1} \ldots C_{pj} \overset{*}{\Rightarrow} X_{f+1} \ldots X_{i-1}$$

and $$C_{p(j+1)} \ldots C_{p\bar{p}} \overset{*}{\Rightarrow} X_i \ldots X_\ell$$

so (p, j+1, f) is an i-state of a derivation of $X_1 \ldots X_n$ as follows:

(a) $R \overset{*}{\Rightarrow} X_1 \ldots X_f D_p X_{\ell+1} \ldots X_n$

(b) $C_{p1} \ldots C_{pj} C_{p(j+1)} \overset{*}{\Rightarrow} X_{f+1} \ldots X_{i-1} X_i$ Since $C_{p(j+1)} = X_i$

(c) $C_{p(j+2)} \ldots C_{p\bar{p}} \overset{*}{\Rightarrow} X_{i+1} \ldots X_\ell$. //

Theorem 2. If $X_1 \ldots X_n$ is accepted by the algorithm, then it is a sentence.

Proof: If $X_1 \ldots X_n$ is accepted by the algorithm, then $S_{n+1} = \{<0, 2, 0, \dashv^k>\}$. And by theorem 1, since $\dashv^k \in H_k(\dashv^k)$, (0, 2, 0)

is an (n+1)-state of $X_1 \ldots X_n \dashv^{k+1}$. So by (b) in the definition of i-state,

$$R \dashv \overset{*}{\Rightarrow} X_1 \ldots X_n \dashv$$

or $R \overset{*}{\Rightarrow} X_1 \ldots X_n$, so it is a sentence. //

Theorem 3. If $\langle p, j, f, \alpha \rangle \in S_i$, $C_{p(j+1)} \overset{*}{\Rightarrow} X_{i+1} \ldots X_\ell$, and $X_{\ell+1} \ldots X_{\ell+k} \in H_k(C_{p(j+2)} \ldots C_{p\bar{p}}\ \alpha)$, then $\langle p, j+1, f, \alpha \rangle \in S_\ell$.

Proof: By induction on m (see page 4) in the definition of $\overset{*}{\Rightarrow}$ in $C_{p(j+1)} \overset{*}{\Rightarrow} X_{i+1} \ldots X_\ell$.

Basis: If m = 0, then $\ell = i+1$ and $C_{p(j+1)} = X_{i+1}$, so $\langle p, j+1, f, \alpha \rangle$ is added to S_{i+1} by the scanner.

Induction Step: if m > 0, $\exists$ q such that $C_{p(j+1)} = D_q \rightarrow C_{q1} \ldots C_{q\bar{q}}$ and $\exists\ t_0 \leq t_1 \leq \ldots \leq t_{\bar{q}}$ such that $t_0 = i$, $t_{\bar{q}} = \ell$ and

$$C_{q1} \overset{*}{\Rightarrow} X_{t_0+1} \ldots X_{t_1}, \ldots, C_{q\bar{q}} \overset{*}{\Rightarrow} X_{t_{(\bar{q}-1)}+1} \ldots X_{t_{\bar{q}}}$$

Because of the predictor acting on $\langle p, j, f, \alpha \rangle$ in S_i, $\langle q, 0, i, X_{\ell+1} \ldots X_{\ell+k} \rangle$ will be added to S_i since $X_{\ell+1} \ldots X_{\ell+k} \in H_k(C_{p(j+2)} \ldots C_{p\bar{p}}\ \alpha)$. And by inductive hypothesis we know that if $\langle q, r, i, X_{\ell+1} \ldots X_{\ell+k} \rangle$ is in S_{t_r}, then $\langle q, r+1, i, X_{\ell+1} \ldots X_{\ell+k} \rangle$ will be in $S_{t_{(r+1)}}$ $(0 \leq r \leq q-1)$ since

$$C_{q(r+1)} \overset{*}{\Rightarrow} X_{t_r+1} \cdots X_{t_{(r+1)}}$$

and $$X_{t_{(r+1)}+1} \cdots X_{t_{(r+1)}+k} \in H_k(C_{q(r+2)} \cdots C_{q\bar{q}}\ X_{\ell+1} \cdots X_{\ell+k})$$

So by induction on r, $S_{t_{\bar{q}}} = S_\ell$ contains $\langle q, \bar{q}, i, X_{\ell+1} \cdots X_{\ell+k} \rangle$. And the completer, acting on this state, adds $\langle p, j+1, f, \alpha \rangle$ to S_ℓ. //

Theorem 4. If $X_1 \cdots X_n$ is a sentence, then it is accepted by the algorithm.

Proof: S_0 contains $\langle 0, 0, 0, \dashv^k \rangle$, $R \overset{*}{\Rightarrow} X_1 \cdots X_n$, and $\dashv^k \in H_k(\dashv^{k+1})$. So by theorem 3, S_n contains $\langle 0, 1, 0, \dashv^k \rangle$. And since $\dashv = C_{02}$, the scanner adds $\langle 0, 2, 0, \dashv^k \rangle$ to S_{n+1}. This is the only state that S_{n+1} can contain since $\dashv$ appears nowhere else in the grammar. So $X_1 \cdots X_n$ is accepted. //

Theorems 2 and 4 together show that the algorithm is a recognizer for all context-free grammars. Notice that our recognizer proofs require no restrictions at all on the grammar. That is, unlike most algorithms, ours handles correctly circular grammars, disconnected grammars, grammars which generate strings with an infinite number of parses, even grammars which generate the empty language.

VI. RECOGNITION TIME

Now we will try to characterize precisely the speed of the algorithm. First we describe its implementation. The formal model of a computer which we are using has been chosen purposely to reflect as much as possible the random access properties of real machines. Therefore it is not restricted to the square-at-a-time movement of a Turing machine.

We will not describe our model in detail here; that will be done in Section XV. Instead we will describe the implementation as if it were done on a real machine. Our implementation description assumes a knowledge of basic list processing techniques.

Implementation

(1) For each non-terminal, 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 scanned 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 ($0 \leq f \leq i$) 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 n. 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 non-terminal 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 nil state ($A \rightarrow .\ \alpha\ i$) 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. This does not change the time bounds, however.

Now we will derive an upper bound on the recognition time. First some definitions.

Definition. Let m-1 be the maximum $\bar{p}$ ($0 \leq p \leq d-1$).

Let a be the maximum number of alternatives of any non-terminal.

Let t be the number of terminals in G.

Let u be the number of non-terminals in G.

Let v = t+u be the number of terminals and non-terminals in G.

Let C be any constant independent of the length of the string.

Definition. A step is an algorithm which takes a fixed number of basic operations in our model independent of the size of the grammar or of the string being recognized.

The following theorem establishes the fact that our algorithm is an n^3 recognizer in general.

Theorem 5. There is a bound of the form $Cn^3+O(n^2)$ on the number of steps required to compute REC(G, $X_1 \ldots X_n$, k) for any k.

Proof: (a) There are at most $dmit^k$ states in any state set S_{i-1}, so it takes at most $dmit^k$ steps to scan them. The scanner adds just one state for each state to which it is applicable. The predictor adds at most at^k states for each state to which it is applicable. (The members of H_k can be calculated independent of the recognition process.) To test each state added to see if it is already in the state set costs at most dmt^k. So scanning the states in S_i plus the scanner and predictor operations takes at most $(at^k)\ (dmit^k)\ (dmt^k)$ steps.

(b) The completer performs more than one step at a particular state $\langle p, j, f, \alpha \rangle$ only if $j = \bar{p}$ and $\alpha = X_{i+1} \ldots X_{i+k}$. For each of these states it adds at most $dmft^k \leq dmit^k$ states, one for each possible state in S_f. It takes at most dmt^k steps to test each state added. So the completer performs at most $(di)\ (dmit^k)\ (dmt^k)$ steps.

Summing over all state sets, we get

$$\sum_{i=1}^{n+1} (admit^{2k} + d^2mi^2t^k)\ (dmt^k)$$

$$= [admt^{2k} \sum_{i=1}^{n+1} i + d^2mt^k \sum_{i=1}^{n+1} i^2]\ (dmt^k)$$

$$= [admt^{2k} \frac{(n+1)(n+2)}{2} + d^2mt^k \frac{(n+1)(n+2)(2n+3)}{6}]\ (dmt^k)$$

$$\approx d^3m^2t^{2k}n^3\ //$$

Since this bound holds for any k (including k = 0) we could do without the look-ahead here. It will be more useful in obtaining time n bounds (Section IX). The n^3 bound is no better than that obtained by Younger [Yo 67], but the algorithm is better in that it does not require the grammar to be put into any special form (Younger's requires normal form).

VII. BOUNDED DIRECT AMBIGUITY

We next ask the question: What classes of grammars can be done in time n^2 by the algorithm? We will define one such class in terms of the amount of ambiguity of the grammar.

Definition. The degree of direct ambiguity of a non-terminal A with respect to a string $X_1 \ldots X_n$ is the number of distinct pairs $\langle p, t \rangle$, where p is a production such that $D_p = A$, and t is an $(\bar{p}+1)$-tuple $\langle q_1, \ldots, q_{\bar{p}+1} \rangle$ such that $q_1 = 0$, $q_{\bar{p}+1} = n$ and

$$C_{pj} \overset{*}{\Rightarrow} X_{q_j+1} \ldots X_{q_{(j+1)}} \qquad (1 \leq j \leq \bar{p})$$

Intuitively, the degree of direct ambiguity is the number of "top-level parses". That is, instead of considering all the different parses of $X_1 \ldots X_n$ as A, we consider only the first production of each parse, say $A \rightarrow BC$, and the substrings of $X_1 \ldots X_n$ which the symbols in this production generate, $B \overset{*}{\Rightarrow} X_1 \ldots X_m$, $C \overset{*}{\Rightarrow} X_{m+1} \ldots X_n$. Each such distinct pair $\langle p, (0, m, n) \rangle$ increases the degree of direct ambiguity by 1.

We know the following concept:

Definition. A grammar has bounded ambiguity if there exists a bound for the grammar on the degree of ambiguity of any of its sentences.

The analogy of this definition for direct ambiguity is as follows:

VIII. TIME n^2

Now we attack the n^2 question. The completer is the only thing which forces us to use n^2 steps for each symbol we scan, making the whole thing n^3. So we ask, in what cases does this operation involve only n steps instead of n^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 (proportional to i) steps to do the operation.

In the case that the grammar is unambiguous, we can show that each such state gets added in only one way, but using the concept of bounded direct ambiguity, we can get an even more powerful result. Lemma 1 shows that the number of ways that a state can be added is proportional to the degree of direct ambiguity. Using this, we prove theorem 6, which gives us the n^2 result for all BDA grammars.

Lemma 1. If b is a bound on the degree of direct ambiguity of D_q with respect to any string, then any state $\langle q, j+1, f, \alpha \rangle$ which is added to S_i by the completer can be added in at most ba different ways.

Proof: 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

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

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

$$\langle q, j, f, \alpha\rangle \in S_{f_1} \text{ and } S_{f_2}$$

$$D_{p_1} = C_{q(j+1)} = D_{p_2}$$

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.

Consider the degree of direct ambiguity of D_q with respect to $X_{f+1}\ldots X_i\beta$, where $C_{q(j+2)}\ldots C_{q\bar{q}} \overset{*}{\Rightarrow} \beta$. It must be at least 2, since (in the definition of degree of direct ambiguity) f_1-f and f_2-f are the (j+1)th members of two ($\bar{q}$+1)-tuples which partition $X_{f+1}\ldots X_i\beta$. And each other distinct f_i indicates the existence of another distinct n-tuple. So there can be at most b such integers, since b is a bound on the direct ambiguity.

If $f_1 = f_2$, there can be at most a distinct productions p_i since each is an alternative of $C_{q(j+1)}$. Therefore $\langle q, j+1, f, \alpha\rangle$ can be added in at most ba ways by the completer. //

Using this lemma, we can obtain the n^2 result:

Theorem 6. There is an upper bound at the form $Cn^2+O(n)$ on the number of steps in computing REC(G, $X_1\ldots X_n$, k) for any k and any BDA grammar.

Proof: Let G have degree b of direct ambiguity.

(a) of theorem 5 still holds here.

(b) At most $dmit^k$ states can be added by the completer. By lemma 1, each can be added in at most ba different ways. And it costs at most dmt^k to test each one. This gives us $(badmit^k)(dmt^k)$ steps required for the completer.

Summing for i = 1,...n+1 produces

$$\sum_{i=1}^{n+1} (admit^{2k}+badmit^k)(dmt^k)$$

$$\leq (b+1)(admt^{2k})(dmt^k) \sum_{i=1}^{n+1} i = (b+1)(admt^{2k})(dmt^k)(n+1)(n+2)/2$$

$$\approx bad^2m^2t^{3k}n^2$$

We will now show that our algorithm also achieves Younger's n^2 results for linear and meta-linear grammars [Yo 66].

Definition. A production $A \rightarrow \alpha$ is linear if α contains at most one non-terminal symbol.

Definition. A grammar is linear if all its productions are linear.

Definition. A grammar is meta-linear if no production $A \rightarrow \alpha$ has the root R in α, and if all productions $A \rightarrow \alpha$ where $A \neq R$ are linear.

Theorem 7. A linear grammar has bounded direct ambiguity.

Proof: A non-terminal A has degree of direct ambiguity of at most a (the maximum number of alternatives of any non-terminal) with respect to any string. For each production $A \rightarrow C_{p1}...C_{p\bar{p}}$ where C_{pj} is the non-terminal ($1 \leq j \leq p$) and each string $X_1...X_n$, the only $(\bar{p}+1)$-tuple that partitions the string is $<0, 1,...j-1, j+n-\bar{p},...n-1, n>$. //

Combined with Theorem 6, this gives us the n^2 result for linear grammars.

Theorem 8. There is a bound of the form $Cn^2+O(n)$ on the number of steps in computing REC(G, $X_1...X_n$, k) for any k and any meta-linear grammar G.

Proof: (a) of theorem 5 holds here.

(b) of theorem 6 holds also for all the linear productions in G, by theorem 7. However, since R cannot occur on the right side of any production, all states $<p, j, f, \alpha>$ such that $D_p = R$ will have been created originally (i.e., when $j = 0$) in S_0. Therefore $f = 0$ for all states with non-linear productions, and therefore there can be at most dmt^k of them in S_{i-1}. Each of these states can be added a full ai different ways by the completer (see proof of lemma 1). So we must add a third term, $admit^k$, to the calculations of theorem 6. This however also results in an n^2 term, so we still have the n^2 result. //

So we have achieved the n^2 results for both unambiguous and meta-linear grammars with one algorithm. In addition, our algorithm does not require a normal form grammar while both Kasami's and Cocke's do.

To illustrate these results we will run some example grammars. Grammar BK (page 39) is obtained by simplifying grammar K. It has the same ambiguity properties as K (but the language doesn't); that is, it has BDA, but unbounded ambiguity. Notice that state

$$K \rightarrow KJ_{\bullet}\,1x \quad 0$$

in S_i gets added twice by the completer. (This is signified by the superscript on the 0.) This is because the grammar has degree 2 of direct ambiguity. Actually this grammar not only doesn't require more than time n^2, it requires only time n, since all the state sets after S_1 are more or less the same.

Grammar BP (page 40) is obtained in a similar way from grammar P. It has unbounded direct ambiguity. This is illustrated by the increasing superscript on state

$$P \rightarrow MN. \quad 0$$

The superscript will continue to be i+1 in S_i. However, this is not enough to make this grammar require n^3, since it is meta-linear. This fact means that the state which gets added $\sim$ i times, will be unique (or there will be a bounded number of them) as it is here, so the time is still n^2.

Grammar UBDA (page 42) is not so nice. It has unbounded direct ambiguity and no saving grace. One can tell by the looks of the exponents on states

$$A \to AA. \quad 2$$

$$A \to AA. \quad 1^2$$

$$A \to AA. \quad 0^3$$

in S_4 that there are $\sim i$ states, each of which is added $\sim i$ times. So this really takes time n^3.

