An Improved Context-Free Recognizer

SUSAN L. GRAHAM, MICHAEL A. HARRISON, and WALTER L. RUZZO
University of California at Berkeley

A new algorithm for recognizing and parsing arbitrary context-free languages is presented, and several new results are given on the computational complexity of these problems. The new algorithm is of both practical and theoretical interest. It is conceptually simple and allows a variety of efficient implementations, which are worked out in detail. Two versions are given which run in faster than cubic time. Surprisingly close connections between the Cocke–Kasami–Younger and Earley algorithms are established which reveal that the two algorithms are "almost" identical.

Key Words and Phrases: parsing, context-free grammars, dynamic programming, data structures
CR Categories: 3.42, 4.12, 4.34, 5.23, 5.25

1. INTRODUCTION

Since the introduction of context-free languages and grammars in the late 1950s there has been considerable interest in efficient recognition and parsing algorithms for them. Good linear-time algorithms are now known for many subclasses of context-free grammars, but these methods are too restricted for some applications. In these situations, general context-free language recognition and parsing algorithms are used. In the present paper a new, general, context-free recognizer is presented. Efficient techniques are given to implement it, and some unexpected connections between previously known algorithms are derived. Both theoretical and practical analyses of these methods are given. The analysis considers not only orders of magnitude, but also the constant multipliers which are commonly ignored in theoretical studies but are important in practical situations.

One significant use of the general context-free methods is as part of a system of processing natural languages such as English. We are not suggesting that there is a context-free grammar for English. It is probably more appropriate to view the grammar/parser as a convenient control structure for directing the analysis

The editor-in-chief did not participate in the consideration of this paper for publication; it was processed by the associate editor, M.D. McIlroy.
This research was supported by the National Science Foundation under Grants MCS74-07644-A03 and MCS74-07636-A01, and by an IBM Postdoctoral Fellowship. A preliminary version of this paper [13] was presented at the Eighth Annual ACM Symposium on Theory of Computing, 1976.
Authors' present addresses: S.L. Graham and M.A. Harrison, Computer Science Division, University of California, Berkeley, CA 94720; W.L. Ruzzo, Department of Computer Science, University of Washington, Seattle, WA 98195.

of the input string. The overall analysis is motivated by a linguistic model which is not context free, but which can frequently make use of structures determined by the context-free grammar.

Other applications that have been proposed for general context-free methods include extensible programming languages. Many programming languages have grammars which are suitable for parsing by one of the linear-time methods. However, in some extensible languages the grammar is not fully known to the compiler designers, and a parsing method is needed which can efficiently accommodate a growing grammar. Even if one of the linear-time parsers satisfied this condition, it might still be difficult or impossible for the user to extend the language at will and still keep the grammar in the form required by that particular parsing method, since the user would not be free to tinker with the whole grammar, only the additions. The general context-free methods we discuss later not only free the user from all concern about the form of the grammar, but also can easily handle additions to the grammar. (See, for example, [18, 26, 37].)

Another area where general context-free parsing techniques have been considered is speech recognition. Here the input language can only be approximately defined, and individual inputs can vary widely from the norm. Thus the goal is to find a parse which most closely matches the input. Ambiguity arises, since each unit of the input can be considered to be a "distorted" version of any of several possible sounds with various probabilities (for example, see [23]). A closely related technique has also been proposed for doing error correction while parsing [24].

The first context-free parsers used "backtracking" to search exhaustively for a derivation matching the input string. While these parsers worked reasonably well in some cases, they had a worst-case running time which grew exponentially in the length of the input, making them unsuitable for most applications. References to many of these methods and comparisons of them may be found in [14, 22].

There are two well-known, practical, general context-free recognition methods. The first was discovered independently by Cocke, Kasami, and Younger early in the 1960s [16, 20, 39]. It is essentially a dynamic programming method and takes time proportional to n^3, where n is the length of the input string. The method requires a grammar in Chomsky normal form, but since every context-free language has such a grammar, this is not a fundamental restriction. More recently, Valiant showed that the computation performed by the Cocke–Kasami–Younger algorithm can be related to Boolean matrix multiplication, giving a recognizer running in subcubic time[1] [36]. For sufficiently long inputs this is the fastest known method. However, the overhead for this method is too large to make it useful for values of n in the range of practical interest.

The second major method was discovered by Earley as an extension of Knuth's LR(k) method [7, 8, 21]. Unlike the Cocke–Kasami–Younger algorithm, Earley's algorithm will work for any context-free grammar.

One of the main contributions of this paper is a new general context-free language recognizer, presented in Sections 2 and 3. It is derived from Earley's algorithm, but has a number of advantages over its predecessor. First, and most

[1] Recently, substantial progress has been made on this problem. As of July 1980, methods are known which multiply two $n \times n$ matrices in time proportional to $n^{2.55\cdots}$.

important, it is conceptually simpler than Earley's method. This simplicity has helped reveal a variety of possible implementations and optimizations which make the algorithm applicable in a wide range of circumstances. One simple implementation of our algorithm uses proportional-to-n^2 bit-vector operations on vectors of length n, where n is again the length of the input. For problems of practical size this method may be much faster than Earley's, since word-parallel Boolean operations may be used to manipulate the bit vectors (which would fit in a few words on most computers). Of theoretical interest are versions of the algorithm which run in less than n^3 steps. However, the overhead for these methods may be too large for practical applications. Readers interested in the space complexity of context-free recognition should consult [15, 31, 32, 33].

A second contribution of this paper is to exhibit deep and perhaps unexpected connections between the various parsing methods mentioned above. In particular, we show that the Cocke–Kasami–Younger and Earley methods are "almost" identical. This unification of superficially dissimilar methods should simplify the field. These results are discussed in Section 4.

Throughout the paper, we assume that the reader is familiar with the basic concepts and notation from language theory, such as context-free grammars (cfg), context-free languages (cfl), derivation, derivation tree, and Chomsky normal form grammar; precise definitions may be found in any standard text on the subject, such as [2, 15].

2. THE NEW ALGORITHM

Throughout this section we assume the presence of an arbitrary context-free grammar $G = (V, \Sigma, P, S)$, where V is the total vocabulary, Σ is the set of terminal symbols, P is the finite set of productions, and S is the start symbol. We let $N = V - \Sigma$ denote the set of *variables* or *nonterminals*. It is also convenient to fix $n \geq 1$, to fix the input string to be parsed as $w = a_1 a_2 \cdots a_n$, and, for $1 \leq i \leq n$, $a_i \in \Sigma$. Further, w_i and $w_{i,j}$ for $0 \leq i \leq j$ denote the substrings $a_1 a_2 \cdots a_i$ and $a_{i+1} \cdots a_j$, respectively ($w_0 = w_{i,i} = \Lambda$, the empty string). The *length* of a string w, written $\lg(w)$, is defined to be the number of occurrences of symbols in w. Note that $w_i w_{i,j} = w_j$ and $w_{i,k} w_{k,j} = w_{i,j}$. Also, we define the *size* of G, written as $|G|$, to be $\sum_{A \to \alpha \text{ in } P} \lg(A\alpha)$.

The *recognition problem* is to decide whether or not w is in $L(G)$. A *recognizer* is a procedure which *accepts* (recognizes) those strings in $L(G)$ and rejects all others. A recognizer is said to operate *on-line* if it recognizes each prefix of w before reading any of the input beyond the prefix. More formally, an on-line recognizer is a procedure which sequentially reads its input $a_1 a_2 \cdots a_n$ and sequentially generates an output sequence of 0's and 1's, $r_0 r_1 r_2 \cdots r_n$, where r_i is generated before a_{i+1} is read, and r_i is 1 if $a_1 \cdots a_i$ is in $L(G)$ and 0 otherwise. A recognizer which is not on-line is called *off-line*.

A *parser* is a recognizer which additionally outputs a parse or derivation of each accepted input. The parse may be encoded in a variety of ways, for example, as the list of productions used in a rightmost derivation or as a derivation tree; we will not be concerned with the representation at this time.

It is both necessary and convenient to present an old algorithm first, the so-called Cocke–Kasami–Younger algorithm (hereafter called *CKY*), which was

discovered independently in the mid 1960s by J. Cocke, T. Kasami, and D. Younger [20, 39]. Modern presentations of the algorithm may be found in [2, 12, 15].

Convention. For the moment we assume that the grammar G is a Chomsky normal form grammar.

The CKY algorithm is essentially a dynamic programming method in the sense that a derivation matching a longer portion of the input string is built by "pasting together" previously computed derivations matching shorter portions. More precisely, for $B \in N$ and $x \in \Sigma^*$ we say that B *matches* x if $B \Rightarrow^* x$. Notice that if B matches x, C matches y, and there is a rule $A \rightarrow BC$, then we can paste these derivations together to find that A matches xy, since $A \Rightarrow BC \Rightarrow^* xC \Rightarrow^* xy$. Similarly, we say that a set of variables Q matches x if each element of Q matches x. For $Q, R \subseteq N$ define

$$Q \otimes R = \{A \mid A \rightarrow BC \text{ is in } P \text{ for some } B \in Q \text{ and } C \in R\}.$$

Notice that if Q matches x and R matches y, then $Q \otimes R$ matches xy.

The CKY algorithm constructs an $(n + 1) \times (n + 1)$ upper triangular matrix $\mathbf{t}$ (indexed 0 through n), called the *recognition matrix*, whose entries are sets of variables; the $t_{i,j}$ entry is to be the set of all variables matching the substring $w_{i,j}$. Thus entries in the jth column match suffixes of the first j symbols of the input. The algorithm is given below in abstract form.

```
1 for each column (* i.e., input symbol *) do
2   begin (* let j be current input position *)
3     match jth input symbol
4     for increasing length suffixes do
5       paste derivations of suffix of first j symbols of input
6   end
7 if entire input matched then accept
8                       else reject;
```

The same algorithm follows in Pascal-like notation.[2]

Algorithm 2.1 (Cocke-Kasami-Younger)

```
1 for j := 1 to n do
2   begin
3     t_{j-1,j} := {A | A → a_j is in P};
4     for i := j-2 downto 0 do
5       t_{i,j} := ∪_{i<k<j} t_{i,k} ⊗ t_{k,j}
6   end
7 if S ∈ t_{0,n} then accept
8           else reject;
```

The essential property of the algorithm is given by the following characterization theorem.

[2] In our notation, all "multiply" operators have higher precedence than set union.

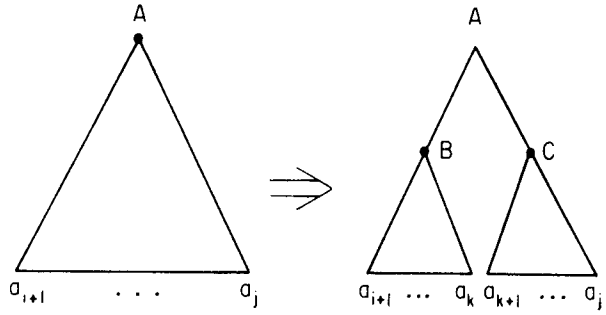

Fig. 1. Proof of Proposition 2.1, "if" direction.

PROPOSITION 2.1. *After executing CKY, for* $0 \leq i < j \leq n$, *we have* $A \in t_{i,j}$ *if and only if* A *matches* $w_{i,j}$.

SKETCH OF PROOF. Detailed proofs are available in the literature [2, 12, 15], but a proof will be sketched, since it parallels the proof which appears in the next section. The proof proceeds by an induction which follows the order in which matrix entries are generated, namely, completing columns from left to right, with each column being completed from bottom to top. Notice that this order completes the row entries to the left of $t_{i,j}$ (i.e., $t_{i,k}$, $k < j$) and in the column below $t_{i,j}$ (i.e., $t_{k,j}$, $i < k$) before $t_{i,j}$ is completed. $t_{i,j}$ is basically the "inner product" of (the nonempty portions of) row i with column j, i.e., a union of $\otimes$-products of corresponding entries of row i and column j. The first part of the proof is to show that each $t_{i,j}$ matches $w_{i,j}$. The argument is straightforward. The second part of the proof is to show that each $t_{i,j}$ contains *all* variables A matching $w_{i,j}$. The main step is for $\lg(w_{i,j}) \geq 2$. Here we use the fact that in the derivation tree for $A \Rightarrow^* w_{i,j}$ some part of $w_{i,j}$ descends from the left child of A and the rest descends from the right. That is, there is some rule $A \rightarrow BC$ and some k, $i < k < j$, such that $B \Rightarrow^* w_{i,k}$ and $C \Rightarrow^* w_{k,j}$. (See Figure 1.) By the induction hypothesis, B must be in $t_{i,k}$ and $C \in t_{k,j}$, so $A \in t_{i,k} \otimes t_{k,j} \subseteq t_{i,j}$. □

The usual version of CKY [2, 20, 25, 39] is different from that presented here in two respects. First, the matrix is organized and indexed differently. Second, the matrix entries are completed along (what in our representation correspond to) diagonals progressively farther from the main diagonal (i.e., $\{t_{i,i+d} \mid 0 \leq i \leq n - d\}$ for $d = 1, 2, \ldots, n$), rather than working up columns progressively farther from the left as we have done. Actually, all that matters is that the row to the left of $t_{i,j}$ and the column below $t_{i,j}$ must be finished before $t_{i,j}$ is computed. We have given the algorithm in this form since it parallels the new algorithm and since it is an on-line recognizer.

The CKY algorithm was the first general context-free recognizer with a subexponential running time, and it is one of the simplest. However, it is not commonly used in practice, for three reasons. First, it is somewhat inconvenient to write and read grammars in Chomsky form. For one thing, breaking up long productions into ones with right sides of length 2 may tend to obscure the structure of the grammar and language. Additionally, in English optional constructions are common; for instance, a noun phrase might be a noun optionally

followed by a modifier. Optional parts are typically incorporated in grammars by a combination of Λ-rules and chain-rules. Eliminating these Λ- and/or chain-rules may require widespread modifications to the grammar. Of course, conversion to Chomsky form may be done mechanically so that the grammar writer is not directly faced with these problems. However, as a second problem, the conversion may square the size of the grammar. The CKY algorithm runs in time proportional to $|G|n^3$. In many applications $|G|$ is much bigger than n, so squaring the size of the grammar can have a drastic effect on performance. For example, in a natural language processing system the grammar might well have several hundred productions, but the input is just one English sentence of 20 or 30 words. The third problem with CKY is that it may spend lots of time making "useless" matches. That is, it finds every variable A matching some substring $w_{i,j}$ without regard to whether or not that match can occur within the context of the rest of the sentence, i.e., whether or not $S \Rightarrow^* w_{0,i}Aw_{j,n}$. Matches which fail to satisfy this criterion may make up the great majority of all matches found. In one experiment reported by Pratt [27], at least 80 percent of the matches were of this type.

The new algorithm has much in common with the Cocke–Kasami–Younger algorithm. In particular, it is also a "dynamic programming" method in which derivations matching longer portions of the input are built up by pasting together previously computed derivations matching shorter portions. The two major differences are that arbitrary grammars are handled (not just Chomsky form ones), and certain "useless" matches are eliminated.[3] We henceforth relax the restriction on grammars to Chomsky form.

In order to handle arbitrary grammars, it will be convenient to speak of matching only part of the right side of a rule to the input, rather than the whole rule. As a notation for dealing with this situation, we introduce the *dotted rule.*

Definition. Let $G = (V, \Sigma, P, S)$ be a context-free grammar and let $\cdot$ be a symbol not in V. If $A \rightarrow \alpha\beta$ is in P, then we say that $A \rightarrow \alpha \cdot \beta$ is a *dotted rule* of G.

The idea is that the dotted rule $A \rightarrow \alpha \cdot \beta$ indicates that α has been matched to the input, but it is not yet known whether β matches. (A similar idea is used to explain LR parsing.) The notion of matching is made precise as follows.

Definition. For $U \in V$, $x \in \Sigma^*$, and $A \rightarrow \alpha \cdot \beta$ a dotted rule, we say that

$$A \rightarrow \alpha \cdot \beta \quad \textit{matches} \quad x \quad \text{if} \quad \alpha \Rightarrow^* x$$

and

$$U \quad \textit{matches} \quad x \quad \text{if} \quad U \Rightarrow^* x.$$

A *set* of variables and/or dotted rules *matches* x if each element of the set matches x.

As with CKY, the new algorithm constructs an $(n + 1) \times (n + 1)$ upper triangular matrix $\mathbf{t} = (t_{i,j})$, $0 \leq i, j \leq n$, whose entries are sets matching substrings of the input, but here we use sets of dotted rules rather than sets of variables.

[3] It should be noted that other methods of solving these problems have been considered, but they seem to have some drawbacks. These are considered in Section 3.

Notice that if $A \to \alpha \cdot B\beta$ matches x and B matches y, then we can "paste" the derivations together to conclude that $A \to \alpha B \cdot \beta$ matches xy, since

$$\alpha B \Rightarrow^* xB \Rightarrow^* xy.$$

Matchings can be combined in this way until the dot has been moved to the right end of the rule, at which point we know that A itself matches the string.

For our algorithm it is more efficient if the result of pasting two derivations together in this way is always a dotted rule which matches a string which is strictly longer than either of the two initial strings. However, in an arbitrary grammar if we have, say, $B \Rightarrow^* \Lambda$, and if $A \to \alpha \cdot B\beta$ matches x, then so does $A \to \alpha B \cdot \beta$. Of course, there could be several consecutive variables which all match Λ. The notion of "pasting together" which we use will automatically move the dot to the right of variables matching Λ, so that we need only worry about combining derivations matching nonnull strings. In addition to saving time, this property also turns out to simplify the algorithm and its proof. The operations we need which are analogous to the $\otimes$ product used in CKY are defined next.

Definition. Let $G = (V, \Sigma, P, S)$ be a context-free grammar. Let Q be a set of dotted rules, and let $R \subseteq V$. Define

$$Q \times R = \{A \to \alpha B\beta \cdot \gamma \mid A \to \alpha \cdot B\beta\gamma \text{ is in } Q, \beta \Rightarrow^* \Lambda, \text{ and } B \in R\},$$

$$Q * R = \{A \to \alpha B\beta \cdot \gamma \mid A \to \alpha \cdot B\beta\gamma \text{ is in } Q, \beta \Rightarrow^* \Lambda, \text{ and } B \Rightarrow^* C \text{ for some } C \in R\}.$$

Notice that there may be several distinct prefixes of $\beta\gamma$ which generate Λ. We include all of them, not just the longest, since any of the variables involved might match the next portion of the input. The $*$ product is the same as the $\times$ product except that it includes effects of chain derivations ($B \Rightarrow^* C$) which would otherwise cause slight complications somewhat analogous to Λ-derivations. The reasons for this definition should become clear later. Notice that $Q \times R \subseteq Q * R$.

Next we extend these products to the case where both arguments are sets of dotted rules.

Definition. Let $G = (V, \Sigma, P, S)$ be a context-free grammar, and let Q, R be sets of dotted rules. Define

$$Q \times R = \{A \to \alpha B\beta \cdot \gamma \mid A \to \alpha \cdot B\beta\gamma \in Q, \beta \Rightarrow^* \Lambda, \text{ and } B \to \eta \cdot \text{ is in } R\},$$

$$Q * R = \{A \to \alpha B\beta \cdot \gamma \mid A \to \alpha \cdot B\beta\gamma \in Q, \beta \Rightarrow^* \Lambda, B \Rightarrow^* C \text{ for some } C \in N, \text{ and } C \to \eta \cdot \text{ is in } R\}.$$

Again note that

$$Q \times R \subseteq Q * R.$$

By observing the fact that a variable matches a string only if the symbols in the right side of one of its productions can be pasted together appropriately, the reader can begin to see the motivation for these definitions.

Example. Consider the grammar

$$S \to ABAC$$
$$A \to \Lambda$$
$$B \to CDC$$
$$C \to \Lambda$$
$$D \to a.$$

Let $Q = \{S \to A \cdot BAC\}$. Then

$$Q \times \{B\} = \{S \to AB \cdot AC, S \to ABA \cdot C, S \to ABAC \cdot\},$$

while

$$Q * \{D\} = \{S \to AB \cdot AC, S \to ABA \cdot C, S \to ABAC \cdot\}.$$

The other major difference between this algorithm and CKY is that we want to eliminate some useless matches. Instead of finding all $A \to \alpha \cdot \beta$ which match some substring $w_{i,j}$ of the input, we find only those dotted rules $A \to \alpha \cdot \beta$ which match $w_{i,j}$ *and* may legally follow the portion of the input to the left of $w_{i,j}$; i.e., $S \Rightarrow^* w_{0,i}A\theta$ for some $\theta \in V^*$. This is a weaker condition than insisting that the dotted rule be consistent with both left and right context ($S \Rightarrow^* w_{0,i}Aw_{j,n}$), but is easier to compute, while being sufficiently restrictive to be of significant practical utility. In fact, in the experiment mentioned earlier [27], a condition of this form eliminated the quoted 80 percent of the matches, while the stronger condition eliminated only a few more. It is not hard to see how this might happen in, say, a grammar for English (which was the basis for Pratt's experiment). For instance, "who" must begin a relative clause in "The boy who . . . ," but it must begin a question in "Who . . . ." Thus, left context alone is sufficient to narrow greatly the range of possibilities in these cases. Such left–right biases are probably very common in most natural and artificial languages.

Definition. For A a variable, $A \to \alpha \cdot \beta$ a dotted rule, and $x \in \Sigma^*$, we say that A *follows* x if $S \Rightarrow^* xA\theta$ for some $\theta \in V^*$, $A \to \alpha \cdot \beta$ *follows* x if A does, and a *set* of variables and/or dotted rules *follows* x if each element of the set follows x.

Given some dotted rules which follow a prefix of the input and match an extension to it, it is easy to find variables which may follow the longer prefix. Suppose $A \to \alpha \cdot B\beta$ follows x and matches y. Then B follows xy, since $S \Rightarrow^* xA\theta \Rightarrow x\alpha B\beta\theta \Rightarrow^* xyB\beta\theta$ for some $\theta \in V^*$. Further, if $B \to C\gamma$, $C \to D\delta$ are rules, then C and D follow xy also.

For convenience we say that in this case B, C, and D follow $A \to \alpha \cdot B\beta$ as well. Generalizing this example, we define a function PREDICT which gives the set of variables or dotted rules following a given set of variables or dotted rules and, in the case of dotted rules, matching Λ.