Grammar BK

root: $K \to \wedge \mid KJ$

$J \to F \mid I$

$F \to x$

$I \to x$

sentences: x^n $(n \geq 0)$

REC(BK, x^n, I):

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

$S_i (2 \leq i \leq n)$	$F \to x.$	$\dashv x$	i-1
	$F \to x.$	$\dashv x$	i-1
	$J \to F.$	$\dashv x$	i-1
	$J \to I.$	$\dashv x$	i-1
	$K \to KJ.$	$\dashv x$	0^2
	$K \to K.J$	$\dashv x$	0
	$\phi \to K.\dashv$		0
	$J \to .F$	$\dashv x$	i
	$J \to .I$	$\dashv x$	i
	$F \to .x$	$\dashv x$	i
	$I \to .x$	$\dashv x$	i
S_{n+1}	$\phi \to K\dashv.$	$\dashv$	0

Grammar BP

root: $P \rightarrow MN$

$M \rightarrow \Lambda \mid Ma$

$N \rightarrow \Lambda \mid Na$

sentences: a^n $(n \geq 0)$

REC(BP, aa, 0):

S_0	$\phi \rightarrow .P\dashv$	0
	$P \rightarrow .MN$	0
	$M \rightarrow .$	0
	$M \rightarrow .Ma$	0
	$P \rightarrow M.N$	0
	$M \rightarrow M.a$	0
	$N \rightarrow .$	0
	$N \rightarrow .Na$	0
	$P \rightarrow MN.$	0
	$N \rightarrow N.a$	0
	$\phi \rightarrow P.\dashv$	0

S_1	$M \rightarrow Ma.$	0
	$N \rightarrow Na.$	0
	$P \rightarrow M.N$	0
	$M \rightarrow M.a$	0
	$P \rightarrow MN.$	0^2
	$N \rightarrow N.a$	0
	$N \rightarrow .$	1
	$N \rightarrow .Na$	1
	$\phi \rightarrow P.\dashv$	0
	$N \rightarrow N.a$	1

S_2	M → Ma.	0
	N → Na.	0
	N → Na.	1
	P → M.N	0
	M → M.a	0
	P → MN.	0^3
	N → N.a	0
	N → N.a	1
	N → .	2
	N → .Na	2
	ϕ → P.⊣	0
	N → N.a	2
S_3	ϕ → P⊣.	0

Grammar UBDA

root: $A \rightarrow x \mid AA$

sentences: x^n $(n \geq 1)$

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

IX. TIME n

We would now like to characterize the class of grammars which the algorithm will do in time n. We notice that for some grammars the numbers 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 will call the latter grammars "bounded state" grammars, and we will show that they can be done in time n.

We can define "bounded state" independently of the algorithm using the concept of i-state which we defined earlier (Section V).

Definition. A grammar is bounded state with look-ahead k, BS(k), if $\exists$ a bound on the number of different i-states of any string $X_1 \ldots X_{i+k}$ $(0 \leq i \leq n)$.

Theorem 9 shows that our definition of bounded state really means what it says.

Theorem 9. If b is a bound on the number of i-states of any string $X_1 \ldots X_{i+k}$, then dmb^{m+k} is a bound on the number of states in S_i.

Proof: Assume that there are $dmb^{m+k}+1$ states in S_i. Then $\exists$ $p, j, \alpha, f_1, \ldots, f_{b^m+1}$ such that the f's are distinct and

$$\langle p, j, f_1, \alpha\rangle, \ldots, \langle p, j, f_{b^m+1}, \alpha\rangle \in S_i$$

If $j = 0$, then $f_t = i$ for all t $(1 \leq t \leq b^m+1)$. So $j \neq 0$. In fact, if $C_{p1} \ldots C_{pj}$ is a terminal string, then $f_t = i-j$ for all t $(1 \leq t \leq b^m+1)$.

So one of $C_{p1},\ldots,C_{pj}$ must be non-terminal. Let h be the largest integer $h \leq j$ such that C_{ph} is non-terminal. We know that $\langle p, h, f_t, \alpha\rangle \in S_g$ $(1 \leq t \leq b^m+1)$ where $g = i-(j-h)$. So these states were added to S_g by the completer acting on some number r of final states

$$\langle q, \bar{q}, \ell_s, X_{g+1}\ldots X_{g+k}\rangle \in S_g \quad (0 \leq \ell_s \leq g) \quad (1 \leq s \leq r) \text{ where } D_q = C_{ph}$$

The look-ahead string for these states must be $X_{g+1}\ldots X_{g+k}$ for the completer to be applicable.

By theorem 1, $(q, \bar{q}, \ell_s)$ is a g-state of $X_1\ldots X_{g+k}$ $(1 \leq s \leq r)$, so $r \leq b$ since there can be at most b of them. But since there are b^m+1 states added by the completer from these r states, at least one of the final states must have $b^{m-1}+1$ parents. That is for some s, S_{ℓ_s} contains $(p, h-1, f_t, \alpha)$ for $b^{m-1}+1$ different t's. We now apply the above argument again as many times as there are non-terminals in $C_{p1}\ldots C_{ph}$, reducing the power of b by one each time. However, there can be at most m non-terminals, so we end up with at least b^0+1 (or 2) states $\langle p, h', f_t, \alpha\rangle$ in some state set $S_{g'}$ $(0 \leq h' \leq h)$ $(0 \leq g' \leq g)$ where $C_{p1}\ldots C_{ph'}$ is terminal. But then the two f_t's must both be g-h'. But the f's are all distinct. Contradiction. //

From this, we easily obtain the time n result for bounded state grammars.

Theorem 10. There is a bound of the form Cn on the number of steps required to compute REC(G, $X_1\ldots X_n$, k) for any BS(k) grammar G.

Proof: By theorem 9, $dmb^{m}t^{k}$ is a bound on the number of states in any state set S_i. This changes (a) of theorem 5 to $(at^{k})(dmb^{m}t^{k})(dmt^{k})$. In (b), we have at worst $(dmb^{m}t^{k})^{2}(dmt^{k})$. Since both of these are constant with respect to n, we get

$$\sum_{i=1}^{n} C = Cn \quad //$$

Now, how does the bounded state grammar concept compare with other time n classes of grammars, especially the LR(k) grammars? It turns out that almost all LR(k) grammars are BS(0) grammars. In fact all except certain right recursive LR(k) grammars are BS(0). (A right recursive grammar is one in which $\exists$ a non-terminal A such that $A \overset{*}{\Rightarrow} \alpha A$ for some A.) However, even very simple right recursive grammars are not BS(k) for any k.

The run on grammar RR with k = 1 illustrates this (page 48). Notice that all state sets have 4 or less states except S_3 which has 7. If the input string were x^n, all states would have 4 or less except S_n which would have 4+n. Grammar RR is, in fact, LR(1), but there is no bound on the size of S_n, so it is not BS(1). One can see that no finite k would remedy this.

However, notice that each of the extra states in S_n actually represents a step in the derivation of x^n. So since the derivation can have at most $\sim$ n steps, the total number of "extra states" is $\sim$ n, and therefore the number of states is $\sim$ n even though there is no bound on the number in a single state set. Because of this, the algorithm will recognize with respect to RR in time n if k = 1. We will prove that

Grammar RR

$A \rightarrow x \mid xA$

sentences: x^n $(n \geq 1)$

REC(RR,xxx,0):

S_0	$\phi \rightarrow .A\dashv$	0
	$A \rightarrow .x$	0
	$A \rightarrow .xA$	0
S_1	$A \rightarrow x.$	0
	$A \rightarrow x.A$	0
	$\phi \rightarrow A.\dashv$	0
	$A \rightarrow .x$	1
	$A \rightarrow .xA$	1
S_2	$A \rightarrow x.$	1
	$A \rightarrow x.A$	1
	$A \rightarrow xA.$	0
	$A \rightarrow .x$	2
	$A \rightarrow .xA$	2
	$\phi \rightarrow A.\dashv$	0
S_3	$A \rightarrow x.$	2
	$A \rightarrow x.A$	2
	$A \rightarrow xA.$	1
	$A \rightarrow .x$	3
	$A \rightarrow .xA$	3
	$A \rightarrow xA.$	0
	$\phi \rightarrow A.\dashv$	0
S_4	$\phi \rightarrow A\dashv.$	0

REC(RR,xxx,1):

S_0	$\phi \rightarrow .A\dashv$	$\dashv$	0
	$A \rightarrow .x$	$\dashv$	0
	$A \rightarrow .xA$	$\dashv$	0
S_1	$A \rightarrow x.$	$\dashv$	0
	$A \rightarrow x.A$	$\dashv$	0
	$A \rightarrow .x$	$\dashv$	1
	$A \rightarrow .xA$	$\dashv$	1
S_2	$A \rightarrow x.$	$\dashv$	1
	$A \rightarrow x.A$	$\dashv$	1
	$A \rightarrow .x$	$\dashv$	2
	$A \rightarrow .xA$	$\dashv$	2
S_3	$A \rightarrow x.$	$\dashv$	2
	$A \rightarrow x.A$	$\dashv$	2
	$A \rightarrow xA.$	$\dashv$	1
	$A \rightarrow .x$	$\dashv$	3
	$A \rightarrow .xA$	$\dashv$	3
	$A \rightarrow xA.$	$\dashv$	0
	$\phi \rightarrow A.\dashv$	$\dashv$	0
S_4	$\phi \rightarrow A\dashv.$	$\dashv$	0

this holds true for all LR(k) grammars. That is, even if they aren't bounded state, they are time n if a look-ahead of k us used.

First we rephrase Knuth's definition of LR(k) [Kn 65].

Definition. For an unambiguous grammar, (m, p) is a handle of $A_1 \ldots A_n$ if

(a) $R \overset{*}{\Rightarrow} A_1 \ldots A_r D_p A_{m+1} \ldots A_n$ $(r = m - \bar{p})$

(b) $D_p \rightarrow A_{r+1} \ldots A_m$ $(r = m - \bar{p})$

and (c) $\nexists$ $\alpha \overset{*}{\underset{\neq}{\Rightarrow}} A_1 \ldots A_{m-1}$ such that $R \overset{*}{\Rightarrow} \alpha A_m \ldots A_n$

Definition. A grammar is LR(k) as follows:

Let $\alpha = X_1 \ldots X_{i+k} \alpha'$

$\beta = X_1 \ldots X_{i+k} \beta'$

where $X_{i+1} \ldots X_{i+k}$, α', β' are terminal.

If (i, p) is a handle of α and (j, q) is a handle of β, then $i = j$ and $p = q$.

Now we define the idea of a "useful" string. Given $X_1 \ldots X_{i+k}$, $A_1 \ldots A_m$ is useful if it is the leftmost portion (generates $X_1 \ldots X_i$) of a sentential form of $X_1 \ldots X_n$, and if it is the topmost such string in the derivation. Lemma 2 proves that this string is unique in LR(k) grammars. Note that it is also important that there is only one derivation of $X_1 \ldots X_i$ from $A_1 \ldots A_m$.

Definition. A string $A_1 \ldots A_m$ is useful if

(1) $R \overset{*}{\Rightarrow} A_1 \ldots A_m X_{i+1} \ldots X_{i+k} \delta$ for some δ

(2) $A_1 \ldots A_m \overset{*}{\Rightarrow} X_1 \ldots X_i$ in only one way

(3) $\nexists\ \gamma \overset{*}{\underset{\neq}{\Rightarrow}} A_1 \ldots A_m$ such that $R \overset{*}{\Rightarrow} \gamma X_{i+1} \ldots X_{i+k} \delta$ for some δ

Lemma 2. If G is LR(k), then for each string $X_1 \ldots X_{i+k}$, $\exists$ at most one useful string $A_1 \ldots A_m$.

Proof: By induction on i.

Basis: If i = 0, then m = 0 and $A_1 \ldots A_m = \Lambda$.

Induction Step: Assume it is true for i. Then the unique useful string for $X_1 \ldots X_{i+k}$, $A_1 \ldots A_m$, is the only string such that either (1) $A_1 \ldots A_m X_{i+1} \ldots X_{i+k+1} \delta$ is a sentential form which can have a handle at m+1 for some δ, or (2) $A_1 \ldots A_m X_{i+1}$ is a useful string for $X_1 \ldots X_{i+k+1}$. Let $X_{i+1} = A_{m+1}$.

A: If (m+1, p) is not a handle of $A_1 \ldots A_{m+1} X_{i+2} \ldots X_{i+k+1} \delta$ for any p, δ, then $A_1 \ldots A_{m+1}$ is the only useful string for $X_1 \ldots X_{i+k+1}$.

If (m+1, p) is a handle of $A_1 \ldots A_{m+1} X_{i+2} \ldots X_{i+k+1} \delta$ for some p, δ, then it is the handle for all δ since the grammar is LR(k). Therefore $A_1 \ldots A_r D_p$ (where $r = m+1-\bar{p}$) is so far the topmost string such that $A_1 \ldots A_r D_p \Rightarrow A_1 \ldots A_{m+1}$, and $A_1 \ldots A_r D_p X_{i+2} \ldots X_{i+k+1} \delta$ is a sentential form which can have a handle at r+1. Let $A_{r+1} = D_p$, let m = r. Repeat the above process starting at A until (m+1, p)

is not a handle. Notice that there is only one derivation of the new $A_1 \ldots A_{m+1}$ from the old $A_1 \ldots A_m X_{i+1}$, so by inductive hypothesis there is only one derivation $A_1 \ldots A_{m+1} \overset{*}{\Rightarrow} X_1 \ldots X_{i+1}$. //

Since the "extra states" above the bound on the size of a state set in LR(k) grammars are added by the completer, we will concentrate on this operation in our proofs. Lemmas 3 and 4 are used in several places later.

Lemma 3. If G is an LR(k) grammar, then $\exists$ at most one state $\langle q, \bar{q}, \ell, X_{i+1} \ldots X_{i+k} \rangle \in S_i$ which has not been added by the completer.

Proof: By theorem 1, $(q, \bar{q}, \ell)$ is an i-state of $X_1 \ldots X_{i+k}$, so

$$D_q \rightarrow C_{q1} \ldots C_{q\bar{q}} \overset{*}{\Rightarrow} X_{\ell+1} \ldots X_i$$

This derivation then is part of the unique derivation $A_1 \ldots A_m \Rightarrow X_1 \ldots X_i$ of the useful string $A_1 \ldots A_m$ of $X_1 \ldots X_{i+k}$ (because of (3) in the definition of useful, and lemma 2). But since this state was not added by the completer, it must have been added by the scanner, so $C_{q\bar{q}} = X_i$. If there is another such final state $\langle q', \bar{q}', \ell', X_{i+1} \ldots X_{i+k} \rangle$, it is also part of the same derivation and has $C_{q'\bar{q}'} = X_i$ for the same reasons. Thus X_i is parsed in two different ways as follows: If $q \neq q'$, then X_i is derived by different productions. If $q = \bar{q}$, but $\ell \neq \ell'$ then the production which X_i is derived from generates 2 different length strings. In any case, if the states are not identical, then they come from different derivations. //

Lemma 4. If G is LR(k), then at most one state $\langle q, \bar{q}, \ell, X_{i+1}\ldots X_{i+k}\rangle$ can be added to S_i by a single completer operation.