Definition. Let $G = (V, \Sigma, P, S)$ be a context-free grammar and let $R \subseteq V$. Define

$$\text{PREDICT}(R) = \{C \to \gamma \cdot \xi \mid C \to \gamma\xi \text{ is in } P, \gamma \Rightarrow^* \Lambda, \text{ and } B \Rightarrow^* C\eta \text{ for some } B \in R \text{ and some } \eta \in V^*\}.$$

If R is a set of dotted rules then

$$\text{PREDICT}(R) = \text{PREDICT}(\{B \mid A \to \alpha \cdot B\beta \text{ is in } R\}).$$

It is important to note that PREDICT depends only on the grammar and can be precomputed for each variable or dotted rule.

Example. Using the same grammar as the previous example, we have

$$\text{PREDICT}(\{S\}) = \{S \to \cdot ABAC, S \to A \cdot BAC, A \to \cdot, B \to \cdot CDC, B \to C \cdot DC, C \to \cdot, D \to \cdot a\}.$$

It should be remarked that PREDICT is something of a misnomer. The name has been borrowed from Earley's algorithm, which is very closely connected with this algorithm, as is shown in Section 4. However, it does not really predict rules or variables which will be found. It would be more accurate to say that it is used to restrict attention to those variables which, if matched, would allow extension of partially matched rules found previously Thus, "restrictor," "extendor," or perhaps "wishor" might be better names than "predictor."

Now we can present the main algorithm.

Algorithm 2.2. Let $G = (V, \Sigma, P, S)$ be any context-free grammar. Let $w = a_1 \cdots a_n$, where $n \geq 0$ and $a_k \in \Sigma$ for each k, $1 \leq k \leq n$, be the string to be recognized. Form an $(n + 1) \times (n + 1)$ matrix $\mathbf{t} = (t_{i,j})$ (indexed 0 through n in both dimensions) as follows.

```
1  begin
2    t_{0,0} := PREDICT({S}); (* match, Λ *)
3    for j := 1 to n do
4    begin (* build column j, given columns 0, ..., j − 1 *)
5      t_{j−1,j} := t_{j−1,j−1} * {a_j}; (* paste input symbol to Λ derivations that precede it *)
6      for i := j − 2 downto 0 do
7      begin
8        r := (∪_{i<k<j−1}(t_{i,k} × t_{k,j})) ∪ t_{i,j−1} × (t_{j−1,j} ∪ {a_j}); (* paste non-Λ derivations *)
9        t_{i,j} := r ∪ t_{i,i} * r (* paste matched suffix to Λ derivations that precede it and
                                       extend match to reflect chain rules *)
10     end;
11     t_{j,j} := PREDICT (∪_{0≤i≤j−1} t_{i,j});
12   end;
13   if some S → α· is in t_{0,n}  then accept
14                                 else reject
15 end.
```

The matrix $\mathbf{t}$ constructed by the algorithm is called the *recognition matrix.*

The order of computation of the new algorithm is similar to the CKY method. Columns of the recognition matrix are completed from left to right (the "**for** j . . . " loop starting on line 3). With one exception the elements of each column are completed in order from bottom to top (the "**for** i . . . " loop starting on line 6). The exception is that the bottommost element, i.e., the one on the main diagonal, is completed last (line 11). As in CKY, the element immediately above the main diagonal is treated as a special case (line 5). All other off-diagonal elements $t_{i,j}$ are formed as follows. First, form the "inner product" of (the nonempty portions of) row i with column j, i.e., a union of $\times$-products of elements from row i with (correspondingly positioned) elements from column j (line 8). When computing this product, treat $t_{j-1,j}$ as if it contained the jth input symbol a_j (second term of line 8). The second step is to augment the "inner product" by taking a $*$-product

with the diagonal element on the ith row (line 9). This single step in the algorithm incorporates all the derivation steps using chain rules to match a suffix of the input read so far. The result is $t_{i,j}$.

To establish the correctness of the algorithm, one proves the following characterization.

THEOREM 2.1. *After executing Algorithm* 2.2, *a dotted rule* $A \to \alpha \cdot \beta$ *will be in* $t_{i,j}$ *if and only if it follows* w_i *and matches* $w_{i,j}$; *i.e.*,

$$S \Rightarrow^* w_i A\theta \quad \text{for some} \quad \theta \in V^* \quad \text{and} \quad \alpha \Rightarrow^* w_{i,j}.$$

We will not give the full formal proof here. A complete proof may be found in [31]. Also, see [15] for a proof of an algorithm which is quite close to the present one. Although we shall not do the detailed proof, there is much to be learned by looking at the structure of the proof, because many of the lemmas explain the insights which can be capitalized upon in implementation and optimization. Therefore the statements of the lemmas will be presented without proof.

One key property of the $\times$- and $*$-products is that like $\otimes$ for CKY, they "paste together" derivations corresponding to all of the elements of the two sets. The following lemma captures this fact.

LEMMA 2.1. *If* Q *matches* x *and* R *matches* y, *then* $Q \times R$ *and* $Q * R$ *match* xy.

Further the products preserve the following relationship:

LEMMA 2.2. *If* Q *follows* x, *so do* $Q \times R$ *and* $Q * R$.

Finally, the PREDICT function "extends" the following relationship:

LEMMA 2.3. *If* Q *follows* x *and matches* y, *then* $PREDICT(Q)$ *follows* xy *and matches* Λ.

With these lemmas it is easy to prove the "only if" half of the characterization theorem:

LEMMA 2.4. *Each set* $t_{i,j}$ *constructed by Algorithm* 2.2 *follows* w_i *and matches* $w_{i,j}$.

PROOF. By an induction which follows the order of computation of the algorithm. □

LEMMA 2.5. *The algorithm places every dotted rule which follows* w_i *and matches* $w_{i,j}$ *into* $t_{i,j}$.

PROOF. Again, by an induction following the order of computation. (Cf. [15, 31].) □

The key idea, as in the proof for the CKY algorithm, is that a dotted rule matching a long portion of the input implies the existence of two dotted rules matching shorter portions, which (inductively) have been previously placed in the matrix. The argument is more complicated technically than for CKY due to the possibility of Λ- and chain-derivations.

Lemmas 2.4 and 2.5 together constitute a proof of Theorem 2.1. The correctness of Algorithm 2.2 is a simple corollary:

COROLLARY 2.1. *Algorithm* 2.2 *is correct, i.e., it accepts* w *if and only if* $w \in L(G)$.

PROOF. The algorithm accepts if and only if there is some dotted rule $S \rightarrow \alpha \cdot$ in $t_{0,n}$. By Theorem 2.1 this happens if and only if $S \Rightarrow^* w_{0,n} = w$. □

We close this section with two variants of the algorithm which will be useful in the next section. The first is derived by changing the order of computation slightly: we interchange the order of the two innermost loops (i on lines 6–10 and k on line 8). Instead of getting all of the contributions to an element $t_{i,j}$ of the jth column by taking products along the ith row and down the jth column (before considering $i - 1$), we get all of the contributions to the jth column from the kth column (before considering $k - 1$). Instead of the single temporary variable r in Algorithm 2.2, we need separate temporaries to hold the partial results for each row of the jth column. It is convenient to use the jth column of the matrix for this purpose; we hope that the reader will not be confused by the use of the variable $t_{i,j}$ to represent partial results at one point and final results at a later point. As we will see, in the algorithm below $t_{j,j}$ is completed at line 12, $t_{j-1,j}$ at line 5, and $t_{k,j}$ for $k < j - 1$ at line 9.

Algorithm 2.3 (notation as in Algorithm 2.2)

```
1  begin
2    t_{0,0} := PREDICT({S});
3    for j := 1 to n do
4    begin
5      t_{j-1,j} := t_{j-1,j-1}*{a_j};
6      for 0 ≤ i ≤ j − 2 do t_{i,j} := t_{i,j-1} × (t_{j-1,j} ∪ {a_j});
7      for k := j − 2 downto 0 do
8      begin
9        t_{k,j} := t_{k,j} ∪ t_{k,k}*t_{k,j};
10       for 0 ≤ i ≤ k − 1 do t_{i,j} := t_{i,j} ∪ t_{i,k} × t_{k,j}
11     end;
12     t_{j,j} := PREDICT(∪_{0≤i≤j-1} t_{i,j})
13   end;
14   if some S → α· is in t_{0,n}  then accept
15                                 else reject
16 end.
```

The use of the notation "**for** $a \leq i \leq b$" on lines 6, 10, and 12 rather than the usual "**for** $i := a$ **to** b" or "**for** $i := b$ **downto** a" denotes the fact that the order of execution is not relevant. In fact, the loop body could be executed in parallel for all values of i without invalidating the algorithm.

We will not give a formal argument that this algorithm is equivalent to Algorithm 2.2, but it is not difficult to see. The main idea is to show that after executing the body of the loop on lines 8–11 for some k, $t_{i,j}$ are complete for $i \geq k$, and for $i < k$, $t_{i,j}$ contains all the contributions resulting from columns to the

right of $k-1$, namely,

$$\bigcup_{k \le l < j-1} t_{i,l} \times t_{l,j} \cup t_{i,j-1} \times (t_{j-1,j} \cup \{a_j\}).$$

The result then follows by induction.

The second variant involves a slightly more substantive change to Algorithm 2.3. Note that the algorithm completes $t_{k,j}$ by doing a $*$-product with $t_{k,k}$ (line 9), thereby incorporating chain rules into the match, and then takes $\times$-products of the rest of column k with $t_{k,j}$ (line 10). It turns out to be equivalent to take $*$-products of all of column k with the contents of $t_{k,j}$ *before* it is completed at line 9; i.e., "$\times$" can be replaced by "$*$" on line 6 and lines 9 and 10 can be replaced by

$$\begin{array}{ll} 9' & r := t_{k,j}; \\ 10' & \textbf{for } 0 \le i \le k \textbf{ do } t_{i,j} := t_{i,j} \cup t_{i,k} * r \end{array} \qquad (2.1)$$

to yield a new algorithm which will be further developed into Algorithm 2.4.

Letting r be as above, we see that Algorithm 2.3 computes "$t_{i,j} \cup t_{i,k} \times (r \cup t_{k,k} * r)$." We will show that

$$t_{i,k} \times (r \cup t_{k,k} * r) = t_{i,k} * r.$$

In view of Lemmas 2.1 and 2.2 it is easy to see that Algorithm 2.4 with (2.1) computes sets $t_{i,j}$ which follow w_i and match $w_{i,j}$. Thus the correctness of Algorithm 2.4 with (2.1) will follow if we show that the new method omits none of the items found by the old, i.e., that

$$t_{i,k} \times (r \cup t_{k,k} * r) \subseteq t_{i,k} * r.$$

Now $t_{i,k} \times r \subseteq t_{i,k} * r$, so we just need to show that

$$t_{i,k} \times (t_{k,k} * r) \subseteq t_{i,k} * r.$$

In fact, this is true of any sets provided only that $t_{k,k}$ matches Λ, since all of the items in $t_{k,k} * r$ then are part of some chain derivation, which can be found directly by computing $t_{i,j} * r$. If $A \to \alpha \cdot \beta$ is in the expression on the left, there must be α_1, α_2, γ_1, γ_2, δ, B, C, and D such that

$$\begin{array}{lll} \alpha = \alpha_1 B \alpha_2, & & \\ A \to \alpha_1 \cdot B \alpha_2 \beta & \text{is in} & t_{i,k}, \\ \alpha_2 \Rightarrow^* \Lambda, & & \\ B \to \gamma_1 \cdot C \gamma_2 & \text{is in} & t_{k,k}, \\ \gamma_1 \gamma_2 \Rightarrow^* \Lambda, & & \\ C \Rightarrow^* D, & & \\ D \to \delta \cdot & \text{is in} & r \end{array}$$

(giving $B \to \gamma_1 C \gamma_2 \cdot$ in $t_{k,k} * r$ and hence $A \to \alpha_1 B \alpha_2 \cdot \beta$ in $t_{i,k} \times (t_{k,k} * r)$). But then $B \Rightarrow^* D$, so $A \to \alpha_1 B \alpha_2 \cdot \beta$ is in $t_{i,k} * r$, which is what we needed to show.