Proof: Suppose $\langle q, \bar{q}, \ell, X_{i+1}\ldots X_{i+k}\rangle$ is added by the completer from $\langle p, \bar{p}, f, X_{i+1}\ldots X_{i+k}\rangle \in S_i$ and $\langle q, \bar{q}\text{-}1, \ell, X_{i+1}\ldots X_{i+k}\rangle \in S_f$. Then by theorem 1, $(q, \bar{q}, \ell)$ and $(p, \bar{p}, f)$ are i-states of $X_1\ldots X_{i+k}$. So

$$D_q \rightarrow C_{q1}\ldots C_{q\bar{q}} \overset{*}{\Rightarrow} X_{\ell+1}\ldots X_i$$

and

$$D_p \rightarrow C_{p1}\ldots C_{p\bar{p}} \overset{*}{\Rightarrow} X_{f+1}\ldots X_i$$

But then

$$X_{f+1}\ldots X_{f+k} \in H_k(C_{q\bar{q}}\, X_{i+1}\ldots X_{i+k})$$

so $(q, \bar{q}\text{-}1, \ell)$ is an f-state of $X_1\ldots X_{f+k}$. Therefore

$$C_{q1}\ldots C_{q(\bar{q}-1)} \overset{*}{\Rightarrow} X_{\ell+1}\ldots X_f$$

But all these derivations are part of the unique derivation

$$A_1\ldots A_m \overset{*}{\Rightarrow} X_1\ldots X_i$$

of the useful string $A_1\ldots A_m$. So since $C_{q\bar{q}} = D_p$, $D_q \rightarrow C_{q1}\ldots C_{q(\bar{q}-1)}\, D_p \overset{*}{\Rightarrow} X_{\ell+1}\ldots X_i$. If there is another state $\langle q', \bar{q}', \ell', X_{i+1}\ldots X_i$ added by the same completer operation, then by the same argument as above,

$$D_{q'} \rightarrow C_{q'1}\ldots C_{q'(\bar{q}'-1)}\, D_p \overset{*}{\Rightarrow} X_{\ell'+1}\ldots X_i$$

But by the same argument as in lemma 3 (using D_p instead of X_i), this is impossible. //

Now we will show that the number of states added by a single completer operation is bounded (lemma 5). Then we show that the number of states in a state set not added by the completer is bounded (lemma 6) and that the total number of completer operations is $\sim n$ (lemmas 7 and 8). Thus the total number of states is $\sim n$ (theorem 11).

Lemma 5. If G is LR(K), then dmt^k is a bound on the number of states that can be added by a completer operation.

Proof: If dmt^k+1 states are added by one completer operation to S_i then 2 of them must be $\langle p, j+1, f_1, \alpha\rangle$ and $\langle p, j+1, f_2, \alpha\rangle$ for some $p, j+1, \alpha, f_1 \neq f_2$. So $\exists$ ℓ such that $\langle p, j, f_1, \alpha\rangle$ and $\langle p, j, f_2, \alpha\rangle \in S_\ell$ and $\exists$ q such that $\langle q, \bar{q}, \ell, X_{i+1}\ldots X_{i+k}\rangle \in S_i$. Now this state was originally added to S_ℓ by the predictor as $\langle q, 0, \ell, X_{i+1}\ldots X_{i+k}\rangle$ so

$$X_{i+1}\ldots X_{i+k} \in H_k(C_{p(j+2)}\ldots C_{p\bar{p}}\ \alpha)$$

Therefore let β be a string such that

$$C_{p(j+2)}\ldots C_{p\bar{p}}\ \alpha \overset{*}{\Rightarrow} X_{i+1}\ldots X_{i+k}\beta$$

Let $g = i+k+|\beta|$. Let $X_{i+k+1}\ldots X_g = \beta$.

Then by applying theorem 3 $\bar{p}-(j+1)$ times, we get that S_g contains $\langle p, \bar{p}, f_1, \alpha\rangle$ and $\langle p, \bar{p}, f_2, \alpha\rangle$, and $\alpha = X_{g+1}\ldots X_{g+k}$. If $\bar{p} = j+1$, we already know that these states were added by a single completer

operation. If $C_{p\bar{p}}$ is terminal, they were added by the scanner. If $C_{p\bar{p}}$ is non-terminal and $\bar{p} > j+1$, then they were both added by a single completer operation on some state $\langle r, \bar{r}, h, \alpha\rangle$ for some r, h.

If they were both added by the scanner, lemma 3 is violated. If they were both added by a single completer, lemma 4 is violated. //

Lemma 6. If G is an LR(k) grammar, then dmt^k is a bound on the number of states in S_i not added by the completer.

Proof: Suppose the number of these states is dmt^k+1. Then $\exists$ p, j, $f_1 \neq f_2$, α such that $\langle p, j, f_1, \alpha\rangle$ and (p, j, f_2, α) are in S_i, but not by the completer. If $j = 0$, then $f_1 = f_2 = i$. If $C_{p1}\ldots C_{pj}$ is a terminal string, then $f_1 = f_2 = i-j$. So one of $C_{p1}\ldots C_{pj}$ must be non-terminal. Let h be the largest integer $h \leq j$ such that C_{ph} is non-terminal. We know that $\langle p, h, f_1, \alpha\rangle$ and $\langle p, h, f_2, \alpha\rangle \in S_g$ (where $g = i-(j-h)$), and these states were added by the completer. Choosing the first of these, since it was added by the completer $\exists$ ℓ such that $\langle p, h-1, f_1, \alpha\rangle \in \ell$ and $\exists$ q such that $\langle q, \bar{q}, \ell, X_{g+1}\ldots X_{g+k}\rangle \in S_i$. This state was originally added to S_ℓ by the predictor as $\langle q, 0, \ell, X_{i+1}\ldots X_{i+k}\rangle$, so

$$X_{i+1}\ldots X_{i+k} \in H_k(C_{p(h+1)}\ldots C_{p\bar{p}}\ \alpha)$$

so by the same argument as in lemma 5, $\exists$ an extension $X_1\ldots X_g$ such that S_g contains $\langle p, \bar{p}, f_1, \alpha\rangle$ and $\langle p, \bar{p}, f_2, \alpha\rangle$, $\alpha = X_{g+1}\ldots X_{g+k}$, and these two states were added either by a single completer operation or by the scanner, unless $\bar{p} = h$. But this would mean that $j = h$, and this is impossible, since C_{ph} is non-terminal but $\langle p, j, f_1, \alpha\rangle$ and $\langle p, j, f_2, \alpha\rangle$ were not added by the completer.

If they were both added by the scanner, lemma 3 is violated. If they were both added by a single completer, lemma 4 is violated. //

Lemma 7. If G is LR(k), every completer operation represents a different step in the derivation of $X_1 \ldots X_n$.

Proof: By theorem 1, any state $\langle p, \bar{p}, f, X_{i+1} \ldots X_{i+k} \rangle$ to which the completer is applicable means that $(p, \bar{p}, f)$ an i-state of $X_1 \ldots X_{i+k}$. So

$$D_p \rightarrow C_{p1} \ldots C_{p\bar{p}} \stackrel{*}{\Rightarrow} X_{f+1} \ldots X_i$$

and

$$R \stackrel{*}{\Rightarrow} X_1 \ldots X_f D_p X_{i+1} \ldots X_{i+k}\ \delta$$

for some extension of $X_1 \ldots X_{i+k}$. But since the grammar is LR(k), a step (at i) in the derivation of one extension of $X_1 \ldots X_{i+k}$ is a step in the derivation of any extension of $X_1 \ldots X_{i+k}$, including $X_1 \ldots X_n$.

Furthermore, every such completer operation represents a different step in the derivation because states are not duplicated in the same state set. So in a different completer operation, either p, f, or i must be different, making it a different step in the derivation. //

Lemma 8. There is a bound of the form Cn on the length of the leftmost derivation of a sentence of length n with respect to an unambiguous grammar.

Proof: If any non-terminal N has the property that $R \stackrel{*}{\Rightarrow} \alpha N \beta$ for some α, β and $N \stackrel{*}{\Rightarrow} \Lambda$, then there is only one derivation $N \stackrel{*}{\Rightarrow} \Lambda$ since the grammar is

unambiguous. Let z be the length of the longest such derivation for all the non-terminals in G.

In any derivation of a sentence in G, let an occurrence of a non-terminal N be a null occurrence if Λ is eventually derived from that occurrence of N in that derivation.

Let the width of a sentential form be the number of symbols in it which are not null occurrences. Each step in a derivation does one of four things:

(1) Increases the width of the sentential form.

(2) Is a step in a null derivation.

(3) Transforms a non-terminal into a single terminal (plus possibly some null occurrences).

(4) Transforms a non-terminal into another non-terminal (plus possibly some null occurrences).

In the leftmost derivation of a sentence, there can be at most u occurrences of steps of type (4) before there is a step of type (1) or (3) (here we are ignoring type (2)). If there were u+1 steps of type (4), then we would have $N \overset{*}{\Rightarrow}_1 N \overset{*}{\Rightarrow}_2 \alpha$ in a non-trivial way as part of the derivation for some non-terminal N. But then we could also have a shorter derivation by bypassing the second N, $N \overset{*}{\Rightarrow}_1 \alpha$. This violates the fact that the grammar is unambiguous. There can be at most n type (3) steps since the sentence is of length n. And there can be at most n-1 type (1) steps for the same reason. So, ignoring type (2) steps, there are at most (n+n-1)u steps.

Let an initial null occurrence of a non-terminal be a null occurrence which is not an intermediate step in a null derivation. Initial null occurrences can only be created by steps of type (1), (3), or (4), and at most m of them can be created by each such step. So there are at most (n+n-1)um initial null occurrences. So there are at most (n+n-1)um different null derivations, and each has at most z steps. Therefore (n+n-1)umz is a bound on the number of type (2) steps.

So the total number of steps is bounded by $(n+n-1)u(mz+1) \approx 2umzn$. //

Theorem 11. If G is LR(k), then there is a bound of the form Cn on the total number of states constructed.

Proof: By lemma 5, dmt^k is a bound on the number of states not added by the completer in any state set. So $(n+1)dmt^k$ is a bound on the total number not added by the completer.

By lemma 7, every completer operation represents a different step in the derivation of $X_1 \ldots X_n$ for an LR(k) grammar. So the total number of completer operations is bounded by the length of a derivation, which is the same as the length of a leftmost derivation, or Cn, by lemma 8. By lemma 6, at most dmt^k states can be added by a completer operation. So the total number of states added by the completer is bounded by $Cdmt^k n$.

Summing these two gives us the result:

$$(n+1)dmt^k + Cdmt^k n \approx Cdmt^k n \text{ //}$$

We will generalize the theorem to be proved to include not only LR(k) grammars, but any finite union of them. A union of two grammars is just the straight-forward way of combining the two grammars to obtain a grammar which generates a language which is the union of the two original languages.

Definition. A union U of b grammars $G_1,\ldots,G_b$ with roots $R_1,\ldots,R_b$ is obtained as follows:

(1) Systematically change the non-terminal symbols of eac grammar, so that no non-terminal appears in more than one grammar.

(2) Let the non-terminals of U be the union of the changed non-terminals of $G_1,\ldots,G_b$ plus R.

(3) Let the terminals of U be the union of the terminals of $G_1,\ldots,G_b$.

(4) Let the productions of U be the union of the productions of $G_1,\ldots,G_b$ plus the following productions:

$$R \rightarrow R_1$$
$$\vdots$$
$$R \rightarrow R_b$$

(5) Let R be the root of U.

Theorem 12. There is a bound of the form Cn on the number of steps in computing REC(G, $X_1\ldots X_n$, k) if G is a finite union of LR(k) grammars

(write U LR( )).

Proof: Let G be a union of b grammars.

The first states added to S_0 are

$$D_0 \rightarrow .R \quad \dashv^k \quad 0$$

$$R \rightarrow .R_1 \quad \dashv^k \quad 0$$

$$\vdots \qquad \vdots \qquad \vdots$$

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

The algorithm then proceeds to process each of the grammars in parallel. The b different recognition processes cannot interfere with each other in any way because the non-terminals from each grammar are distinct and therefore the productions (and thus the states) are all distinct. So the bound for such a union grammar is b times the bound on an LR(k) grammar. (More generally, the time required for any union grammar is the sum of the times for the component grammars.)

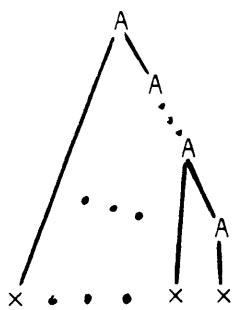
The bound for an LR(k) grammar is obtained as follows:

(a) The total number of states is bounded by Cn, so it takes at most Cn steps to scan them all. The scanner adds one state and the predictor adds at most at^k states for each state scanned. So including dmt^k to test each state, we get $C(dmt^k)(at^k)n$ steps for these.

(b) The completer performs at most dmt^k steps for each state it adds, since it adds each one in only one way because the grammar is unambiguous. So it performs at most $Cdmt^kn$ steps. //

Notice that this result and the previous bounded state result do not hold for any k. That is, in this case, we know the algorithm will

run in time n on a union of LR(k) grammars only if the look-ahead it uses is at least as large as k. Grammar RR provides an example of what happens if k isn't large enough. We have already seen that it is recognized in time n if k = 1. But if k = 0, (page 47) we can see that S_1 contains 5 states, S_2 6 states, and in general S_i will contain i+4 states up through S_n. Thus the total number of states is $\sim n^2$ and so is the time. The reason for this is as follows: After we have scanned x^i, we could be finished scanning x's (if i = n) and therefore we would need to construct states which represent the whole parse of the string as follows:



In the case that k = 1, we look-ahead at the next symbol before we construct all these states and we do so only if the next symbol is a "4", that is, we do this only once. But if we don't look-ahead, we are forced to construct all these states at each step just in case that step one was the last one.

So with the algorithm as it is, we need the look-ahead to get the time n result for all LR(k) grammars. There seems to be a way of obtaining the LR(k) time n result using k = 0 by a slight change in the algorithm. However, we have not investigated this in detail,

so it is only a conjecture, and we won't describe it here. Even if this modification were valid, the look-ahead is still very important in that a look-ahead of just 1 will cut down the time used significantly in many grammars even if it doesn't produce any new theoretical results.

We now have characterized two classes of grammars which can be done in time n--bounded state and finite unions of LR(k). Thus our algorithm does in time n a much larger class of grammars than any previous one. We don't, however, have a full characterization of the time n grammars for our algorithm. It is clear that there will be some grammars which are right recursive (and therefore not bounded state) which are not finite unions of LR(k), yet which are simple enough to be time n. One possible characterization for the time n grammars is that they are all grammars for which the total number of states constructed is proportional to n. This has not been investigated.

Notice that the time n conditions, like the LR(k) condition, is not symmetric. That is, if we implemented a recognizer exactly like the one in this paper except that it went from right to left, then we would get a different set of grammars to be time n. Grammar RL (page 63) is an example of one which is time n^2 from left to right, as the example run clearly shows. However, it seems clear from the structure of the grammar that it runs in time n from right to left.

The time n grammars seem to be rather large; they seem to include most unambiguous grammars. In fact the only unambiguous grammars we have found which are not time n in either direction are certain pallindrome grammars with unmarked centers and variations on them.

(A language is a palindrome if each of its sentences is the same reversed.) Grammar PAL (page 65) illustrates this. 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 $\sim n^2$.

The time n grammars also include a surprising number of quite ambiguous grammars. Grammar BK (page 39), we have already seen, has unbounded ambiguity, yet it is a time n grammar.

Even more surprising is that grammar GAP has unbounded direct ambiguity and yet is a time n grammar. On page 67 is a run on this grammar. One can see that all the state sets after S_2 will also contain seven states until a "c" is scanned. Furthermore, when recognizing ab^nc, S_{n+2} will be

B → bc.	n
B → bB.	n-1
⋮	⋮
B → bB.	1
R → AB.	0^n
ϕ → R.	0

So it contains $\sim$ n states; but since the rest of the state sets are of bounded size, GAP is a time n grammar.