It will be convenient to use the following new notation. Let the vector $\bar{t}_k$

represent the kth column of the matrix; $\vec{t}_k * r$ and $\vec{t} \cup \vec{t}'$ are the obvious componentwise operations. Thus, for example, line 10′ above becomes

$$\vec{t}_j := \vec{t}_j \cup \vec{t}_k * r.$$

Using this notation the algorithm is the following.

Algorithm 2.4 (notation as in Algorithm 2.2)

```
1  begin
2    t_{0,0} := PREDICT({S}); (* match Λ *)
3    for j := 1 to n do
4    begin
5      t_j := t_{j-1} * {a_j}; (* paste input symbol to derivations that precede it *)
6        for k := j - 2 downto 0 do
7          t_j := t_j ∪ t_k * t_{k,j}; (* paste matched suffix to derivations that precede it, extend-
                                          ing match to reflect chain rules *)
8      t_{j,j} := PREDICT(∪_{0≤i≤j-1} t_{i,j}) (* match Λ suffixes *)
9    end;
10   if some S → α · is in t_{0,n}  then accept
11                                  else reject
12 end.
```

In this algorithm line 5 does the same computation as lines 5–6 of Algorithm 2.3 for the same reasons that line 7 can replace lines 9 and 10 of the previous algorithm. The proof is identical, except that C may be in Σ and "$D = a_j$" replaces "$D \rightarrow \delta \cdot$ is in r."

In line 7 notice that $t_{k,j}$ is an element of $\vec{t}_j$, so it may seem that to implement this statement correctly, $t_k * t_{k,j}$ must be completely computed before $\vec{t}_j$ is modified, or $t_{k,j}$ must be saved in a temporary (such as "r" in lines 9′ and 10′) before $\vec{t}_j$ is modified. However, all the elements added to $t_{k,j}$ by this step must match $w_{k,j}$, so by Lemmas 2.1 and 2.2 no unwanted items would be added to $\vec{t}_j$ even if some or all of the additions to $t_{k,j}$ were made before or during the execution of line 7. Of course, the computation might be slowed by the unnecessary consideration of these new elements while computing $t_k * t_{k,j}$.

As an example consider the grammar

$$S \rightarrow AS \mid b$$

$$A \rightarrow aA \mid bA \mid \Lambda.$$

Figure 2 shows the state of the recognition matrix for this grammar and input $w = aab$ both before and after the second-to-last row of the last column has been completed (i.e., before and after executing "$\vec{t}_3 := \vec{t}_3 \cup \vec{t}_1 * t_{1,3}$" on line 7 of Algorithm 2.4.

In the next section we look at implementation of the algorithm in detail. Choice of data structure for **t**, computation of ×, *, and PREDICT, preprocessing the grammar, and various optimizations are considered.

3. IMPLEMENTATION CONSIDERATIONS

The algorithms given in the previous section were presented in terms of fairly high-level set-theoretic operations. A program written in a very high-level lan-

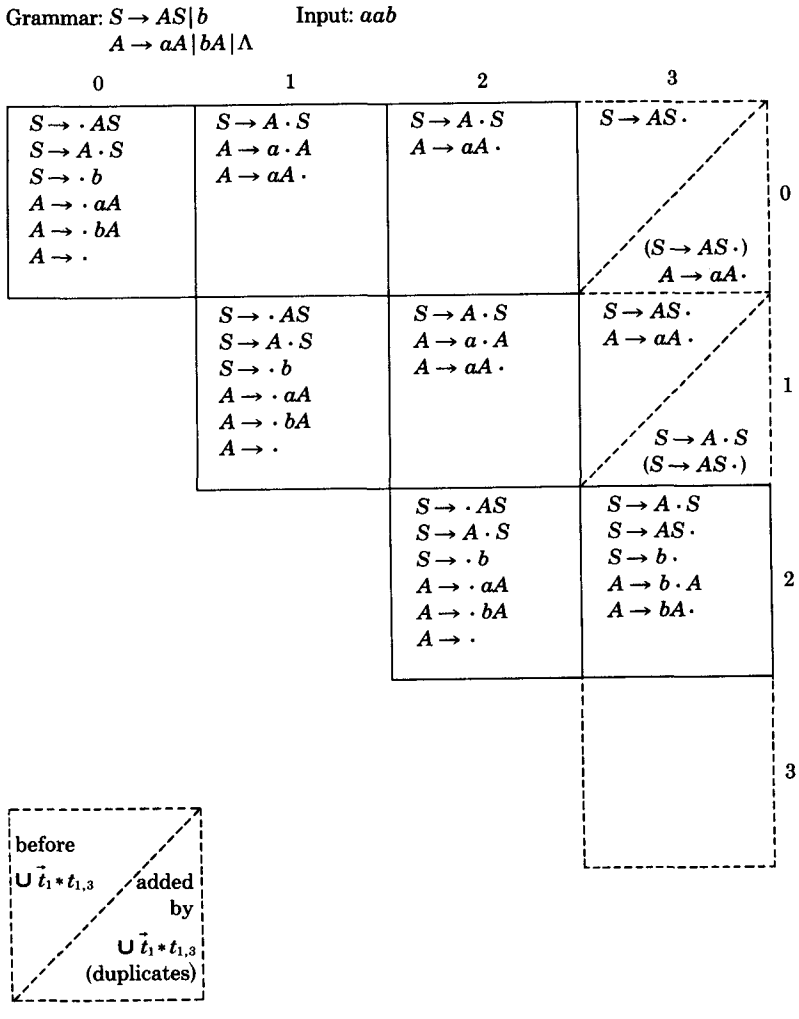

Fig. 2. Example of recognition matrix.

guage which directly implemented the algorithm would be quite inefficient. In many situations, significantly more efficient implementations are possible. In this section we consider such implementations and the circumstances in which they are advantageous.[4] Unfortunately, a "best" implementation is not given, since the suitability and efficiency of the various methods depends on the application, the grammar, and the input.

Some of the characteristics of "typical" applications need to be discussed first. As mentioned earlier, it is not unusual for the grammar to be much larger than the input. In addition, the recognition matrix tends to be sparse, i.e., have many

[4] Researchers studying "automatic" implementation of data structures should regard this analysis as a test for their methods.

empty positions and only a few dotted rules in each of the nonempty positions. In spite of this sparseness, it still can be quite large, even when compressed. Thus in many applications space may be at least as valuable as time.

For example, a grammar for English was supplied by Vaughan Pratt [29]. He describes the grammar as an experiment in seeing how simple the grammar could be made at the expense of complicating other portions of his system. The grammar has about 90 nonterminals, 160 productions, and about 450 dotted rules. By design, there are no Λ-rules, nor rules with right sides longer than two symbols, but there are chain rules. For a typical input, about 10 to 20 percent of the matrix entries were empty. Except for the entries on the diagonal, the average number of dotted rules per entry was about 20, and the maximum was about 40. These numbers were insensitive to the length of the input, so the space required to store the matrix grew as n^2. In the nonempty cells, one-third to one-half of the entries were of the form $A \rightarrow \alpha \cdot$; typically five or more of these dotted rules had distinct left sides. The elements on the diagonal were very different. Nearly all of them had 150 or more entries; thus they contained almost every rule in the grammar!

The implementations described below should perform well in most applications, particularly those having the characteristics discussed above. For instance, the space and time requirements are determined by the number of dotted rules generated, rather than being fixed by n, the length of the input. The time complexity is at worst $\mathcal{O}(n^3)$,[5] and will be much better in some cases, such as unambiguous grammars. Further, the time requirements are at worst linear in the size of the grammar, so the method will be reasonably efficient in the common case where the grammar is large but the inputs are short.

Although the implementations described should perform well in most practical applications, the ingenious constructor of counterexamples should have little difficulty constructing grammars and/or inputs which thwart all of the optimizations embodied in the various implementations. We give one such example here; some others will appear later in this section. These examples should serve to indicate how bad a "worst case" could be. Consider for any $k \geq 1$ the grammar $G_k = (V_k, \Sigma_k, P_k, S_0)$, where

x^k denotes k repetitions of x,
$V_k = \{S_0, S_1, \cdots, S_{k-1}, a\}$,
$\Sigma_k = \{a\}$,

and P_k has the rules

$$
\begin{array}{lll}
(1) & S_0 & \rightarrow \Lambda \\
(2) & S_0 & \rightarrow a \\
(3) & S_0 & \rightarrow S_1^k \\
(4) & S_1 & \rightarrow S_2^k \\
\vdots & & \vdots \\
(k+1) & S_{k-1} & \rightarrow S_0^k.
\end{array}
$$

[5] $f(n) = \mathcal{O}(g(n))$ if and only if there are positive constants c and n_0 such that for all $n > n_0$, $|f(n)| \leq cg(n)$.

It is easy to see that for all $i, j \geq 0$ and all $0 \leq p \leq k-1$, (a) $S_0 \Rightarrow^*_{G_k} a^i S_p$ and (b) $S_p \Rightarrow^*_{G_k} a^j$. Consider the ith row of the matrix on input a^n. From fact (a) we see that every dotted rule follows a^i, and so every dotted rule is allowable in row i. Note that every dotted rule except $S_0 \rightarrow a \cdot$ matches Λ, so $t_{i,i}$ will contain all but one of the dotted rules in the grammar! Now using fact (b) we see that for all $j \geq 1$ and all $0 \leq p \leq k-1$, $1 \leq q \leq k$, $S_{(p-1) \bmod k} \rightarrow S_p^q \cdot S_p^{k-q}$ matches a^j. Thus all k^2 of those dotted rules will be in $t_{i,i+j}$ for all $j \geq 1$. ($S_0 \rightarrow a \cdot$ will also be in $t_{i,i+1}$.) Thus we see that in contrast to the "typical" example described previously, we can construct examples where not only are there no empty positions in the recognition matrix, but each entry is very "full," containing at least k^2 of the $k^2 + k + 3$ dotted rules of the grammar! We can conclude that any optimizations we might consider which depend on a "sparse" recognition matrix will be of little or no use in the worst case. Nevertheless, such optimizations are extremely valuable in most practical applications.

In the remainder of this section we discuss the $\times$, $*$, and PREDICT operations, choice of data structure for representing the recognition matrix, extracting parses, and some extensions to the algorithm which may save space and/or time. First we give a brief description of the representation of the grammar which will be assumed throughout.

The representation of the grammar is important. The representation described below seems natural, convenient, and efficient. We do not claim that it is optimal, however. We assume that each possible dotted rule is assigned a unique number. These numbers are then used to index one or more tables which tell (i) what nonterminal symbol is on the left side of the dotted rule, (ii) whether the dot is at the right end of the rule, (iii) if not, which symbol occurs to the right of the dot, and (iv) the number of the dotted rule formed by moving the dot one symbol further to the right (if possible). In addition, we have a table which indicates for each symbol in the vocabulary whether it is a terminal or nonterminal symbol, and which gives for each nonterminal A a list of the (numbers of) dotted rules of the form $A \rightarrow \cdot \alpha$. Figure 3 illustrates this representation for the simple grammar used at the end of the previous section (which we will use in examples throughout this section). Note that in practice the entries corresponding to columns (i) and (iii) of the upper table in Figure 3 would use the symbol numbers given in the lower table; the symbols themselves are used there only for clarity. Similarly, the information in the last columns of both tables is redundant and need not be stored in practice. Also the second and third columns of the upper table and the columns of the lower table could be combined if an appropriate coding convention were used.