Grammar RL

$R \rightarrow A \mid aRb$

$A \rightarrow a \mid Aa$

sentences: $a^n a^m b^n$ $(m \geq 0,\ n \geq 1)$

REC(RL, a^3, 0):

S_0	$\phi \rightarrow .R\dashv$	0	✓
	$R \rightarrow .A$	0	✓
	$R \rightarrow .aRb$	0	
	$A \rightarrow .a$	0	
	$A \rightarrow .Aa$	0	✓
S_1	$R \rightarrow a.Rb$	0	✓
	$A \rightarrow a.$	0	
	$R \rightarrow .A$	1	✓
	$R \rightarrow .aRb$	1	
	$R \rightarrow A.$	0	
	$A \rightarrow A.a$	0	
	$A \rightarrow .a$	1	
	$A \rightarrow .Aa$	1	✓
	$\phi \rightarrow R.\dashv$	0	
S_2	$R \rightarrow a.Rb$	1	✓
	$A \rightarrow a.$	1	
	$A \rightarrow Aa.$	0	
	$R \rightarrow .A$	2	✓
	$R \rightarrow .aRb$	2	
	$R \rightarrow A.$	1	
	$A \rightarrow A.a$	1	
	$R \rightarrow A.$	0	
	$A \rightarrow A.a$	0	
	$A \rightarrow .a$	2	
	$A \rightarrow .Aa$	2	✓
	$R \rightarrow aR.b$	0	
	$\phi \rightarrow R.\dashv$	0	

S_3	$R \rightarrow a.Rb$	2	✓
	$A \rightarrow a.$	2	
	$A \rightarrow Aa.$	1	
	$A \rightarrow Aa.$	0	
	$R \rightarrow .A$	3	✓
	$R \rightarrow .aRb$	3	
	$R \rightarrow A.$	2	
	$A \rightarrow A.a$	2	
	$R \rightarrow A.$	1	
	$A \rightarrow A.a$	1	
	$R \rightarrow A.$	0	
	$A \rightarrow A.a$	0	
	$A \rightarrow .a$	3	
	$A \rightarrow .Aa$	3	✓
	$R \rightarrow aR.b$	1	
	$R \rightarrow aR.b$	0	
	$\phi \rightarrow R.\dashv$	0	
S_4	$\phi \rightarrow R.\dashv$	0	

Grammar PAL

$A \rightarrow x \mid xAx$

sentences: x^n ($n \geq 1$, n odd)

REC(PAL, x^5, 0):

S_0

$\phi \rightarrow .A\dashv$	0
$A \rightarrow .x$	0
$A \rightarrow .xAx$	0

S_1

$A \rightarrow x.$	0
$A \rightarrow x.Ax$	0
$\phi \rightarrow A.$	0
$A \rightarrow .x\dashv$	1
$A \rightarrow .xAx$	1

S_2

$A \rightarrow x.$	1
$A \rightarrow x.Ax$	1
$A \rightarrow xA.x$	0
$A \rightarrow .x$	2
$A \rightarrow .xAx$	2

S_3

$A \rightarrow xAx.$	0
$A \rightarrow x.$	2
$A \rightarrow x.Ax$	2
$\phi \rightarrow A.\dashv$	0
$A \rightarrow xA.x$	1
$A \rightarrow .x$	3
$A \rightarrow .xAx$	3

S_4

$A \rightarrow xAx.$	1
$A \rightarrow x.$	3
$A \rightarrow x.Ax$	3
$A \rightarrow xA.x$	0
$A \rightarrow xA.x$	2
$A \rightarrow .x$	4
$A \rightarrow .xAx$	4

S_5	$A \rightarrow xAx.$	0
	$A \rightarrow xAx.$	2
	$A \rightarrow x.$	4
	$A \rightarrow x.Ax$	4
	$\phi \rightarrow A.\dashv$	0
	$A \rightarrow xA.x$	1
	$A \rightarrow xA.x$	3
	$A \rightarrow .x$	5
	$A \rightarrow .xAx$	5
S_6	$\phi \rightarrow A\dashv.$	0

Grammar GAP

root: $R \rightarrow AB$

$A \rightarrow a \mid Ab$

$B \rightarrow bc \mid bB$

sentences: $ab^{i}c$ $(i \geq 1)$

REC(GAP, $ab^{3}c$, 0):

S_0	$\phi \rightarrow .R\dashv$	0
	$R \rightarrow .AB$	0
	$A \rightarrow .a$	0
	$A \rightarrow .Ab$	0
S_1	$A \rightarrow a.$	0
	$R \rightarrow A.B$	0
	$A \rightarrow A.b$	0
	$B \rightarrow .bc$	1
	$B \rightarrow .bB$	1
S_2	$A \rightarrow Ab.$	0
	$B \rightarrow b.c$	1
	$B \rightarrow b.B$	1
	$R \rightarrow A.B$	0
	$A \rightarrow A.b$	0
	$B \rightarrow .bc$	2
	$B \rightarrow .bB$	2
S_3	$A \rightarrow Ab.$	0
	$B \rightarrow b.c$	2
	$B \rightarrow b.B$	2
	$R \rightarrow A.B$	0
	$A \rightarrow A.b$	0
	$B \rightarrow .bc$	3
	$B \rightarrow .bB$	3
S_4	$A \rightarrow Ab.$	0
	$B \rightarrow b.c$	3
	$B \rightarrow b.B$	3
	$R \rightarrow A.B$	0
	$A \rightarrow A.b$	0
	$B \rightarrow .bc$	4
	$B \rightarrow .bB$	4

S_5	$B \rightarrow bc.$	3
	$B \rightarrow bB.$	2
	$B \rightarrow bB.$	1
	$R \rightarrow AB.$	0^3
	$\phi \rightarrow R.\dashv$	0
S_6	$\phi \rightarrow R\dashv.$	0

X. THE COMPILED ALGORITHM

We can recognize a large class of grammars in time n, but for practical purposes we are forced to ask, how large is the constant coefficient of n? Unfortunately, it turns out that if one uses the algorithm as given, the coefficient is likely to be quite large, at least compared with those of most of the time n algorithms. This is understandable in a way, since they are specialized algorithms, tailored to specific classes of grammars, and ours is a general algorithm which just happens to be time n on many grammars, without its being told that they are of any special form.

But we can expect better performance from our algorithm if we are willing to tell it that it has a time n grammar (time n for our algorithm). If we know that, we can run the grammar through a compilation process to produce a recognizer for the grammar. Then we can use only the compiled recognizer each time we recognize a string. This will cut down the coefficient of n to approximately the same magnitude as the other time n algorithms. The reason for the cutdown is that the compilation process does much of the work once that would otherwise have been done with every run of the recognizer. Let's look at an example. If we examine an ordinary run on grammar LR (page 71), we can see that all the states after S_1 are identical. So we can compile an Algol-like parser for it as follows:

```
i ← 1;
S0:  If Xi = x then
        begin i ← i+1; go to S1 end
     else Error;

S1:  If Xi = x then
        begin i ← i+1; go      1 end
     else if Xi = ⊣ then Halt
     else Error.
```

We have combined S_1 and S_2 because their actions are the same even though the state sets are different.

We won't be as lucky with all time n grammars as with this one. We won't always get them to repeat their states so easily, especially to repeat them with the same values for the pointers for all length input strings. But we do know that in the case of a time n grammar, there is a bound on the number of states in any state set.* So if we ignore the pointers, there are only a finite number of distinct state sets which can ever appear in the recognition of a string with respect to this grammar. Therefore, given a particular grammar, we would like to compute all such state sets at compile time, compiling a recognizer which works according to the algorithm, but only examines the pointers and the input str during the run-time recognition process. For each

*This isn't really true for right recursive grammars, but it will be in the compiled algorithm.

Grammar LR

$A \rightarrow x \mid Ax$

sentences x^n $(n \geq 1)$

REC(A,xx,0):

S_0	$\emptyset \rightarrow .A\dashv$	0
	$A \rightarrow .x$	0
	$A \rightarrow .Ax$	0
S_1	$A \rightarrow x.$	0
	$\emptyset \rightarrow A.\dashv$	0
	$A \rightarrow A.x$	0
S_2	$A \rightarrow Ax.$	0
	$\emptyset \rightarrow A.\dashv$	0
	$A \rightarrow A.x$	0
S_3	$\emptyset \rightarrow A\dashv.$	0

compile-time state set we compile a block of code which at run time tests the input string and the values of the pointers in order to construct a new set of pointers and go on to a new block of code.

As an example, we show the compile-time state sets for grammar RR (page 73, left side). Notice that along with each state in the state set, instead of a pointer, we keep a list of all the state sets which can be pointed to by a pointer in that position. The code on the right side of page 73 is abbreviated code for the Algol code to manipulate the pointers and input string. The expansion of this code is on pages 74-75, and a sample run is on page 76. We have numbered the productions in the grammar, so that these numbers could be used as indices for the run time data structures in the compiled code. The block of compiled code for each state set is put immediately to the right of it.

The meaning of both the abbreviated code and the compiled Algol code is explained in the following full description of the compilation algorithm. It is advisable to read this description first and then examine the grammar RR compiled code. The run-time data structures for which we are compiling code are as follows:

Run-Time Data Structures

(1) A three dimensional array F[0:n+1,0:d-1,0:m-1]. We store at F[i,p,j] all f such that $\langle p,j,f,\alpha \rangle \in S_i$ for some α.

(2) The input string is X[0:n+k+1].

(3) A vector S[0:n+1] contains the names of the state sets current at each i.

Grammar RR

$A \rightarrow {}^{1}x \mid {}^{2}xA$

sentences: x^n $(n \geq 1)$

			Condition	Action	Go To
S_0	$\emptyset \rightarrow .A\dashv$	$\dashv$ S_0		0 1,2	
	$A \rightarrow .x$	$\dashv$ S_0	x	(1,0)	
	$A \rightarrow .xA$	$\dashv$ S_0		(2,0)	S_1
S_1	$A \rightarrow x.$	$\dashv$ S_0, S_1		1,2	
	$A \rightarrow x.A$	$\dashv$ S_0, S_1	x	(1,0)	
	$A \rightarrow .x$	$\dashv$ S_1		(2,0)	S_1
	$A \rightarrow .xA$	$\dashv$ S_1	$\dashv, 1=S_1$	(2,1)	S_2
			$\dashv, 1=S_0$	(0,0)	S_3
S_2	$A \rightarrow xA.$	$\dashv$ S_0, S_1	$\dashv, 2=S_1$	(2,1)	S_2
			$\dashv, 2=S_0$	(0,0)	S_3
S_3	$\emptyset \rightarrow A.\dashv$	$\dashv$ S_0	$\dashv$	(0,1)	S_4
S_4	$\emptyset \rightarrow A\dashv$	$\dashv$ S_0		HALT	

Compiled Code

```
i ← 0;
S0: F[i,0,0] ← i; F[i,1,0] ← i;  F[i,2,0] ← i; S[i] ← S0;
    if X[i+1] = x then
    begin F[i+1,1,1] ← F[i,1,0]; F[i+1,2,1] ← F[i,2,0]; i ← i+1;
        go to S1 end else
    Error;

S1: F[i,1,0] ← i; F[i,2,0] ← i; S[i] ← S1;
    if X[i+1] = x then
    begin F[i+1,1,1] ← F[i,1,0]; F[i+1,2,1] ← F[i,2,0]; i ← i+1;
        go to S1 end else
    if X[i+1] = -1 then
        if S[F[i,1,1]] = S1 then
        begin F[i,2,2] ← F[F[i,1,1],2,1]; go to S2 end else
        if S[F[i,1,1]] = S0 then
        begin F[i,0,1] ← F[F[i,1,1],0,0]; go to S3 end
        Error else
    Error;

S2: S[i] ← S2;
    if X[i+1] = -1 then
        if S[F[i,2,2]] = S1 then
        begin F[i,2,2] ← F[F[i,2,2],2,1]; go to S2 end else
```

if S[F[i,2,2]] = S_0 then

begin F[i,0,1] ← F[F[i,2,2],0,0]; go to S_3

Error else

Error;

S_3: if X_{i+1} = 1 then

begin F[i,0,2] ← F[i,0,1]; go to S_4 end else

Error;

S_4: Halt;

Trace of Compiled Run

i	0	1	2	3	4
X[i]		×	×	×	+
S[i]	S_0	S_1	S_1	S_1, S_2, S_2, S_3	
F[i,1,0] & F[i,2,0]	0	1	2	3	
F[i,1,1] & F[i,2,1]		0	1	2	
F[i,2,2]				1,0	
F[i,0,0]	0				
F[i,0,1]				0	
F[i,0,2]					0

At compile time we manipulate objects which we will call C-states, because they are almost like states in the original algorithm, except that they contain a pointer to a collection of states rather than a pointer to a place in the input string. Throughout this section, we will refer to C-states as states.

A state collection is the means by which we identify state sets which are the same except for their pointers.

Definition: A C-state is a quadruple $<p,j,T,\alpha>$, where p, j, and α are as in an ordinary state and T is a pointer to a state collection.

Definition: A state collection is an equivalence class of state sets under the relation $S_1 \sim S_2$ if there is a 1-1 correspondence between their states such that $<p_1,j_1,T_1,\alpha_1>$ and $<p_2,j_2,T_2,\alpha_2>$ correspond if $p_1 = p_2$, $j_1 = j_2$, and $\alpha_1 = \alpha_2$.

Definition: $\rightarrow S$, where S is a state set, is a pointer to the state collection of which S is a member.

A precise description of the compilation algorithm is on pages 80-83. The following is an intuitive description: It is a recursive procedure which takes as input a state set. It is supposed to compile the code for that state set and for all state sets which we might go to from that state set in recognizing any input string.

It works as follows: Given a state set S, it computes the predictor on S and compiles any code needed to perform this at run time. Then, if

there are no final states in S, it takes each legal terminal x that could appear next in the input string. For each of these, it pretends that x was next, computes the scanner under this assumption (compiling code for it), and calls itself recursively to compile the new state set which results. Thus we are in some sense running the recognizer on all possible input strings at compile time. This can't really be done, of course, so we must have a termination condition to keep our recursive calls on the compilation algorithm from going on indefinitely. Before we describe this, however, we will describe how the completer works.

If there is a final state in the state set, then for each possible look-ahead string α, we first compile a test to see if the next k symbols match α. Then we compute the completer on all final states for which α is the look-ahead string (compiling the code for this), we delete the final states, replacing them by the states added by the completer, and then we call the compilation algorithm recursively on the newly created state.

Notice, however, that we cannot compute the completer in the ordinary way. We are at compile time, so we have no input string and no pointer f. Instead we have a pointer to a state collection. We take each of the state sets in this collection one by one and assume that it was the state pointed to; then we compute the completer in the ordinary way. For example, in grammar RR (page 73), S_1 is a state collection consisting of two states--one with S_0 as the pointer for the first two states and one with S_1. At one point we are working with the state set consisting of the single state

$$A \rightarrow xA. \dashv S_1$$

In computing the completer on this, we can get two different state sets depending on which of the members of S_1 we use. The two state sets are

$$A \rightarrow xA. \dashv S_0$$

and

$$A \rightarrow xA. \dashv S_1$$

The reason we use pointers to state collections instead of just to states is so we can add a terminating condition to the algorithm. We describe that now. This terminating condition must have the property that the algorithm terminates only if we have processed all possible state sets that can be encountered in recognizing any string with respect to the grammar. Suppose that in the process of compiling a state set S (and the recursive calls it produces), we encounter another call on S. Can this second call on S produce any new state sets? The answer is that it can, but only if some new state set has been created since the last time that S was compiled. If no new state sets have been created since that time, then this call can produce only the state sets that the old one produces, so we can terminate this branch of the algorithm now.

This termination testing is done by the first four blocks of the flowchart (page 81). Each state is marked processed as it is compiled. However, whenever a new state is encountered we mark all states unprocessed. So if we try to compile a state which is marked processed,

that means that no new state sets were encountered since the last time it was compiled.

Note that at the end of the completer, we must compute the scanner on any states which were not included in the processing done by the completer.

For each state collection, the compilation algorithm compiles a block of code labeled by the name of the state llection. These blocks consist of an action followed by a sequence of tests. Each test consists of a condition, action, and go to. The tests are implicitly followed by a branch to an error exit if they all fail. In the description of the compilation algorithm, "compile under →S" means compile that code into the block of code labeled →S.

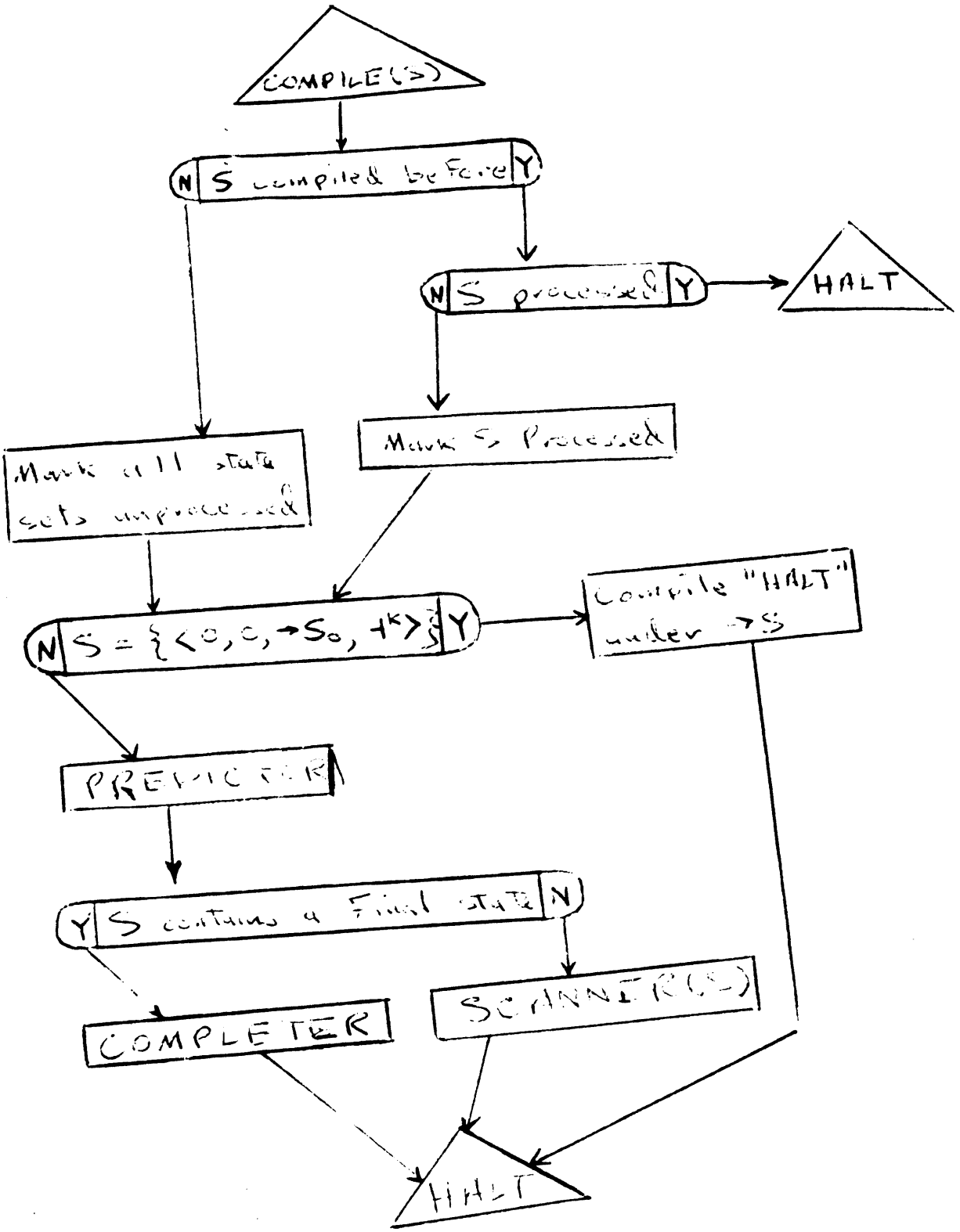
The compilation algorithm we have described here assumes that there are no null productions. This problem can be overcome, but we choose to ignore it in this description.

This compilation algorithm has not been either debugged on a machine or proven correct, so it may have some small bugs, but the general idea is not in error.

The Compilation Algorithm--COMP(G,k):

Add $<0,0,\rightarrow S,\dashv^{k}>$ to S.

Compute COMPILE(S).


COMPILE(S)
N
S compiled before
Y
N
S processed
Y
HALT
Mark all state sets unprocessed
Mark S Processed
N
S = {<0, c, →S0, +K>}
Y
Compile "HALT" under →S
PREDICTOR
Y
S contains a Final state
N
COMPLETER
SCANNER(S)
HALT

PREDICTOR

Scan the states of S in **order.** For each state $\langle p,j,T,\alpha\rangle$, if $j < \bar{p}$ and $C_{p(j+1)}$ is non-terminal, then for each q such that $C_{p(j+1)} = D_q$, and for each $\beta \in H_k(C_{p(j+2)}\ldots C_{p\bar{p}}\alpha))$, add $\langle q,0,\rightarrow S,\beta\rangle$ to S and compile under $\rightarrow S$: "q." This is interpreted "F[i,q,0] ← i."

COMPLETER

For each α such that $\exists \langle p,\bar{p},T,\alpha\rangle \in S$, let $\langle p_1,\bar{p}_1,T_1\alpha\rangle,\ldots,$ $\langle p_r,\bar{p}_r,T_r,\alpha\rangle$ be the final states in S with look-ahead string α. For each possible combination of a state set $S_{i_1} \in T_1,\ldots,S_{i_r} \in T_r$, do the following 3 steps:

(1) $S'_\alpha \leftarrow \{\langle p,j,T,\beta\rangle \mid j < p \text{ and } \alpha \in H_k(C_{p(j+1)},\ldots,C_{p\bar{p}}\beta)\}$

(2) For each state $\langle q_j,\ell_j,U,\beta\rangle \in S_{i_j}$ such that $C_{q_j(\ell_j+1)} = D_{p_j}$, add $\langle q_j,\ell_j+1,U,\beta\rangle$ to S'_α $(j = 1,\ldots,r)$.

(3) Compute COMPILE(S'_α)

Compile under $\rightarrow S$:

$$\begin{array}{lll} \text{"}\alpha, p_1 = T_1 & (q_1,\ell_1) & \text{[for each } q_1,\ell_1 \text{ as in (2)]} \\ \quad\vdots & \quad\vdots & \quad\vdots \\ \quad p_r = T_r & (q_r,\ell_r) \quad S'_\alpha \text{"} & \text{[for each } q_r,\ell_r \text{ as in (2)]} \end{array}$$

This is interpreted

"if $X[i+1]\ldots X[i+k] = \alpha \wedge$

$S[F[i,p_1,\bar{p}_1]] = \rightarrow T_1 \wedge \ldots \wedge S[F[i,p_r,\bar{p}_r]] = \rightarrow T_r$ then

$$\begin{array}{ll} \text{begin } F[i,q_1,\ell_1+1] \leftarrow F[F[i,p_1,\bar{p}_1],q_1,\ell_1] & \text{[for each } q_1,\ell_1] \\ \quad\vdots & \quad\vdots \\ \quad F[i,q_r,\ell_r+1] \leftarrow F[F[i,p_r,\bar{p}_r],q_r,\ell_r] & \text{[for each } q_r,\ell_r] \end{array}$$

$S[i] \leftarrow \rightarrow S'_\alpha$

Go To S'

end"

Let S" ← {states s | s $\notin S'_\alpha$ for any of the above α}

Compute SCANNER(S').

SCANNER(U)

For each x such that $\exists$ <p,j,T,α> $\in$ U such that

$x = C_{p(j+1)}$,

U' ← {<p,j+1,T,α> | <p,j,T,α> $\in$ U and $C_{p(j+1)} = x$}

Compile under →S:

"x (p_1, j_1)
⋮
(p_m, j_m) U' "

where $\{(p_1, j_1), \ldots, (p_m, j_m)\} = \{(p,j) \mid \exists\ \alpha, T$ such that <p,j+1,T,α> $\in$ U'$\}$

The above compiled code means

"if X[i+1] = x then

begin $F[i+1, p_1, j_1+1] \leftarrow F[i, p_1, j_1]$
⋮ ⋮
$F[i+1, p_m, j_m+1] \leftarrow F[i, p_m, j_m]$

i ← i+1

S[i] ← →U'

Go To U'

end"

Compute COMPILE(S')

If we examine the grammar RR example (page 73) in detail we will find that some of the run-time pointer manipulation which gets compiled is unnecessary. In particular the four circled parts of the abbreviated code are unnecessary. Extraneous code like this may appear because the pointer is set but never tested in a completer operation, or if it is tested, there may be only one possible value for it, so the test is meaningless.

In any case, we can prune away all extraneous code by making an analysis of the compiled code. This takes the form of working backwards from all meaningful pointer tests and marking all the code which leads up to them. We could then delete all unmarked code as being useless. We expect that such a pruning procedure would significantly improve the run-time speed of the compiled recognizer. For many grammars we could also extend this pruning procedure to delete state sets which accomplish nothing (after pruning) and to combine any two state sets whose actions are the same. Grammar LR2 (page 125) provides a good example of pruning.

Now we investigate just what properties this compilation algorithm has. If the algorithm does terminate, it's clear that no new state sets could have been created, so we do have a valid recognizer for the grammar. Can we also say that the grammar was a time n grammar? Unfortunately, not quite. What we do know is that there are only a finite number of state sets which can be created by the grammar and

therefore there is a bound on the size of any of them. But these state sets are not quite the same as those of the original algorithm. The difference comes in the way we treat the completer. In the original algorithm, we compute the completer on all the states in the state set, including those which may have just been added by the completer, until there are no more states to which the completer is applicable. But in the compilation algorithm, we apply the completer once, deleting the old final states and substituting the new ones to form a new state set. Then if the completer is applicable again, we apply it to the new state set, again deleting the final states after it is applied. So in some cases where we might end up with $\sim i$ states in S_i according to the old algorithm (because of the accretion of old final states), we can get at most a bounded amount by the compilation algorithm because of the deletion of final states. Grammar RR is a perfect example of this. In the original run (page 47), S_3 has 7 states (and S_n would have had n+4 states) but in the compiled run (page 73), no state set has more than 4 states in it.

From this insight we should be able to come up with a definition of what grammars can be compiled by this algorithm. We will not try to make it independent of the algorithm.

Definition: A grammar G is compilable with look-ahead k, C(k), if $\exists$ a bound on the number of states in any state set not added by the completer, and $\exists$ a bound on the number of states added by any single completer operation.

This is equivalent to saying that there is a bound on the number of states in any state set generated by the compilation algorithm. Now by this definition, grammar RR is C(1) even though it is not BS(1), but it is LR(1) and time n using k = 1. It also provides an example of something which is compilable but not time n. It is C(0), but as we have seen before (page 47), it definitely takes time n^2 when run with k = 0.

So, in conclusion, we have four classes of grammars having the following relations:

$$U\ LR(k) \subset \text{time } n$$

$$\text{bounded state} \subset \text{time } n$$

$$\text{time } n \subset \text{compilable}$$

and neither U LR(k) nor bounded state is included in the other. Furthermore, all these inclusions are proper. We have already shown all these facts except that there are time n grammars and bounded state grammars which are not U LR(k). This will be shown in Section XI.

So we know that if the compilation algorithm terminates, the grammar is compilable. It is also obvious that for any non-compilable grammars, the algorithm won't terminate, because there will be an infinite number of different state sets to be processed, and so the terminating conditions will never be met. However, it is unfortunately also possible for the algorithm to go on indefinitely on a compilable grammar. This can happen in the following way:

Notice that we have guaranteed that if the algorithm terminates, all the state sets that could be needed will have been created. We did

not guarantee that only those needed would be created. In fact this is not necessarily so. When we perform the completer on a state set S, we pick up all the state sets in the state collection which is pointed to by some final state s in S, not just the one state set which was current when that s was created originally. So in using all these state sets, we may create a new state set which couldn't have been created by any run of the original algorithm. And, of course, if we can produce one unneeded state set, the processing of it may produce an infinite number of unneeded ones, where the number of needed ones was finite.

Given this, one wonders why we use state collections at all. Why not just perform the completer on the one state set which was valid when s was created and avoid this trouble? We could, but close inspection of the algorithm will reveal that this would invalidate the terminating condition we are using.

So in conclusion, if the compilation algorithm terminates, the grammar is compilable. If it doesn't terminate, the grammar may or may not be compilable. In Section XII we show that one cannot expect to do better than this in deciding whether or not a grammar is compilable.

If the compilation algorithm doesn't always terminate, even on compilable grammars, how can we use it in a practical sense to compile a recognizer for a grammar? One solution is as follows: We run the algorithm with a built-in limit on how far it can go. This could be a time limit, a limit on the size of any state set constructed, a limit on the total number of state sets constructed, etc. With some

experience we could obtain reasonable estimates for these limits as a function of the size of the grammar. Then we run the compilation algorithm with the limiting condition. If it compiles the grammar, we are through. If it terminates because of the limiting condition, we print out the state sets constructed so far and examine them. In most cases, we should be able to tell by examination whether or not it appears that the algorithm will not terminate--this might be because a state set appears to be growing in size in some uniform way and there appears to be no reason to limit its growth. If it appears that the algorithm won't terminate, we can give up on this grammar deciding that it is not compilable (or it might be one of those few compilable grammars on which the algorithm doesn't terminate). If it appears that the algorithm will terminate, we run it again with a limiting condition and repeat the process. This should not be a serious problem, in compilers at least, because it seems likely that all programming language grammars will be easily compilable.

XI. AN EXAMPLE

In Section IX we claim that there are time n grammars which are not finite unions of LR(k) and we claim that the time n grammars are very large. In Section X we also claim that U LR(k) does not include bounded state. In this section we want to give these claims some validity by producing grammar N, which is obviously not U LR(k), but even more strongly, L(N) is not a finite union of deterministic languages (those languages generated by LR(k) grammars). We run the compilation algorithm on grammar N and it terminates. So N is compilable. Further, N is not right recursive, so N must be bounded state as well, and certainly a time n grammar.

L(N) was obtained from [UI 65], pages 22-25, where it is proved that it is not a finite union of deterministic languages. Grammar N, L(N), and the compilation with k = 1 are on pages 90-96. This should also serve as a more interesting example of the application of the compilation algorithm. We have used some abbreviations: "+ac" in the condition column means that the next symbol can be any of these. "8=$S_{1,2}$" means that 8 can point back to either S_1 or S_2.