No other data are needed for our purposes, although we will discuss below the utility of tabulating certain other information which can be derived from the grammar (e.g., which variables generate Λ).

The definition of the $\times$-product given in the previous section was designed to allow a concise statement of the algorithm.[6] For computational purposes it is better to break down the definition somewhat. First, notice that when forming the product $Q \times R$, much of R is extraneous. All we need is a summary of

[6] Recall that $Q \times R = \{A \rightarrow \alpha B \beta \cdot \gamma \mid A \rightarrow \alpha \cdot B \beta \gamma \text{ is in } Q, \beta \Rightarrow^* \Lambda, \text{ and } B \in R\}$.

Grammar: $S \rightarrow AS \mid b$
$A \rightarrow aA \mid bA \mid \Lambda$

Dotted rule number	(i) Left side	(ii) Dot at right end?	(iii) Symbol to right of dot	(iv) Next dotted rule number	Dotted rule (need not be stored)
1	S		A	2	$S \rightarrow \cdot AS$
2	S		S	3	$S \rightarrow A \cdot S$
3	S	Y	—	—	$S \rightarrow AS \cdot$
4	S		b	5	$S \rightarrow \cdot b$
5	S	Y	—	—	$S \rightarrow b \cdot$
6	A		a	7	$A \rightarrow \cdot aA$
7	A		A	8	$A \rightarrow a \cdot A$
8	A	Y	—	—	$A \rightarrow aA \cdot$
9	A		b	10	$A \rightarrow \cdot bA$
10	A		A	11	$A \rightarrow b \cdot A$
11	A	Y	—	—	$A \rightarrow bA \cdot$
12	A	Y	—	—	$A \rightarrow \cdot$

Symbol number	Is it a terminal?	Dotted rule numbers ("$A \rightarrow \cdot a$")	Symbol ("A") (need not be stored)
1		1, 4	S
2		6, 9, 12	A
3	Y	—	a
4	Y	—	b

Fig. 3. Grammar representation.

information about some of the left-hand sides of dotted rules in R; i.e., define (for R a set of dotted rules and/or elements of V)

$$\text{FINAL}(R) = \{U \in V \mid U \text{ is in } R \text{ or some } U \rightarrow \alpha \cdot \text{ is in } R\}.$$

For example,

$$\text{FINAL}(\{A \rightarrow aA \cdot, A \rightarrow bA \cdot, A \rightarrow \cdot, S \rightarrow A \cdot S, a\}) = \{A, a\}.$$

It is easy to see that

$$Q \times R = Q \times \text{FINAL}(R).$$

Note that FINAL(R) is easily computed from R, especially if R has a list representation. Even more important, the product may be computed elementwise; i.e.,

$$Q \times \text{FINAL}(R) = \bigcup_{\substack{q \in Q \\ r \in \text{FINAL}(R)}} \{q\} \times \{r\}. \tag{3.1}$$

Finally, notice that such a product of singletons $\{A \rightarrow \alpha \cdot U\beta\} \times \{U'\}$ is empty if and only if $U \neq U'$. If $U = U'$, the product is easy to compute. Form a sequence

of (one or more) dotted versions of the rule $A \to \alpha U \beta$ by first moving the dot to the right of U, then to the right of the first, second, ... symbols of β, ending with the dot to the left of the leftmost symbol of β which *cannot* generate Λ. The latter can be easily checked if we precompute and store a (bit) table indicating whether or not $U \Rightarrow^* \Lambda$ for each U in V. Note that $U \Rightarrow^* \Lambda$ does not depend on the input, so it can be precomputed. Well-known simple algorithms for doing this computation in time proportional to the size of the grammar may be found in the literature (e.g., [15]).

One optimization is useful. In the recognition algorithms the product set will be built incrementally as we compute various of the singleton products in (3.1). Suppose we attempt to add some item $A \to \alpha \cdot \beta$ to the partially completed product set and find that it was previously entered. Then we do not need to see whether the first, second, ... symbols of β generate Λ, etc., since that will have been tested and appropriate entries made when $A \to \alpha \cdot \beta$ was first added to the set. In our algorithms all uses of the products will be in statements of the form $T := T \cup Q \times R$,[7] so this optimization may be even more effective if the product routine can "see" T also. This suggests the following sort of algorithm.

```
1  procedure ×PROD(T, Q, R)
2  (* compute T := T ∪ Q × R; *)
3    procedure ADD(T, A → α · β)
4    (* compute T := T ∪ {A → αβ′ · β″ | β′β″ = β and β′ ⇒* Λ}; *)
5    begin
6      if A → α · β is in T then return;
7      T := T ∪ {A → α · β};
8      if |β| ≥ 1 then
9        begin
10       (* suppose β is Uδ for some U ∈ V, δ ∈ V* *)
11         if U ⇒* Λ then ADD(T, A → αU · δ)
12       end
13   end
14 begin
15   R′ := FINAL(R);
16   for each A → α · Uβ in Q do
17     if U ∈ R′ then
18       ADD(T, A → αU · β)
19 end
```

The loops on lines 16 and 17 could be reversed, i.e., "**for** each $U \in R'$ **do if** $A \to \alpha \cdot U\beta$ in Q ...." Which form is better depends on the representation of the recognition matrix, an issue we discuss later.

Notice that the test for duplicates on line 6 eliminates the need for such a test in the union on line 7, so it has not cost us anything. As the following example illustrates, the benefit is a savings which may be as large as a factor of $|G|$ (the grammar size).

For any integer $g > 0$, consider the grammar

$$S \to A^g$$

$$A \to \Lambda$$

[7] Recall that following our convention about precedence, $T \cup Q \times R = T \cup (Q \times R)$.

and the sets

$$T = \emptyset,$$
$$Q = \{S \rightarrow A^i \cdot A^{g-i} \mid 0 \leq i \leq g\},$$
$$R = \{A\}.$$

Then while computing "×PROD(T, Q, R)," the "ADD" procedure will be called by "×PROD" g times, namely, once for each rule $S \rightarrow A^i \cdot A^{g-i}$, where $1 \leq i \leq g$. Without the test for duplicates, each call would result in $g - i$ additional calls, one for each $S \rightarrow A^{i+k} \cdot A^{g-i-k}$, where $1 \leq k \leq g - i$. This is a total of about $g^2/2$ calls on "ADD." However, with the test for duplicates only $2g - 1$ calls on "ADD" will be made. That is, if the first call from ×PROD is for $S \rightarrow A \cdot A^{g-1}$, it will recursively generate $g - 1$ other calls for the other dotted rules; all of the $g - 1$ subsequent calls from ×PROD will immediately find duplicates and return without generating subcalls. In practical applications it is probable that long strings of variables generating Λ do not occur often, but some savings can be expected.

There are several options available for implementing the $*$ operation. The simplest is the following. For any $R \subseteq V$, define CHNTO(R) = $\{X \in V \mid X \Rightarrow^* Y$ for some $Y \in R\}$. The procedure is based on the trivial identity $Q * R = Q \times$ CHNTO(FINAL(R)).

```
procedure STARPROD1(T, Q, R)
(* compute T := T ∪ Q * R *)
begin
  R' := FINAL(R);
  R'' := CHNTO(R');
  ×PROD(T, Q, R'')
end
```

R' is easily computed, especially if R is stored compactly, for example, as a linked list. The only question is how to compute R'' quickly. Well-known algorithms for computing the relation $X \Rightarrow^* Y$ can be found in any standard text on language theory (e.g., [15]). Using this relation, it is easy to precompute and store the value of "CHNTO" for all grammar symbols. Then R'' may be computed as

$$R'' = \bigcup_{Y \in R'} \text{CHNTO}(\{Y\}). \tag{3.2}$$

Unless the CHNTO($\{Y\}$) are extremely small sets, bit-vector representations for them and for R'' are attractive. Typical values for $|V|$ are around 100, so each vector would take only a few words on most computers, and (3.2) could be computed using about $|R'| \cdot (|V|/w)$ word-parallel "OR" instructions, where w is the length of a word. This is actually an $\mathcal{O}(|G|^2)$ operation in the worst case, but the constant is so small that it is quite fast in practice. An $\mathcal{O}(|G|)$ method is also possible, but G must be large before the method is faster. See [31] for this method.

There is another reasonable implementation of the $*$-operation which also illuminates some interesting properties of the recognition algorithm. One of the advantages of this second method is that the $X \Rightarrow^* Y$ relation does not need to be precomputed or stored. The method depends on the fact that $*$ is only needed

for statements of the form

$$R := R \cup Q * R, \tag{3.3}$$

where Q has the property that

$$\text{PREDICT}(Q) \subseteq Q. \tag{3.4}$$

Note that in the recognition algorithms (2.2–2.4) all the sets $t_{k,k}$ have this property. In this situation (3.3) can be computed by iterating a ×-product; for example,

```
repeat
  OLDR := R;
  ×PROD(R, Q, R)
until (R = OLDR)
```
(3.5)

Basically, (3.5) correctly implements (3.3) since Q contains all of the rules which are used in any chain derivations we need, so iterating the ×-product will eventually follow the chain back to its source. For example, if $Q = \{A \rightarrow \cdot B, B \rightarrow \cdot C, C \rightarrow \cdot D, D \rightarrow \cdot a\}$ and $R = \{D \rightarrow a \cdot\}$, then the first iteration of the loop above would add $C \rightarrow D \cdot$ to R, the second would add $B \rightarrow C \cdot$, the third would add $A \rightarrow B \cdot$, and the fourth and last would add nothing.

The procedure (3.5) can be refined considerably—in each iteration we really need to consider only those items added to R by the previous iteration. The following algorithm reflects this observation. Here it is convenient to assume a particular representation for R, namely, a linked list. Using this representation, it is easy to process each element of R exactly once by processing the list in order and making additions at the far end.

```
1 procedure STARPROD2(Q, R)
  (* compute R := R ∪ Q * R when
     (i) PREDICT(Q) ⊆ Q, and
     (ii) R is stored as a linked list *)
2 begin
3   for each r in R, in order (i.e., eventually including
      those added in the following step) do
4     append (Q × {r}) − R to R
5 end
```

In practice, a straightforward implementation of the above procedure may be quite adequate, since Q and R are often rather small sets. However, in the worst case the performance can be $\mathcal{O}(|G|^2)$, primarily owing to the searches of Q and R implicit in line 4. The following four techniques will reduce the time to $\mathcal{O}(|G|)$; any or all of them may be useful in practice. First, the same technique for eliminating duplicates used in the "ADD" routine of "×PROD" is applicable. Second, whenever $Q \times \{U \rightarrow \alpha \cdot\}$ is computed, note that U was used, so that one does not compute a ×-product with some other rule $U \rightarrow \beta \cdot$. Third, store R so that one can quickly test whether or not a particular item is in R. This will speed the elimination of duplicates on line 4. For example, a bit vector with one bit per dotted rule could be used. Using this representation, note that it is not necessary to have R as a linked list; all we need is a list of all $U \in \text{FINAL}(R)$ for which $Q \times \{U\}$ has not yet been computed. Fourth and last, store Q so that for any U, all

rules of the form $A \rightarrow \alpha \cdot U\beta$ may be quickly found. For example, keep all such items on one linked list whose start is found in an array indexed by U.