Grammar N

$A \rightarrow {}^{8}aAb \mid {}^{9}ab$

$C \rightarrow {}^{6}aCbb \mid {}^{7}abb$

$D \rightarrow {}^{3}A \mid {}^{4}Ac \mid {}^{5}Ccc$

$R \rightarrow {}^{1}\Lambda \mid {}^{2}RD$

$L(A) = a^n b^n \quad (n \geq 1)$

$L(C) = a^n b^{2n} \quad (n \geq 1)$

$L(D) = a^n b^n \quad (n \geq 1)$
$\quad\quad \cup\ a^n b^n c \quad (n \geq 1)$
$\quad\quad \cup\ a^n b^{2n} cc \quad (n \geq 1)$

$L(R) = L(D)^*$

		Condition	Action	Go To
			0,1,2,3,4,5,6,7,8,9	
S_0	$\emptyset \rightarrow .R\dashv \quad \dashv \quad S_0$	a	(0,0)	
			(2,0)	
	$R \rightarrow . \quad \dashv a \quad S_0$		(6,0)	
	$R \rightarrow .RD \quad \dashv a \quad S_0$		(7,0)	
	$\emptyset \rightarrow R.\dashv \quad \dashv \quad S_0$		(8,0)	
	$R \rightarrow R.D \quad \dashv a \quad S_0$		(9,0)	S_1
	$D \rightarrow .A \quad \dashv a \quad S_0$			
	$D \rightarrow .A \quad \dashv a \quad S_0$	$\dashv$	(0,1)	S_{33}
	$D \rightarrow .Ccc \quad \dashv a \quad S_0$			
	$A \rightarrow .aAb \quad \dashv ac \quad S_0$			
	$A \rightarrow .ab \quad \dashv ac \quad S_0$			
	$C \rightarrow .aCbb \quad c \quad S_0$			
	$C \rightarrow . \quad c \quad S_0$			

			Condition		Action	Go To
					6,7,8,9	
S_1	A → a.Ab	⊣ ac	S_0, S_{26}			
	A → a.b	⊣ ac	S_0, S_{26}			
	C → a.Cbb	c	S_0, S_{26}	a	(6,0)	
	C → a.bb	c	S_0, S_{26}		(7,0)	
	A → .aAb	b	S_1		(8,0)	
	A → .ab	b	S_1		(9,0)	S_2
	C → .aCbb	b	S_1			
	C → .abb	b	S_1	b	(9,1)	
					(7,1)	S_{31}
					6,7,8,9	
S_2	A → a.Ab	b	S_1, S_2			
	A → a.b	b	S_1, S_2			
	C → a.Cbb	b	S_1, S_2	a	(6,0)	
	C → a.bb	b	S_1, S_2		(7,0)	
	A → .aAb	b	S_2		(8,0)	
	A → .ab	b	S_2		(9,0)	S_2
	C → .aCbb	b	S_2			
	C → .abb	b	S_2	b	(7,1)	
					(9,1)	S_3
S_3	A → ab.	b	S_2, S_1	b, 9=S_2	(8,1)	S_4
	C → ab.b	b	S_2, S_1	b, 9=S_1	(8,1)	S_6
S_4	A → aA.b	b	S_2, S_1	b	(8,2)	
	C → ab.b	b	S_2, S_1		(7,2)	S_5

			Condition	Action	Go To
S_5	A → aAb.	b S_2,S_1			
	C → abb.	b S_2,S_1	b,8=S_2	(8,1)	
			7=S_2	(6,1)	S_8
			b,8=S_1	(8,1)	
			7=S_1	(6,1)	S_{29}
S_6	A → aA.b	⊣ ac S_0,S_{26}	b	(8,2)	
	C → ab.b	b S_2,S_1		(7,2)	S_7
S_7	A → aAb.	⊣ ac S_0,S_{26}			
	C → abb.	b S_2,S_1	⊣ ac,8=S_0,S_{26}	(3,0)	
				(4,0)	S_{27}
			b,7=S_2	(6,1)	S_{16}
			b,7=S_1	(6,1)	S_{19}
S_8	A → aA.b	b S_2,S_1	b	(6,2)	
	C → aC.bb	b S_2,S_1		(8,2)	S_9
S_9	A → aAb.	b S_2,S_1	b,8=S_2	(8,1)	S_{10}
	C → aCb.b	b S_2,S_1	b,8=?	(8,1)	S_{14}
S_{10}	A → aA.b	S_2,S_1	b	6,2)	
	C → aCb.b	b S_2,S_1		(8,3)	S_{11}

			Condition	Action	Go To
S_{11}	$A \rightarrow aAb.$ b	S_2, S_1	$b, 8 = S_2$	(8,1)	
				(6,1)	S_8
	$C \rightarrow aCbb.$ b	S_2, S_1	$6 = S_2$		
			$b, 8 = S_1$	(8,1)	
			$6 = S_1$	(6,1)	S_{12}
S_{12}	$A \rightarrow aA.b \dashv ac$	S_0, S_{26}	b	(8,2)	
	$C \rightarrow aC.bb$ c	S_0, S_{26}		(6,2)	S_{13}
S_{13}	$A \rightarrow aAb. \dashv ac$	S_0, S_{26}			
	$C \rightarrow aCb.b$ c	S_0, S_{26}	$\dashv ac, 8 = S_0, S_{26}$	(3,0)	
				(4,0)	S_{27}
			b	(6,3)	S_{21}
S_{14}	$A \rightarrow aA.b \dashv ac$	S_0, S_{26}	b	(8,2)	
	$C \rightarrow aCb.b$ b	S_2, S_1		(6,3)	S_{15}
S_{15}	$A \rightarrow aAb. \dashv ac$	S_0, S_{26}	$b, 6 = S_2$	(6,1)	S_{16}
	$C \rightarrow aCbb.$ b	S_2, S_1	$b, 6 = S_1$	(6,1)	S_{19}
			$\dashv ac, 8 = S_0, S_{26}$	(3,0)	
				(4,0)	S_{27}
S_{16}	$C \rightarrow aC.bb$ b	S_1, S_2	b	(8,2)	S_{17}
S_{17}	$C \rightarrow aCb.b$ b	S_1, S_2	b	(8,3)	S_{18}

				Condition	Action	Go To
S_{18}	C → aCbb.	b	S_1, S_2	b,8=S_2	(8,1)	S_{16}
				b,8=S_1	(8,1)	S_{19}
S_{19}	C → aC.bb	c	S_0, S_{26}	b	(6,2)	S_{20}
S_{20}	C → aCb.b	c	S_0, S_{26}	b		S_{21}
S_{21}	C → aCbb.	c	S_0, S_{26}	c,6=S_0, S_{26}	(5,0)	S_{22}
S_{22}	D → C.cc	⊣ a	S_0, S_{26}	c	(5,1)	S_{23}
S_{23}	D → Cc.c	⊣ a	S_0, S_{26}	c	(5,2)	S_{24}
S_{24}	D → Ccc.	⊣ a	S_0, S_{26}	⊣ a,5=S_0, S_{26}	(2,1)	S_{25}
S_{25}	R → RD.	⊣ a	S_0	⊣ a,2=S_0	(2,0)	
					(0,0)	S_{26}
S_{26}	Ø → R.⊣	⊣	S_0		3,4,5,6,7,8,9	
	R → R.D	⊣ a	S_0	a	(6,0)	
	D → .A	⊣ a	S_{26}		(7,0)	
	D .Ac	⊣ a	S_{26}		(8,0)	
	D Ccc	⊣ a	S_{26}		(9,0)	S_1
	A → .aAb	⊣ ac	S_{26}			

			Condition	Action	Go To
	A → .ab	⊣ ac S_{26}	⊣	(0,1)	S_{33}
	C → .aCbb	c S_{26}			
	C → .abb	c S_{26}			
S_{27}	D → A.	⊣ a S_0, S_{26}			
	D → A.c	⊣ a S_0, S_{26}	⊣ a,3=S_0, S_{26}	(2,1)	S_{25}
			c	(4,1)	S_{28}
S_{28}	D → Ac.	⊣ a S_0, S_{26}	⊣ a,4=S_0, S_{26}	(2,1)	S_{25}
S_{29}	A → aA.b	⊣ ac S_0, S_{26}	b	(8,2)	
	C → aC.bb	c S_0, S_{26}		(6,2)	S_{30}
S_{30}	A → aAb.	⊣ ac S_0, S_{26}			
	C → aCb.b	c S_0, S_{26}	⊣ ac,8=S_0, S_{26}	(3,0)	
				(4,0)	S_{27}
			b	(6,3)	S_{21}
S_{31}	A → ab.	⊣ ac S_0, S_{26}	⊣ ac,9=S_0, S_{26}	(3,0)	
	C → ab.b	c S_0, S_{26}		(4,0)	S_{27}
			b	(7,2)	S_{32}

		Condition	Action	Go To
S_{32}	C → abb. c .	S_0 c,7=S_0,S_{26}	(5,0)	S_{22}
S_{33}	Ø → R4 . 4	S_0	HALT	

XII. SOME UNDECIDABILITY RESULTS*

Since we have a set of compilable grammars for which a fast recognizer can be compiled, we would like to know (1) Can it be determined for all grammars whether or not they are compilable? and (2) If not, can we at least have an algorithm that terminates if a grammar is compilable and not otherwise? In other words, are the compilable grammars decidable, or at least enumerable. In this section we show that they are neither.

We will use the famous Post correspondence problem [Po 47] to get our results.

Definition: Let $X = (x_1,\ldots,x_w)$, $Y = (y_1,\ldots,y_w)$. The Post correspondence problem P(X,Y) is solvable iff $\exists$ a sequence $i_1\ldots i_y$ $(y \geq 1)$ $(1 \leq i_1,\ldots,i_y \leq w)$ such that $x_{i_1}\ldots x_{i_y} = y_{i_1}\ldots y_{i_y}$.

This problem was shown by Post to be undecidable. It is used in [BPS 64] to show that determining whether the intersection of two context-free grammars is empty is also an undecidable question. We use that same construction to show that this question is also undecidable for compilable grammars. This construction uses the following set of compilable grammars.

*The proofs in this section were suggested by R. W. Floyd.

Definition: The grammar G(X) is as follows

root: $R \rightarrow ab^{i}cx_{i} \mid ab^{i}Rx_{i}$ $(1 \leq i \leq w)$

sentences: $ab^{i_y}\ldots ab^{i_1}c\, x_{i_1}\ldots x_{i_y}$ $(y \geq 1,\ 1 \leq i_1,\ldots,i_y \leq w)$

Definition: $\emptyset$ is the empty set.

Theorem 13: The problem, "Is $L(G_1) \cap L(G_2)$ empty?" is undecidable for G_1 and G_2 compilable grammars.

Proof: G(X) is LR(0) for any w-tuple X as follows: All sentential forms are of the form

$$ab^{i_y}\ldots ab^{i_1}\{c \text{ or } R\}\alpha$$

For some $y \geq 1$, $1 \leq i_1,\ldots,i_y \leq w$, α. The handle ends somewhere in α. The exact place however is determined unambiguously by i_1 (which is obtained by simple counting, since it takes on at most w values). So the handle is determined without looking to its right at all. Therefore the grammar is LR(0) and consequently compilable.

P(X,Y) is solvable $\Leftrightarrow L(G(X)) \cap L(G(Y)) \neq \emptyset$ as follows: If P(X,Y) is solvable, then $\exists\ i_1\ldots i_y$ such that $x_{i_1}\ldots x_{i_y} = y_{i_1}\ldots y_{i_y}$. But then

$ab^{i_y}\ldots ab^{i_1}c\, x_{i_1}\ldots x_{i_y}$ (which $\in L(G(X))$)

$= ab^{i_y}\ldots ab^{i_1}c\, y_{i_1}\ldots y_{i_y}$ (which $\in L(G(Y))$). So the intersection is non-empty. If $L(G(X)) \cap L(G(Y))$ is non-empty, let

$ab^{i_y}\ldots ab^{i_1}c\alpha$ be the common sentence. Then $\alpha = x_{i_1}\ldots x_{i_y}$ and $\alpha = y_{i_1}\ldots y_{i_y}$, so P(X,Y) is solvable.

So if the intersection problem were decidable for compilable grammars, we could decide P(X,Y) as follows: Decide whether or not $L(G(X)) \cap L(G(Y))$ is empty. If it is, then P(X,Y) is unsolvable and vice versa.

But the Post correspondence problem is undecidable.//

Now, using Theorem 13, we can prove that the compilable grammars are undecidable.

Theorem 14: The problem, "Is G compilable?" is undecidable for any grammar G.

Proof: Let G_1 and G_2 be compilable grammars, with roots R_1 and R_2. Consider the grammar $B(G_1,G_2)$:

root: $R \rightarrow A_1A_2$

$A_1 \rightarrow \Lambda | A_1\ c\ R_1$ where $c \notin G_1,G_2$

$A_2 \rightarrow \Lambda | A_2\ c\ R_2$

plus the productions for G_1 and G_2.

sentences: $c\alpha_1\ldots c\alpha_r c\beta_1\ldots c\beta_s$

where $\alpha_1,\ldots,\alpha_r \in G_1$ and $\beta_1,\ldots,\beta_s \in G_2$ $(r,s \geq 0)$

(a) If $\gamma \in L(G_1) \cap L(G_2)$, then $B(G_1,G_2)$ will have the same compilability properties as grammar BP (page 40) with "cγ" replacing "a" in grammar BP. And we observe from the run of BP, that in S_i ($i \geq 1$) we have the states

$$\begin{array}{ll} N \rightarrow Na. & 0 \\ N \rightarrow Na. & 1 \\ \vdots & \\ N \rightarrow Na. & i-1 \end{array}$$

assuring us that the grammar is not compilable.

(b) On the other hand, if $L(G_1) \cap L(G_2) = \emptyset$, then since G_1 and G_2 are compilable and c forms a convenient marker between their sentences, there can be no trouble compiling a recognizer for $B(G_1,G_2)$. The compilation algorithm will compile a recognizer which initially looks for either a G_1 or a G_2 between the c's and as soon as it gets the first G_2, it looks only for G_2's. So $B(G_1,G_2)$ is compilable.

Therefore, $L(G_1) \cap L(G_2) = \emptyset \Leftrightarrow B(G_1,G_2)$ is compilable, and if the compilable question were decidable, we could decide the intersection question for compilable grammars. But this is impossible by Theorem 13.//

We can also prove that the BDA question is undecidable using this same grammar.

Now that we know that the compilable grammars are undecidable, it is reasonable to ask whether or not they are at least enumerable. We use the Post correspondence problem again to show that they are not.

Theorem 15: The set of all solvable Post correspondence problems is enumerable.

Proof: The following algorithm enumerates all solvable Post correspondence problems over a vocabulary V:

Enumerate all triples of integers $\langle w,\ell,y\rangle$. On each one perform the following:

Enumerate all possible pairs of w-tuples $\langle X,Y\rangle = \langle(x_1,\ldots,x_w),(y_1,\ldots,y_w)\rangle$ of strings over V of length $\leq \ell$. There are a finite number of them. For each of these, enumerate all possible y-tuples of integers $(i_1,\ldots,i_y)$ such that $1 \leq i_1,\ldots,i_y \leq w$. There are a finite number of these. If $x_{i_1}\ldots x_{i_y} = y_{i_1}\ldots y_{i_y}$, then output P(X,Y) as a solvable problem.//

Theorem 16: The set of all compilable grammars is not enumerable.

Proof: If we could enumerate all compilable grammars, we would be able to enumerate the unsolvable Post correspondence problems (PCP) as follows:

For each Post correspondence problem P(X,Y), using the proofs of Theorems 13 and 14, we know that ∃ compilable grammars G(X) and G(Y) such that P(X,Y) is unsolvable⇔B(G(X),G(Y)) is compilable.

So enumerate the compilable grammars. As each compilable grammar appears in the enumeration, we test whether it is of the form B(G(X),G(Y)) for some X and Y. If so, we output P(X,Y) as an unsolvable problem.

But by Theorem 15, the solvable PCP's are also enumerable, so the PCP is decidable. This is impossible by [Po 47].//

Since the compilable grammars are not enumerable, we know that we cannot improve much on the compilation algorithm that we have. We certainly can't get it to terminate even on all compilable grammars, for that would be possible only if they were enumerable.

XIII. SPACE

Our algorithm provides no new results in the way of upper bounds on the space required for recognition. The best result in that direction [LHS 65] is $(\log n)^2$, far better than we can do. But on the other hand, this result is only obtained using an extremely slow algorithm. When compared with other fast algorithms, our space results show up quite well.

Theorem 17: There is a bound of the form $Cn^2 + O(n)$ on the number of registers used in computing $REC(G,x_1,\ldots,x_n,k)$ for any k and any grammar G.

Proof: $dmt^k i$ is a bound on the number of states in any state set S_{i-1}. The 3 representations we have for each state set (2,3,4 under implementation, page 26) each require an amount of storage proportional to the number of states in the state set. The storage for the grammar is a constant, and that for the input string is n. So summing the storage over all the state sets:

$$n + \sum_{i=1}^{n+1} dmt^k i = n + dmt^k (n+1)(n+2)/2 \tilde{\approx} dmt^k n^2 //$$

So, in general, the algorithm requires space n^2. This is comparable to Cocke's and Kasami's algorithms, which also require n^2. However, it has one big advantage over Cocke's in that the n^2 is only an upper bound for ours. Cocke's algorithm requires n^2 all the time, while our algorithm very often takes only space n.

In fact one may have noticed that in the state sets that are no longer current, many of the states are no longer useful, and in fact are not accessible at all. The only way that non-current states can be accessed is by the completer, and many of them are either of the wrong form for the completer to access them, or the states which might have caused them to be accessed are no longer current.

Thus we can introduce a "garbage collection" procedure, [Mc 62] page 42, which can be evoked at any time, which will release the storage for those states which are no longer needed. We will specify this procedure only as far as formally defining those states which are still needed at point i in the scan. These are those states for which there exists a sequence of completer operations which could lead to them from a state in S_i.

Definition: A state $s = \langle p,j,f,\alpha \rangle$ is needed at i if $\exists$ a sequence of states $s_0,s_1,\ldots,s_m$ ($s_\ell = \langle p_\ell, j_\ell, f_\ell, \alpha_\ell \rangle$ for $\ell = 0,\ldots,m$) such that $s_0 \in S_i$, $s_m = s$, and $s_\ell \in S_{f_{(\ell-1)}}$ and $C_{p_\ell(j+1)} = D_{p_{(\ell-1)}}$ $(1 \leq \ell \leq m)$.

Under the assumption that this garbage collector is at work, what can we say about the class of space n grammars? They naturally include the time n grammars, for these certainly don't create more than a total of n states in any recognition run. However, because of the garbage collector, quite a few time n^2 grammars will be space n also. We have not been able to characterize them in any meaningful way independent

of the algorithm. However, they include all grammars for which there is a bound on the number of "needed" states in any state set other than the current one.

Grammar PAL (page 65) is an example of a time n^2 grammar which is space n. In examining the run with grammar PAL, we note that in each state set, there is only one state,

$$A \rightarrow x.Ax \quad i-1 \quad \text{in } S_i$$

which can be needed when this state set is no longer current. So when the garbage collector is finished, we will have left the states in the current state set plus one in each of the previous state sets--proportional to n in all. Grammar RL (page 63) is another time n^2 but space n grammar. There are 3 needed states in each non-current state set for this grammar (we have checked ✓ them).

So the space n grammars include the time n grammars plus some more--possibly all unambiguous grammars, but certainly most of them from a practical point of few.

XIV. THE PARSER

We have now investigated the algorithm thoroughly as a recognizer. But we are really interested in parsing for practical purposes. In this section we show how to convert the algorithm into a parser, and we investigate what different properties it has in this form.

The recognizer can be made into a parser in the following way: Each time we perform the completer operation adding a state

$$E \rightarrow \alpha D.\beta \quad g \qquad \text{(ignoring look-ahead)}$$

we construct a pointer from that instance of D in that state to the state

$$D \rightarrow \gamma. \quad f$$

which caused us to do the operation. This indicates that D was parsed as γ. In case that D is ambiguous there will be a set of pointers from it, one for each completer operation which caused

$$E \rightarrow \alpha D.\beta \quad g$$

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 .

In this way, when we reach t -minating state

$$\emptyset \rightarrow R\text{-}1 . \quad 0$$

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.

The following is a more precise description of this parsing algorithm.

Definition: A parse tree is a directed graph with parse states at the nodes. A parse state is a 5-tuple $\langle p,j,f,\alpha,B\rangle$, where p,j,f and α are as in a state and B is a sequence consisting of terminal symbols and sets of pointers to other parse states. The elements of the sequence are $B_1 \ldots B_j$. If C_{ph} is terminal ($1 \leq h \leq j$), then $B_h = C_{ph}$. If C_{ph} is non-terminal, then B_h is a set of pointers to other parse states. Each of these pointers represents a different possible derivation proceeding from C_{ph}; specifically, if a parse state pointed to from a parse state in S_i is $\langle q,\bar{q},\ell,\beta,B'\rangle$, then $D_q = C_{ph}$ and the production $C_{ph} \rightarrow C_{q1} \ldots C_{q\bar{q}}$ is first in some derivation of $X_{\ell+1} \ldots X_i$ from C_{ph}, in a derivation of the entire input string. The root of the parse tree is the parse state $\langle 0,2,0,\dashv^k,B\rangle \in S_{n+1}$.

The Parser

Add the following to the recognition algorithm:

(1) In the predictor, let the parse state to be added be $\langle q,0,i,\beta,\Lambda\rangle$. In the completer and scanner, let the first ℓ or j elements of B of the parse state be carried along to the newly created

parse state. In the completer, let $B_{\ell+1}$ be a pointer to $\langle p,\bar{p},f,\alpha,B\rangle$. In the scanner, let $B_{j+1} = X_{i+1}$.

(2) When a parse state $\langle p,j,f,\alpha,B_1 \ldots B_j\rangle$ is <u>added</u> to a state set, if the set already contains a parse state $\langle p,j,f,\alpha,B'_1 \ldots B'_j\rangle$, then we replace that state with

$$\langle p,j,f,\alpha,(B_1 \cup B'_1)\ldots(B_j \cup B'_j)\rangle$$

<u>Implementation</u>

We store the sets of pointers B_i as linked lists. By the proof of lemma 1, we can get a pointer into a set B_i in only one way, so we can unite the lists (in (2) above) by joining the tail of one to the head of another. This takes a fixed amount of time independent of the size of the lists.

Now we ask, does the parser have any different time bounds than the recognizer? The answer is no. The following theorem proves this for the general case, but it is clear from the proof that it applies to the time n^2 and n results also.

<u>Theorem 18</u>: There is a bound of the form $Cn^3 + O(n^2)$ on the time required to parse $X_1 \ldots X_n$ with respect to any grammar G.

Proof: The only difference between recognition (Theorem 5) and parsing occurs in the completer when we are adding a parse state which already exists in the state set. This may take up to j steps to unite the

lists $B_1 \ldots B_j$. But j is bounded by m. This increases by 1 the exponent of m in the final bound.//

The parser does have different space requirements than the recognizer. This is because space is required to store the factored parse trees. We can derive an n^3 space bound, however, by observing that the time is n^3.

Theorem 19: There is a bound of the form $Cn^3 + O(n^2)$ on the space required to parse $X_1 \ldots X_n$ with respect to any grammar G.

Proof: It takes one operation to use up one storage location, so we get this result by Theorem 18.//

This result indicates that we are able to store in space n^3 (in a factored form) all the parse trees for any grammar and string, even ones which have an exponential number of parses (grammar UBDA, page 42, is such a grammar). Even more remarkable, our parser automatically builds a factored representation of even an infinite number of parses. Grammar INF (page 110) illustrates this. We have circled the last 5 states so they can be pointed to. Notice that R → R.A 0 is not part of the final parse tree (since it is not a final state) even though it was important in getting the parse tree formed.

We also notice that for BDA grammars, the space results for the parser are the same as those of the recognizer. This is because a state can be added in at most a bounded number of ways by the completer, and therefore the additional space required for its parses will be

Grammar INF

$R \rightarrow A \mid RA$

$A \rightarrow x \mid \Lambda$

Sentences: x^n $(n \geq 0)$

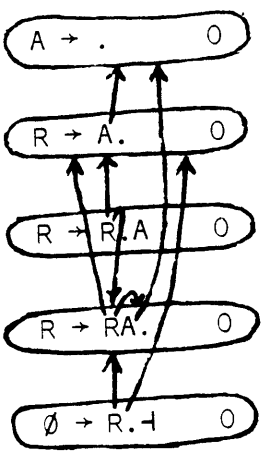
PARSE(INF, ,0):

S_0 $\emptyset \rightarrow .R\dashv$ 0

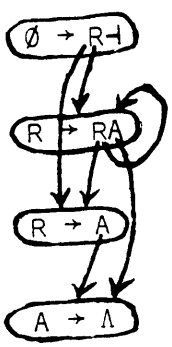
$R \rightarrow .A$ 0

$R \rightarrow .RA$ 0

$A \rightarrow .x$ 0



Parse Trees:



bounded. Thus the parsing space for BDA grammars is n^2 and the space n grammars are the same for the parser and recognizer.

We have not specified how to convert the compiled version of the algorithm into a parser, but this is done in almost exactly the same way as for the original algorithm.

The algorithm we have described in this section provides a way of constructing the parse tree as the analysis is done. An alternative is to run the algorithm through as a recognizer, and then reconstruct each parse from the state sets afterwards. This can be done in a straightforward manner.

XV. TWO MODELS

In this section we describe in more detail the random access model of a computer that we have been using implicitly throughout this paper. We also discuss the possible use of a Turing machine model and the results obtained under this different formalism.

The Random Access Machine (ram)

This model has an unbounded number of registers (counters) each of which may contain any non-negative integer. These registers are named (addressed) by successive non-negative 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.

(3) Add 1 or subtract 1 from the contents of a register (taking $0 - 1 = 0$).

The control for this model is a normal finite state device. The most important property of this machine is that in the above three 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) is what 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.

We will not specify this model in any more detail. It is clear from the above how an appropriate formalism could be developed. It is our opinion that this model represents most accurately the properties of real computers which are relevant to syntax analysis.

The Bounded Activity Machine

Another model, which is frequently used in studies of computational complexity, is a bounded activity machine (bam). This is a Turing machine, which has a fixed number of tapes which may be multi-dimensional and a fixed number of heads. It is a weaker model than the random access machine in that any algorithm can be done at least as fast on a ram as on a bam. The converse is not true. This is because a random access machine can access anything in its memory in one step (if it has the address), while a bam must move one of its heads to the appropriate position first. This could take proportional to n steps in some cases.

Since the bam is a frequently used model, it is reasonable to ask if we can obtain the same results for our algorithm if it is implemented on a bam. We are unable to obtain any of our results by a "direct" encoding of the algorithm onto a bam. All the times get increased by at least a factor of n in this way. We have been able to obtain the n^3 result on a two-dimensional, fixed head Turing machine by making a small change in the algorithm to fit the bam structure, but we have been unable to obtain the n^2 or n results at all on a bam.

This coincides quite closely with the published results. Cocke's n^3 algorithm does work on a bam. Kasami's n^2 algorithm for unambiguous grammars works on a ram only. Knuth's algorithm is able to compile a push down automata (certainly a bam) for LR(k) grammars, but this is not a general context-free algorithm. So our algorithm is as good as other algorithms but no better on a bam.

XVI. EMPIRICAL RESULTS

We have programmed the algorithm and tested it against the top-down and bottom-up parsers evaluated by Griffiths and Petrick [GP 65]. These are the back-tracking algorithms (section III) whose time requirements can be exponential. However, they also can do well on some grammars, and they have both 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 non-deterministic 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). One can see from the data that this is comparable to what Griffiths and Petrick use.

We compare the algorithms on 7 different grammars. We did not use 2 of their examples because the exact grammar they used was not given. For the first four, Griffiths and Petrick were able to find closed form expressions for their results, so we did also (page 118). 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. Therefore we will compare our algorithm to SBU only. We used our algorithm with $k = 0$. The two are

comparable on G1, G2, and G3, the simple grammars, but on G4, which is very ambiguous, ours is clearly superior--n to n^3.

For the next 3 grammars we present only the raw data (pages 119-121). We have also included the data from [GP 65] on PA, the predictive analyzer without path elimination [KO 63]. On the propositional calculus grammar, PA seems to be running in time n^2 while both SBU and ours are 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 back-tracking algorithms. It performs as well as the best of them on all seven grammars, and is substantially faster or ome.

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 in this paper that our algorithm achieves time bounds which are as good as or better than those of any of these algorithms. 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 [KT 67] 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 very curious about the class of grammars that it can parse in time n.

G1

root: $S \rightarrow Ab$

$A \rightarrow a \mid Ab$

G2

root: $S \rightarrow aB$

$B \rightarrow aB \mid b$

G3

root: $S \rightarrow ab \mid aSb$

G4

root: $S \rightarrow AB$

$A \rightarrow a \mid Ab$

$B \rightarrow bc \mid bB \mid Bd$

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	$\sim 2^{n+6}$	$\sim 2^{n+2}$	$\sim 2^{n+5}$	$(n^3+21n^2+46n+15)/3$	$18n+8$

Propositional Calculus Grammar

root: F → C|S|P|U

C → U ⊃ U

U → (F)|∼U|L

L → L'|p|q|r

S → U ∨ S|U ∨ U

P → U ∧ P|U ∧ U

Sentence	Length	PA	SBU	Ours
p	1	14	18	28
(p ∧ q)	5	89	56	68
(p' ∧ q) ∨ r ∨ p ∨ q'	13	232	185	148
p ⊃ ((q ⊃ ∼(r' ∨ (p ∧ q))) ⊃ (q' ∨ r))	26	712	277	277
∼(∼(p' ∧ (q ∨ r) ∧ p'))	17	1955	223	141
((p ∧ q) ∨ (q ∧ r) ∨ (r ∧ p')) ⊃ ∼((p' ∨ q') ∧ (r' ∨ p))	38	2040	562	399

Grammar GRE

root: $X \rightarrow a|Xb|Ya$

$Y \rightarrow c|YdY$

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

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

XVII. THE PRACTICAL USE OF THE ALGORITHM

So far in this paper, we have introduced an algorithm (and a couple of variations of it), and we have studied in detail its formal properties. Now we ask the questions, what is the practical value of the algorithm? in what areas and in what form can it best be put to use?

As we mentioned in the introduction, the two main uses of context-free languages are in natural language processing and in programming language implementation. We will restrict our attention to the latter area because we are not familiar with the practical aspects of computational linguistics, and because it is compiler work which originally motivated us to study parsing.

First we ask, what impact will our algorithm have on the parsing done in production compilers for existing programming languages? The answer is, practically none. Production compilers require -essing time proportional to n with a fairly low coefficient of n. if the syntax of the language is fixed and the available programming effort is large, the parsing can be hand coded for the particular language. This will probably make it faster than or at least as fast as our algorithm or in fact any automatic parser.

The value of any parsing algorithm is for use in compiler-writing languages [BM 62] or similar devices where the programmer is experimenting with the syntax of the language as he implements it, or wants to be able to implement many different languages easily, or for some other reason can't afford to code the parser by hand. Here he is

willing to give up a little in speed to obtain the advantages of having to supply only the BNF of the language in order to get a parser.

It is here that many of the time n algorithms which we mentioned in Section III are used. Most of these, except for Knuth's algorithm, have been used (or have been tried) in compiler systems of one form or another. They all share at least one important drawback. Most programming languages do not readily fall into the class of grammars which the algorithm will handle, and consequently much fiddling must be done with the grammar to get the parser to accept it. This seems not to be the case with the LR(k) algorithm. It seems to be the case that most programming languages naturally have LR(1) grammars, or maybe LR(2) in a few places. And the one programming language which we have discovered that is not LR(k) (ignoring languages which are ambiguous) is compilable by our algorithm quite easily.

So it seems that the compiled version of the LR(k) algorithm and the compiled version of ours which is more or less an extension of the LR(k), hold some promise of alleviating this problem of having to fiddle with the grammar. The coefficient of n is also reasonable for these algorithms. It is very low for the LR(1) algorithm; we estimate that it is about as good as a hand-coded parser. And given the pruning process, the coefficient for our compiled algorithm may go up only as much as the grammar deviates from being LR(1).

It is probably important to say here that we recommend using a look-ahead of 1 with both compiled algorithms. Implementing the full look-ahead for all k would cost more in programming time and efficiency than could be gained from the off chance grammar which might require a

large k. It might be worthwhile to implement the LR(k) algorithm to include a look-ahead of 1 or 2, but no more than this is needed. And a look-ahead of 1 at the most is needed for our algorithm, because it can compile a much larger class of grammars than the LR(k) algorithm, so the extra look-ahead is much less likely to be needed.

This points out a possible advantage of our algorithm over just the LR(k) algorithm, even if we are working only with LR(k) grammars. If we have a grammar which is basically LR(1) everywhere except in one or two "places" (Algol is this way), we would have to do one of three things if we are using only the LR(k) algorithm:

(1) Change the grammar so it is LR(1) everywhere.

(2) Use an LR(2) parser. This, however, means losing efficiency everywhere because of the 2 character look-ahead.