The fourth point above is connected to some issues involved with the choice of representation for **t**, which might as well be discussed here. In the column-oriented version of the recognizer (Algorithm 2.3), each element of a column, say k, is multiplied by $t_{k,j}$. Above we suggest that each set $t_{k,k}$ should be partitioned so that all rules with a dot in front of a U may be found easily. Suppose that each set in column k, not just $t_{k,k}$, is partitioned in this way. Consider the computation performed by "STARPROD2($t_{k,k}$, $t_{k,j}$)." When it computes each $t_{k,k} \times \{r\}$ on line 4, suppose we also compute "**for** $0 \leq i \leq k-1$ **do** $t_{i,j} := t_{i,j} \cup t_{i,k} \times \{r\}$;". This is easily computed, given the suggested partitioning of the $t_{i,k}$. We assert that if Algorithm 2.3 computes $*$-products in this way, then it will be a correct recognizer, even if the other $\times$-products computed in Algorithm 2.3 (i.e., lines 6 and 10) are removed, since "r" in STARPROD2 will eventually run through all elements of the set $t_{k,j}$ which would have been used in the $\times$-product on line 10 of Algorithm 2.3. Thus each $t_{i,k} \times t_{k,j}$ will eventually be computed. (We remark in passing that this computation is not the same as Algorithm 2.4, wherein we compute $*$-products of $t_{i,k}$ with a *partially* completed version of $t_{k,j}$ (i.e., before "$t_{k,j} := t_{k,j} \cup t_{k,k} * t_{k,j}$").)

PREDICT has several similarities to $\times$ and $*$. Given Q, let

$$Q' = \{B \in N \mid A \rightarrow \alpha \cdot B\beta \text{ is in } Q\}, \tag{3.6}$$

$$Q'' = \{C \in N \mid B \Rightarrow^* C\eta \text{ for some } B \in Q' \text{ and some } \eta\}. \tag{3.7}$$

Then

$$\text{PREDICT}(Q) = \text{PREDICT}(Q') \tag{3.8}$$

$$= \{C \rightarrow \gamma \cdot \delta \mid C \in Q'' \text{ and } \gamma \Rightarrow^* \Lambda\}. \tag{3.9}$$

It is easy to compute Q', given Q. For our purposes we are only interested in $Q = \bigcup_{i<j} t_{i,j}$ for some j. We do not actually need to form Q. We can construct Q' directly by scanning the jth column of **t**. The data structure chosen for **t** may make this computation even easier. For example, if all the items in column j with the dot to the left of the same symbol were on one list (a possibility mentioned in the previous section), Q' is just the set of variables whose lists are not empty. Computing Q'' is very similar to computing the R'' of the previous section. For B in N define

$$\text{LEFT}(B) = \{C \mid B \Rightarrow^* C\eta \text{ for some } \eta \in V^*\}.$$

Then

$$Q'' = \bigcup_{B \in Q'} \text{LEFT}(B).$$

As with R'' there are two approaches. The first is to precompute LEFT, store it as an array of bit vectors, and form Q'' by ORing some together. The second is to precompute DIRECTLEFT(B) = $\{C \mid B \rightarrow \gamma C\delta$ is in P, where $\gamma \Rightarrow^* \Lambda\}$, store it as a directed graph, and use a "traverse-and-mark" method. (Note that LEFT is the transitive closure of DIRECTLEFT, and also that a transitive reduction of

DIRECTLEFT would work, too.) The first method should be quite fast in practice, but is $\mathcal{O}(|G|^2)$ in the worst case; the second is $\mathcal{O}(|G|)$, but will be superior only for large grammars.

Given Q'' it is easy to compute PREDICT by using (3.9) and the same technique of moving the dot to the right of variables generating Λ as was used in $\times$ and $*$.

One might be able to save space by storing Q' or Q'' in place of $t_{j,j}$, without incurring too great a time penalty. Since products with $t_{j,j}$ are given special handling anyway, this change will not upset the organization of the program too much. It is argued in [27] that the number of alternative rules for each variable causes PREDICT to generate a large number of items, most of which are found to be dead ends as soon as another one or two input symbols are processed. Thus, storing Q' or Q'' instead of $t_{j,j}$ can save lots of space and may save time which would otherwise be spent searching sets containing many useless items.

As with $*$, it is also possible to compute PREDICT by an iterative process which does not require precomputing LEFT. Note that if $A \to BCD$ is a rule, then $B \in \text{LEFT}(A)$, and if $B \Rightarrow^* \Lambda$, then $C \in \text{LEFT}(A)$, etc. Thus PREDICT can be formed according to the following rule: Whenever $A \to \alpha \cdot B\beta$ is added, also add (recursively) all B rules $B \to \cdot\, \gamma$ unless it was done previously, and if $B \Rightarrow^* \Lambda$, also add (recursively) the rule $A \to \alpha B \cdot \beta$. In effect, this algorithm is the same as the $\mathcal{O}(|G|)$ graph traversal method described above; adding B rules when $A \to \alpha \cdot B\beta$ is added (if they have not already been added) corresponds to following the edge of DIRECTLEFT from A to B and marking B (if it has not already been marked).

All of these methods can be made to work incrementally, i.e., as column j is being built rather than after it is finished. Whenever some $A \to \alpha \cdot B\beta$ is added anywhere in column j, we add PREDICT($\{B\}$) *to* $t_{j,j}$. As usual, to do this efficiently we need to check for duplicates at each step to avoid recomputing a lot of items. This checking may make the incremental approach slightly slower than doing PREDICT on the whole column at once. However, it does have one feature which may sometimes be useful. When doing PREDICT by the iterative method outlined above, the relation $B \Rightarrow^* \Lambda$ does not have to be precomputed, since it will be computed as needed. That is, when adding $A \to \alpha \cdot B\beta$, we compute PREDICT($\{B\}$), so we can tell if $B \Rightarrow^* \Lambda$ by checking whether PREDICT added some $B \to \gamma \cdot$ to $t_{j,j}$. (This is essentially the same as Earley's method [8], which is discussed in Section 4.)

We consider four possible representations for the recognition matrix **t**. All four should give efficient algorithms using a minimum of space. All four are intended for use with one of the column-oriented versions of the algorithm (Algorithm 2.3 or 2.4). These versions have a slight advantage since they do not need to access the matrix by both row and column, as in Algorithm 2.1.

We illustrate the four data structures using the simple grammar introduced at the end of Section 2 and the input $w = aab$ of length $n = 3$. Recall that Figure 2 shows that state of the recognition matrix for this grammar and input both before and after the next-to-the-last row of the last column have been completed (i.e., before and after executing "$\vec{t}_3 := \vec{t}_3 \cup \vec{t}_1 * t_{1,3}$" on line 7 of Algorithm 2.4). We

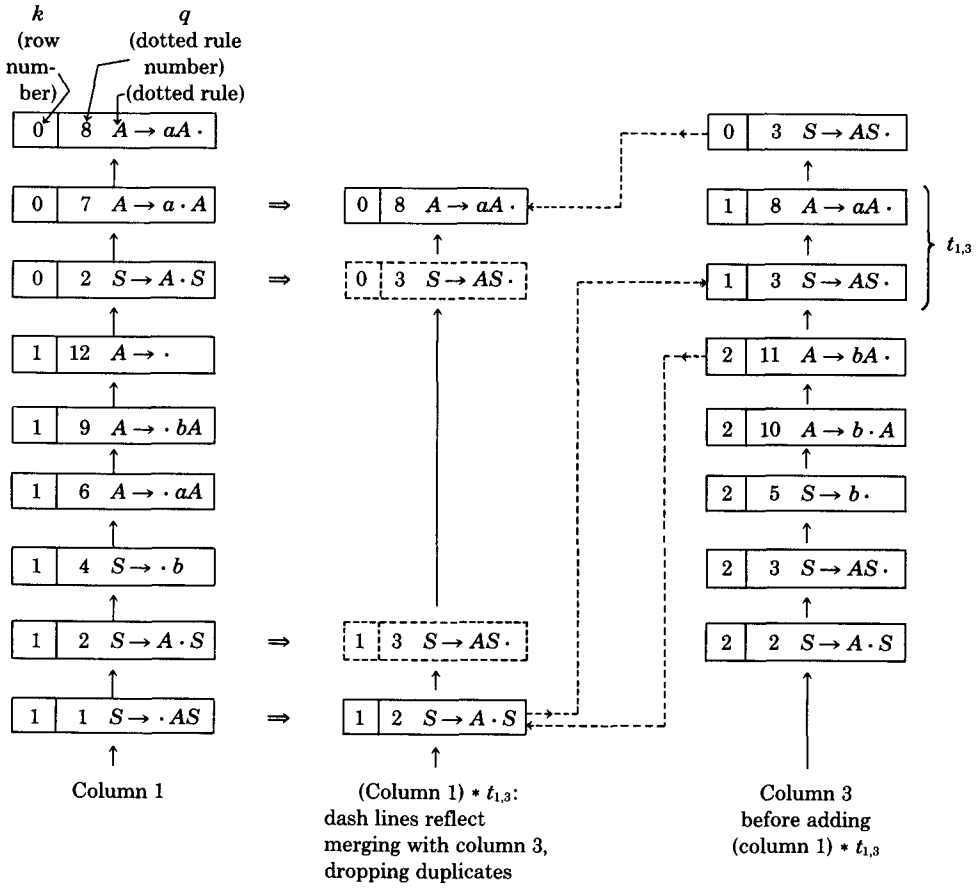

Fig. 4. First data structure for **t**.

illustrate the same situation for each of the four data structures discussed below, although for clarity we only show the representations of the relevant parts of the matrix, namely, column 1 and column 3 both before and after adding $\vec{t}_1 * t_{1,3}$.

The first of the four methods is the simplest. Assume that all dotted rules have been assigned numbers, so that if q corresponds to $A \to \alpha \cdot B\beta$, then $q + 1$ corresponds to $A \to \alpha B \cdot \beta$. Then we store each column as a list of ordered pairs (k, q), where k is a row number and q is (the number of) a dotted rule. Thus the pair (k, q) will be in the list for column j if and only if $q \in t_{k,j}$. Further, the list will be ordered by decreasing k, and secondarily by increasing q. (See Figure 4.)

It would be easy to implement Algorithm 2.4 using this representation and the "STARPROD1" *-product. To carry out the basic step "$\vec{t}_j := \vec{t}_j \cup \vec{t}_k * t_{k,j}$," we do the following. First, scan the portion of the jth list whose first components are k (i.e., the portion representing $t_{k,j}$) and construct the set $R'' = \text{CHNTO}(\text{FINAL}(t_{k,j}))$ for use in "STARPROD1." Note that at the end of this scan we can leave a pointer to the first item of the form $(k + 1, q)$, so that when we repeat this process for $k + 1$ we do not have to start searching the list from

the beginning to find the segment corresponding to $t_{k+1,j}$. Second, we form $\vec{t}_j := \vec{t}_j \cup \vec{t}_k \times R''$, by merging $\vec{t}_k \times R''$ into the list for column j. The numbering scheme for dotted rules is such that $q \times R''$ will always be a set of the form $\{q + 1, q + 2, \ldots, q + l\}$, where $l \geq 0$. Thus it would be easy to construct an ordered list representing $\vec{t}_k \times R''$ while making one sequential pass over the ordered list representing $\vec{t}_k$. It is then trivial to merge this ordered list into the ordered list for t_j, with duplicates discarded. However, since generating and discarding duplicates may be expensive (recall the example following ×PROD), it is probably better to combine the processing of list k with the duplicate checking and merging into list j, rather than separating the two processes. Thus the overall processing consists of sequentially scanning the portion of the jth list corresponding to $t_{k,j}$, then sequentially scanning the kth list in parallel with the portion of the jth list at and above row k. (See Figure 4.)