(3) Modify the LR(k) algorithm so that part of a grammar could be done with a look-ahead of 1 and part with 2. This has not been done.

However, if we are using our algorithm, we simply run it with a look-ahead of 1, and the few places where a look-ahead of two would have been needed by an LR(k) parser will be taken care of automatically with only as much loss of efficiency as is necessary. Grammar LR2 (page 125) is such an LR(2) grammar. It in fact reflects the problem which makes some Algol grammars LR(2). Think of production 1 as an assignment statement and production 2 as allowing an arbitrary number of labels on such a statement.

Grammar LR2

root: $R \rightarrow {}^{1}1: = 1|^{2}L:R$

$L \rightarrow {}^{3}1$

Sentences: $\{1:\}^{n} = 1 \ (n \geq 1)$

			Condition	Action	Go To
S_0	$\emptyset \rightarrow .R\dashv$	$\dashv$ S_0		0,1,2,3	
	$R \rightarrow .1:=1$	$\dashv$ S_0	1	(1,0)	
	$R \rightarrow .L:R$	$\dashv$ S_0		(3,0)	S_1
	$L \rightarrow .1$	: S_0			
S_1	$R \rightarrow 1.:=1$	$\dashv$ S_0,S_3			
	$L \rightarrow 1.$	: S_0,S_3	:,3=S_0	(2,0)	S_2
			:,3=S_3	(2,0)	S_2
S_2	$R \rightarrow 1.:=1$	$\dashv$ S_0,S_3	:	(1,1)	
	$R \rightarrow L.:R$	$\dashv$ S_0,S_3		(2.1)	S_3
S_3	$R \rightarrow 1:.=1$	$\dashv$ S_0,S_3		1,2,3	
	$R \rightarrow L:.R$	$\dashv$ S_0,S_3	=	(1,2)	S_4
	$R \rightarrow .1:=1$	$\dashv$ S_3	1	(1,0)	
	$R \rightarrow .L:R$	$\dashv$ S_3		(3,0)	S_1
	$L \rightarrow .1$	: S_3			

		Condition	Action	Go To
S_4	$R \rightarrow 1:=.1 \quad \dashv \quad S_0, S_3$	1	(1,3)	S_5
S_5	$R \rightarrow 1:=1 \quad \dashv \quad S_0, S_3$	$\dashv, 1=S_0$	(0,0)	S_7
		$\dashv, 1=S_3$	(2,2)	S_6
S_6	$R \rightarrow L:R. \quad \dashv \quad S_0, S_3$	$\dashv, 2=S_0$	(0,0)	S_7
		$\dashv, 2=S_3$	(2,2)	S_6
S_7	$\emptyset \rightarrow R.\dashv \quad \dashv \quad S_0$	$\dashv$	(0,1)	S_8
S_8	$\emptyset \rightarrow R\dashv \quad \dashv \quad S_0$		HALT	

Pruned Code

	Condition	Action	Go To
S_0		1,2	
	1	(1,0)	
		(2,0)	S_2
S_2	:	(1,1)	
		(2,1)	S_3
S_3		1,2	
	=	(1,2)	S_4
	1	(1,0)	
		(2,0)	S_2
S_4	1	(1,3)	S_5
S_5	$\dashv,1=S_0$	HALT	
	$\dashv,1=S_3$	(2,2)	S_6
S_6	$\dashv,2=S_0$	HALT	
	$\dashv,2=S_3$	(2,2)	S_6

Grammar EXP(n)

$R \rightarrow A_i$ $\quad (1 \leq j \leq n)$

$A_i \rightarrow c_j A_i$ $\quad (1 \leq i \neq j \leq n)$

$A_i \rightarrow c_i B_i \mid d_i$ $\quad (1 \leq i \leq n)$

$B_i \rightarrow c_j B_i \mid d_i$ $\quad (1 \leq i,j \leq n)$

Example: EXP(2)

$R \rightarrow A_1 \mid A_2$

$A_1 \rightarrow c_2 A_1 \mid c_1 B_1 \mid d_1$

$A_2 \rightarrow c_1 A_2 \mid c_2 B_2 \mid d_2$

$B_1 \rightarrow c_1 B_1 \mid c_2 B_1 \mid d_1$

$B_2 \rightarrow c_1 B_2 \mid c_2 B_2 \mid d_2$

There is one large drawback to both the LR(k) algorithm and our algorithm in their compiled forms, however. That is that in generating all possible state sets at compile time, we may have to generate such a large number that we exceed the space of the machine on large grammars. This can be stated more precisely as follows: The space required for the recognition process is proportional to n, but in the compiled version of the algorithm, the space used depends on the size of the grammar as well as on the size of the string. And we have no guarantee that it depends linearly on the size of the grammar. In fact, page 128 contains a class of LR(0) grammars discovered by John Reynolds for which the number of state sets generated by Knuth's compilation algorithm goes up exponentially with the size of the grammar.

Furthermore, [Ko 67] reports that in trying Knuth's compilation algorithm on a grammar for Algol, the number of state sets generated blew up. He presents a method for factoring an LR(1) grammar in pieces, applying the compilation algorithm to each of these, and then connecting the pieces into one parser. He claims that this cuts down on the number of state sets generated. We suspect that there are other small modifications in both Knuth's and our compilation algorithms which will guarantee a linear growth in the number of state sets with respect to the grammar for a large practical set of grammars. However, until this is accomplished, the compiled versions of these algorithms will not be of any practical value.

Now, what about the original version? It is obviously useful in compiler-writing systems where the speed of the algorithm is not as

important as the fact that it will accept any grammar without modifications and won't run too slowly on any of them. Here it is a matter of choice. Until the problems which we have mentioned are removed from Knuth's and our compiled algorithms, one has the choice between a large coefficient of n or a large amount of fiddling needed to get the grammar accepted by the recognizer.

There is another use to which the original version is even better suited. That is the extendible language systems [GP 67] which are being built now. These systems have the property that the syntax of the source language may be changed or extended as part of the program itself. Many of them allow syntax extensions to be made at any point in the program, so as soon as some of them become conversational [St 68], then syntax changes will be made capriciously and frequently.

These systems will not tolerate a costly compilation of a full parser each time a change is made in the syntax. The time n algorithms which are in use now all require such compilations (in one form or another) and unless they can be altered to allow incremental recompilation when small changes are made, they may be useless in extendible systems. Thus if such systems want to allow full syntax extension, they may have to go to non-compiled algorithms. In this class ours seems to be the best. It will almost certainly do all the practical grammars it gets in time n and if it does get any n^2 or n^3 grammars, it will do at least as well as any other algorithm. The results in Section XVI also indicate that our constant coefficients are as good as or better than other algorithms. Krutar [Kr 68] is currently implementing such an extendible language system using our algorithm.

We should mention here one possible disadvantage of either version of our algorithm that is not shared by the time n algorithms. Our algorithm, or any of the others, may be caused to output a parse tree for the string it has recognized when it is finished. And in fact, some compiler-writing systems do work this way; the output of the parsing phase is a tree, which is then processed by the next phase of the compiler. Some other systems [BM 62], however, do not build up the whole parse tree before the next phase is called, but build up pieces of the tree and pass these on to the next phase, which works on them before the parsing can continue. Some systems even call the next phase every time a parsing decision is made [Fe 66].

All the time n algorithms (including Knuth's) are set up so that every time a parsing decision is made, it is indeed going to be part of the final parse tree for the sentence. This is not true of our algorithm. Because of the fact that we carry along all possible parses as we go, many parsing decisions (completer operations) are made which later turn out to be extraneous. For this reason, we can't afford to call the next phase every time a parsing decision is made.

This problem can be remedied in the following way: Any time that a completer operation is to be performed, it can be determined directly whether or not it is known then that this decision will be part of the final parse. If the final state causing the completer is the only one in its state set which matches the next k symbols, then it will be part of the final parse. If there are two or more such states, then the states that they produce by the completer can be marked and the parsing can continue. Later, as soon as a condition is reached where the

current state set contains only states which were produced by one of those final states, then that final state is the one which represented the correct parsing decision, and the next phase can be called for it.

Even without this marking procedure, we can guarantee that all parsing decisions will be correct at the time they are made for at least all LR(k) grammars for which we use a look-ahead of k.

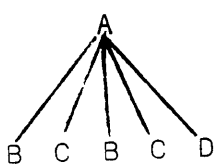
Our algorithm has the useful property that it can be modified to handle an extension of context-free grammars which contains the Kleene star notation. In this notation,

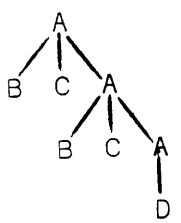
$$A \rightarrow \{BC\}^*D$$

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

$$A \rightarrow D|BCA$$

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 the algorithm involves two additional operations:

(1) Any state of the form

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

is replaced by

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

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

(2) Any state of the form

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

is replaced by

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

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

Another possible advantage of our algorithm over the time n algorithms (including Knuth's) is that we can handle ambiguous grammars, and in fact handle many of them in time n. Syntactic ambiguity can be valuable in compiler-writing languages to represent constructs which are syntactically ambiguous, but which can be disambiguated easily by using semantic considerations. All of the time n algorithms work only on subsets of the unambiguous grammars.

We should mention here that one may not want to implement the algorithm exactly as we have specified (Section VI). There are certainly other implementations which still give the quoted time and space bounds. Our implementation is merely to show that it is possible to achieve these bounds.

XVIII. FURTHER RESEARCH AND CONCLUSION

Some formal work still remains to be done on this algorithm:

(1) Find a good algorithm-independent characterization of the time n grammars. Show that it includes bounded state and U LR(k) grammars.

(2) Do the same for the space n grammars; do they include the unambiguous grammars?

(3) Prove our conjecture (Section VII) that there is a language with inherent unbounded direct ambiguity, and that there is a language with inherent unbounded ambiguity which is BDA.

Much practical testing still is needed:

(1) Implement and test the garbage collector (Section XIII). Are there any space problems with it in use?

(2) Implement and test the compilation algorithm and the pruning procedure. In a practical sense, how large is the set of compilable grammars? What problems are there because the algorithm may loop? What is the coefficient of n like in practice.

(3) Imbed either the compiled or the original algorithm in a compiler-writing or extendible-language system. What are the interface problems?

Further research in the general field of context-free parsing might center on a number of different things. It may be possible to develop a general algorithm with an n^2 time bound. It seems unlikely that a general algorithm with a time n bound exists, although it has not been proven impossible. It would be worthwhile to try to extend

the class of grammars which can be done in time n, possibly to include all unambiguous grammars.

For the researcher in automata theory or complexity theory, it would be interesting to try to obtain our n^2 or n results on a bam, or to try to prove that a general time n algorithm is impossible (for a bam at first try anyway).

We discussed some compiler motivated research problems in Section XVII. A modification for Knuth's (or our) compilation algorithm is needed which cuts down on the growth of the space with respect to the size of the grammar. For use in conversational extendible languages, an incremental recompilation feature would be very valuable. Perhaps a completely different compilation algorithm should be devised which either overcomes some of the problems of the others or covers a wider class of grammars.

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 to have specified to it the class of grammars it is operating on. In other words, Knuth's algorithm works only on LR(k) grammars and Kasami's only on unambiguous ones, but ours works on them all and seems to do about as well as it can automatically.

Thus we have developed a context-free parsing algorithm which is quite efficient in the formal sense, and also shows promise of being very useful in several applications related to the implementation of programming languages.

REFERENCES

[BM 62] Brooker, R. A. and Morris, D. A general translator program for phrase structure languages. Journal ACM 9, Jan. 1962, p. 1.

[BPS 64] Bar-Hillel, Y., Perles, M. and Shamir, E. On formal properties of simple phrase structure grammars. In Language and Information, Y. Bar-Hillel, Addison-Wesley, Reading Mass., 1964.

[Fe 66] Feldman, J. A. A formal semantics for computer languages and its application in a compiler-compiler. Comm. ACM 9, Jan. 1966, pp. 3-9.

[Fl 63] Floyd, R. W. Syntactic analysis and operator precedence. Journal ACM 10, July 1963, pp. 316-333.

[Fl 64a] Floyd, R. W. Bounded context syntax analysis. Comm. ACM 7, Feb. 1964, pp. 62-66.

[Fl 64b] Floyd, R. W. The syntax of programming languages--a survey. IEEE Transactions on Electronic Computers, Vol. EC-13, No. 4, Aug. 1964.

[GP 65] Griffiths, T. and Petrick, S. On the relative efficiencies of context-free grammar recognizers. Comm. ACM 8, May 1965, pp. 289-300.

[GP 67] Galler, B. A. and Perlis, A. J. A proposal for definitions in Algol. Comm. ACM 10, April 1967, pp. 204-219.

[Gr 66] Griffiths, T. V. An upper bound on the time required for parsing by predictive analysis with abortive path elimination. Fourth Annual Meeting of the Association for Machine Translation and Computational Linguistics, UCLA, July 26-27, 1966.

[Ha 62] Hays, D. Automatic language-data processing. In Computer Applications in the Behavioral Sciences, H. Borko, ed. Prentice Hall, Englewood, New Jersey, 1962.

[Ir 61] Irons, E. A syntax directed compiler for Algol 60. Comm. ACM 4, Jan. 1961, p. 51.

[Ka 66] Kasami, T. An efficient recognition and syntax analysis algorithm for context free languages. University of Illinois, 1966.

[Ka 67] Kasami, T. A note on computing time for recognition of languages generated by linear grammars. Information and Control 10, 1967, pp. 209-214.

[Kn 65] Knuth, D. E. On the translation of languages from left to right. Information and Control 8, 1965, pp. 607-639.

[KO 63] Kuno, S. and Oettinger, A. G. Multiple path syntactic analyzer. Information Processing 62, C. M. Popplewell, ed., North Holland, Amsterdam, 1962-1963.

[Ko 67] Korenjak, A. J. A practical approach to the construction of deterministic language processors. RCA Laboratories, Princeton, New Jersey.

[Kr 68] Krutar, R. Unpublished. Computer Science Department, Carnegie-Mellon University.

[KT 68] Kasami, T. and Torii, K. Some results on syntax analysis of context free languages. Record of Technical Group on Automata Theory of Institute of Electronic Communication Engineers, Japan, Jan. 1967.

[Ku 65] Kuno, S. The predictive analyzer and a path elimination technique. Comm. ACM 8, 453-463.

[LHS 65] Lewis, P. M., Hartmanis, J. and Stearns, R. E. Memory bounds for recognition of context-free and context-sensitive languages. IEEE Conference Record on Switching Circuit Theory and Logical Design, IEEE Pub. 16C13, 1965, pp. 191-202.

[Ma 68] Martin, W. A. A left to right then right to left parsing algorithm. Project MAC, MIT, Feb. 1968.

[Mc 62] McCarthy, J., Abrahams, P., Edwards, D., Hart, T. and Levin, M. LISP 1.5 Programmers Manual. MIT Press, Cambridge, Mass., 1962.

[Na 63] Naur, P., ed. Revised Algol report. Comm. ACM 6, Jan 1963, pp. 1-17.

[Po 47] Post, E. L. Recursive unsolvability of a problem of Thue. Journal of Symbolic Logic 12, pp. 1-11.

[St 68] Standish, T. A preliminary sketch of a polymorphic programming language. Computer Science Department, Carnegie-Mellon University, June 1968.

[Ul 65] Ullman, J. Pushdown automata with bounded backtrack. SDC, Santa Monica, California, 1965.

[Wi 68] Williams, J. Unpublished. Computer Science Department, University of Wisconsin.

[WW 66] Wirth, N. and Weber, H. EULER: A generalization of ALGOL and its formal definition: Part I. Comm. ACM 9, Jan. 1966, pp. 13-23.

[Yo 66] Younger, D. H. Context free language processing in time n^3. G.E. Research and Development Center, Schenectady, New York, 1966.

[Yo 67] Younger, D. H. Recognition and parsing of context free languages in time n^3. Information and Control 10, 1967, pp. 189-208.