The running time of this procedure can be estimated easily. Let E be the total number of entries in columns 0 through $j - 1$. As k runs from $j - 1$ down to 0, we will look at each of the columns to the left of j exactly once, for a total of E inspections. As each column is processed, we run through an equal number of rows of the jth column. Assuming the density of entries in the jth column is about the same as in the other columns, we will look at list entries in the jth column about E times also. Thus the total work involves processing about $2E$ list entries. Notice that the method automatically takes advantage of sparseness in the matrix for both storage and time efficiency, rather than having its performance directly dictated by the size of the grammar and/or input.

The one drawback of this method is that we must search all the entries of each column, even though only a few dotted rules of interest may be found. This causes the algorithm to use $\Omega(n^3)$ times[8] on unambiguous grammars when time $\mathcal{O}(n^2)$ is achievable. The following example demonstrates this phenomenon. However, in practical situations where the grammar is ambiguous and the matrix is very sparse, it is not clear whether this effect is a serious impediment.

Consider the following (unambiguous) grammar:

$$
\begin{aligned}
S &\to A\not{c} \mid BC\$ \\
A &\to aAB \mid \Lambda \\
B &\to aB \mid b \mid c \\
C &\to bCD \mid \Lambda \\
D &\to bD \mid c
\end{aligned}
$$

and the input $a^p b^p c^p$ of length $n = 3p$. We show that for $0 \leq i \leq p$, the $(p + i)$th column contains $p - i$ dotted rules $A \to aA \cdot B$, and the $(p + i)$th row contains $p - i$ dotted rules $C \to bCD \cdot$, positioned in the matrix as indicated in Figure 5. The algorithm described above will process each of the $p - i$ dotted rules $C \to bCD \cdot$ in row $p + i$ by scanning all of column $p + i$. The total time for this scan will be $\sum_{i=0}^{p} (p - i)^2 = \Omega(p^3) = \Omega(n^3)$. Actually, each of those columns has only one item of interest, namely, $C \to b \cdot CD$ in $t_{p+i,p+i+1}$. To establish our claim, note

[8] Ω is used for lower bounds analogously to the use of $\mathcal{O}$ for upper bounds. $f(n) = \Omega(g(n))$ if and only if there exist $c > 0$, $n_0 > 0$ for all $n > n_0$, $f(n) \geq cg(n)$ (cf. [15]).

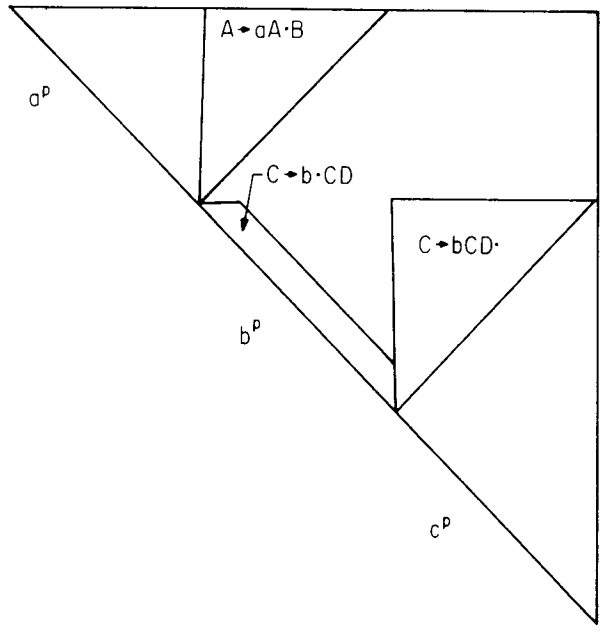

Fig. 5. Example for time n^3 with unambiguous grammar.

that $A \Rightarrow^* a^k(a^*(b \cup c))^k$, $k \geq 0$, and in particular, $A \Rightarrow^* a^k b^l$, $k \geq l \geq 0$. Further, $S \Rightarrow^* a^i AB^i\not{e}$, so the dotted rule $A \rightarrow aA \cdot B$ belongs at least in all $t_{i,j}$, where the substring from $i + 1$ to j follows a's and spans more a's than b's, i.e., the region $0 \leq i < p$ and $p \leq j < 2p - i$ sketched in Figure 5. Likewise $C \Rightarrow^* b^k c^l$, $k \geq l \geq 0$, and $S \Rightarrow^* a^p b b^i CD^i\$$, so $C \rightarrow bCD \cdot$ belongs in the region $p \leq i < 2p$ and $2p \leq j < 3p - i$, as claimed.

The second data structure we consider eliminates the possibility of scanning a long list of irrelevant entries. Instead of having all dotted rules in a column on a single list, we use a separate list for each column for each symbol U in the vocabulary, with all dotted rules of the form $A \rightarrow \alpha \cdot U\beta$ on the list for U. Additionally, we have a list per column for all items of the form $A \rightarrow \alpha \cdot$. (See Figure 6.) The algorithm proceeds as before, except that for each U with $U \rightarrow \gamma \cdot$ in $t_{k,j}$ we process only the list associated with U for column k. Every item on the list generates an addition to column j, although some may be duplicates. If column j were maintained as a single list and later converted into the multiple-list form, we would simplify the bookkeeping required while building column j. However, sequential merges into the list(s) representing column j may also introduce an extra factor of n into the time bound. This could be avoided by keeping pointers to the segments of the list(s) representing each row. This situation is shown in Figure 6. Alternatively, one could store the jth column as a hash table, with the pairs (k, q) as the keys to be hashed. Either scheme makes it unnecessary to keep the other lists sorted. With the hashing scheme, if the *-products are to be computed using STARPROD1, we should also link together all entries having the same "k" values (i.e., the entries in $t_{k,j}$). If STARPROD2 is used, it would be sufficient to put all entries on one list in the order in which they

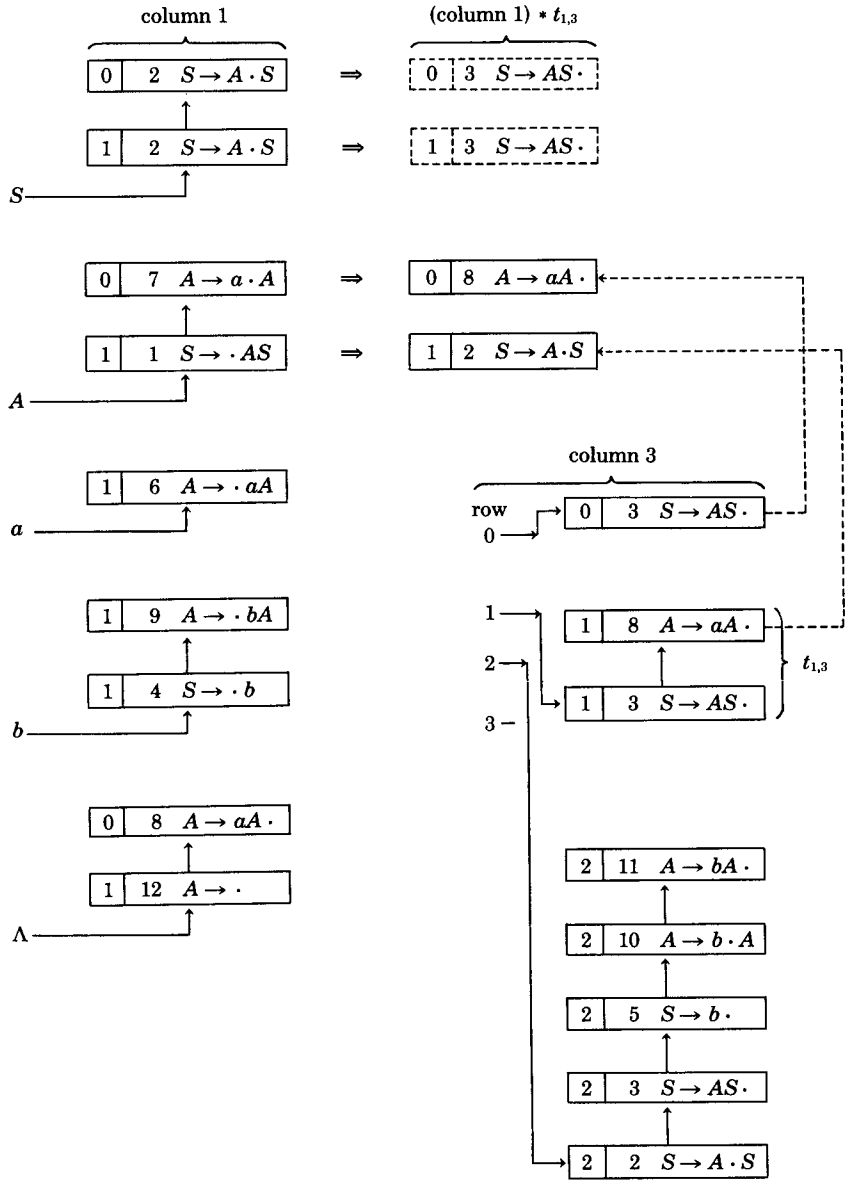

Fig. 6. Second data structure for **t**.

are added to the hash table. (This version is almost the same as Earley's algorithm; see Section 4.) With this approach one would want the various lists in each of the previous columns to be accessed through an array indexed by the symbol following the dot; i.e., DOTBEFORE(k, U) points to the list of items $A \rightarrow \alpha \cdot U\beta$ for column k. This array would be unnecessary if STARPROD1 were used; it would be sufficient to have a linked list of the nonempty lists for each column. This representation would probably take much less space.

With any of these schemes, the time spent constructing the jth column is

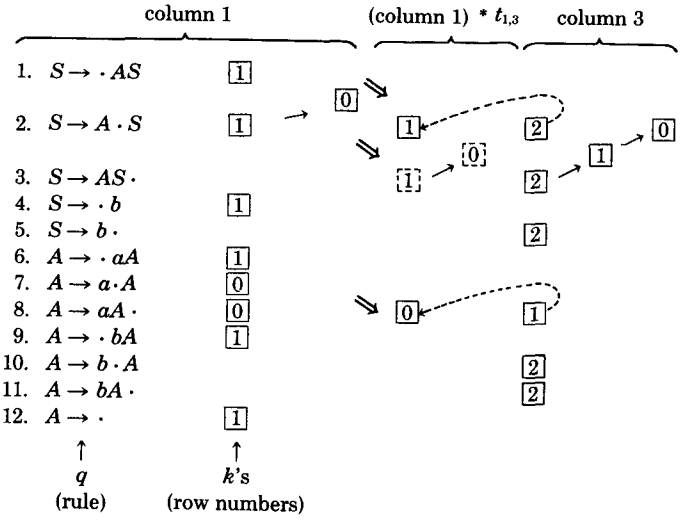

Fig. 7. Third data structure for t.

proportional to the number of additions made to it, *including* duplicates. If the grammar is unambiguous, there will be no duplicates, so the work will be $\mathcal{O}(n)$ per column, or $\mathcal{O}(n^2)$ overall.

A third representation for the matrix is a slight modification of the previous one. Instead of representing each column by a list of pairs (k, q) for each vocabulary symbol, we could store for each dotted rule q a list of the rows k in which it occurs. (See Figure 7.) This representation could save space, since the rule number would not need to be repeated in each entry. Further, in practice the lengths of the lists of rows associated with each dotted rule may be short enough so that building the lists by merging (as we did in the first method in this section) will prove acceptably efficient. (However, in the worst case it can be $\Omega(n^3)$ on unambiguous grammars.) In such a case we could have a simple algorithm with uniform representations for all columns, including the jth. For this version we will use STARPROD1. To form the product $\vec{t}_k * t_{k,j}$, we merge the list for q in column k with the list(s) for $q + 1$ $(q + 2, \ldots)$ in column j whenever $\{q\} \times \text{CHNTO}(\text{FINAL}(t_{k,j}))$ is not empty. If the set of lists for each column is kept as a list ordered by the dotted rule number, the product can be formed during one pass in parallel over the list of lists for column k and that for column j.

The fourth and last implementation we consider is a slight variation of the previous one, but is probably the most efficient of all the methods discussed. Instead of representing column k by keeping a list of row numbers for each dotted rule q, we keep one bit vector of length n, whose ith bit will be a 1 if and only if $q \in t_{i,k}$. The algorithm will be the same as before, except that merging lists of row numbers is accomplished by ORing the corresponding bit vectors. (See Figure 8.) The advantage of this method is that for most applications n will be small enough that each bit vector will fit in a few computer words, so the "OR" will take a few instructions at most. Thus the algorithm will take time $\mathcal{O}(n^2 \cdot \lceil n/w \rceil)$, where w is the computer word length. Of course, this is an $\mathcal{O}(n^3)$ algorithm, but for n in the range of practical interest it will behave more like an $\mathcal{O}(n^2)$ method, perhaps an

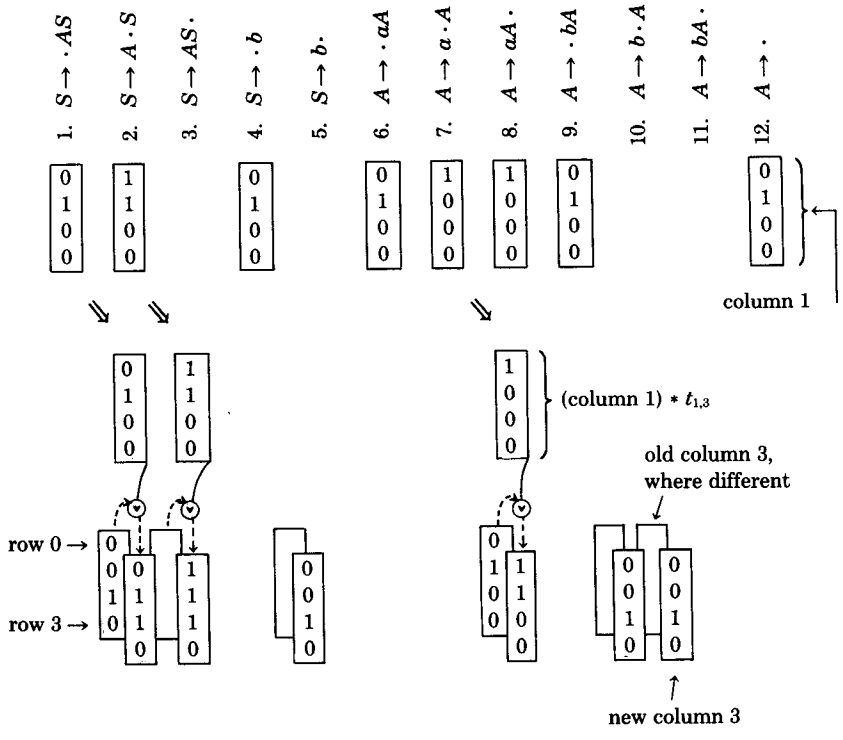

Fig. 8. Fourth data structure for **t**.

order of magnitude faster than the other methods, even for highly ambiguous inputs. Further, the amount of memory required for this method will be comparable to that used by the previous methods and may be even smaller. For example, for $n = 50$ the space for one bit vector is comparable to that for only two or three entries of a linked list of row numbers (as used in the third method, above). It is quite plausible that each dotted rule occurs an average of two or three times in each column of a 51 × 51 matrix, so the list representation would use about as much space as the bit-vector representation. The only drawback with this method is that it may be harder to recover the parses than with the other methods.

The important issue of recognition versus parsing has been ignored so far. The previous algorithms are recognizers, not parsers. For some applications that may be sufficient, but usually parsing is of interest. There are well-known algorithms for extracting some parse from the completed matrix in time $\mathcal{O}(n^2)$ [2, 12]; an improved version running in time $\mathcal{O}(n \log n)$ is also known [4]. For many applications those results are not sufficient. What is desired is a representation of *all* parses, so that we may find the one which is "best" according to some semantic criteria. Ruzzo [31] discusses modifications to the algorithms above to provide such information. Relative bounds on the complexities of algorithms for solving these problems may be found in [31, 32].

It is of some interest to reduce the size of the recognition matrix. Our algorithm ensures that all items entered in any column, say the jth, are "consistent" with the first j input symbols. However, we can make no such claim about consistency

with the portion of the input to the right of j. In fact, many dotted rules may be entered which are not part of any parse of the (whole) input. Obviously, an efficient way to eliminate these useless entries might save both space and time.

Several researchers have considered various "lookahead" techniques for preventing some of the useless entries from being generated [5, 8, 19, 35]. Their methods are directly applicable to our algorithm.

The lookahead methods were primarily aimed at improving performance on restricted classes of grammars such as LR(k) and LL(k). For these grammar classes the lookahead methods prevent the generation of enough items to guarantee linear-time operation (versus $\Omega(n^2)$ without lookahead). However, since these methods use only a bounded lookahead, they still may generate entries which are not consistent with the portion of the input beyond the lookahead. For grammars not in the restricted classes mentioned above, such as ambiguous grammars, many useless entries may be generated, even with lookahead. A method is described subsequently which eliminates all useless entries from the matrix after it has been completed. Of course, this method cannot reduce the time spent constructing the matrix. However, for many applications, processing by the semantic routines can be expected to take more time and possibly more space than constructing the recognition matrix. Thus, reducing the matrix before the semantic routines are run could be very profitable.

Before describing our reduction method, a more precise definition of *useful* entries seems appropriate. A dotted rule $A \rightarrow \alpha \cdot \beta$ in $t_{i,j}$ is *useful* if there is some $l \geq j$ such that $S \Rightarrow^* a_1 \cdots a_i A a_{l+1} \cdots a_n$, $\alpha \Rightarrow^* a_{i+1} \cdots a_j$, and $\beta \Rightarrow^* a_{j+1} \cdots a_l$. Thus a useful entry is one which is used in some derivation of the input string from S. The *reduced recognition matrix* contains just the useful entries from the recognition matrix. It is helpful to store with each dotted rule in the matrix a list of pointers to all of the dotted rules which caused it to be entered. We will call these pointers "parse pointers." Thus if $q_l \in t_{i,k_l}$, $r_l \in t_{k_l,j}$, and $s \in \{q_l\} \times \{r_l\}$, we would store with s in $t_{i,j}$ a list of the triples (k_l, q_l, r_l), or pairs of pointers to the list elements representing q_l in t_{i,k_l} and r_l in $t_{k_l,j}$, or at least a list of the k_l's. Notice that the number of parse pointers associated with each dotted rule in $t_{i,j}$ could be proportional to $j - i + 1$. Thus the storage needed for the matrix with parse pointers can be $\Omega(n^3)$.

If parse pointers are present in the original matrix, there is a simple method for constructing the reduced matrix; namely, the useful entries are exactly the entries which can be reached by following parse pointers from some entry $S \rightarrow \alpha \cdot$ in $t_{0,n}$. If the parse pointers are not initially present, they can be generated easily while the reduced matrix is being built as described below.

Note that any dotted rule $S \rightarrow \alpha \cdot$ in $t_{0,n}$ is useful. Further, if $A \rightarrow \alpha B \cdot \beta$ in $t_{i,j}$ is useful, and $A \rightarrow \alpha \cdot B\beta$ is in $t_{i,k}$ and $B \rightarrow \gamma \cdot$ is in $t_{k,j}$, then they are both useful too (see Figure 9). These observations are enough to provide an algorithm. First, mark all dotted rules $S \rightarrow \alpha \cdot$ in $t_{0,n}$. Second, for every i, j, every marked entry s in $t_{i,j}$, all q, r, and each k, $i \leq k \leq j$, if $q \in t_{i,k}$, $r \in t_{k,j}$ (or $k = j - 1$ and $r = a_j$), and $s \in \{q\} \times \{r\}$, then mark q and r. (If parse pointers are being built, add a pointer from s in $t_{i,j}$ to q and r in $t_{i,k}$ and $t_{k,j}$, respectively). Repeat the second step until no new entries can be marked. In fact, provided that care is taken with Λ- and chain-derivations, all useful entries will be marked if the second step is done in

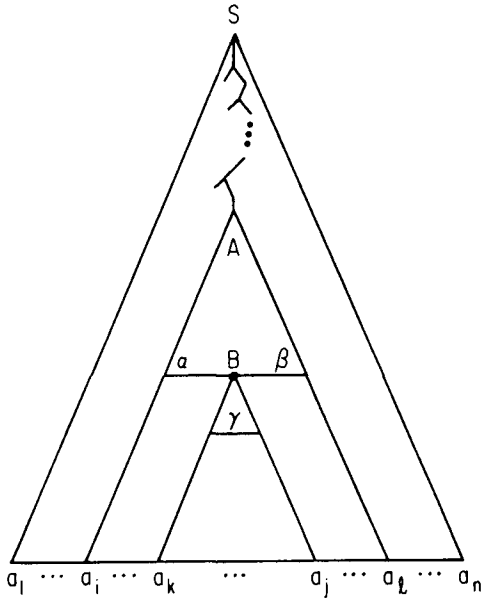

Fig. 9. Marking useful entries.

the order

for $j := n$ **downto** 0 **do**
  for $i = 0$ to j **do**
    for every marked s in $t_{i,j}$ **do** ... (3.10)

Note that this order is just the reverse of the order in which the matrix was computed.

A correctness proof will not be given in detail, but the idea is simple.[9] In the interest of brevity, we consider only Λ- and chain-free grammars. We noted above that all entries marked in this way are useful. To show that every useful entry is eventually marked, proceed by induction on the order in which the sets $t_{i,j}$ are processed in (3.10). Consider any useful entry r in $t_{i,j}$. Suppose r is of the form $B \to \gamma \cdot$. Then there must be some useful entry $A \to \alpha B \cdot \beta$ in $t_{i',j}$ for some $i' < i$ (see Figure 10). By the inductive hypothesis, $B \to \gamma \cdot$ must have been marked when $t_{i',j}$ was processed. Similarly, if r is of the form $A \to \alpha \cdot U\beta$, $U \in V$, it would have been marked when some useful entry $A \to \alpha U \cdot \beta$ in $t_{i,j'}$, for some $j' > j$ was processed (see Figure 11).

There is a straightforward implementation of this procedure for each of the representations of the matrix given earlier. Each implementation uses time roughly comparable to that required for the initial construction of the matrix. We briefly describe the implementation of the bit-vector version since it is more

[9] The correctness proof for the algorithm given in [15] to generate a parse after running Earley's recognizer is almost identical.

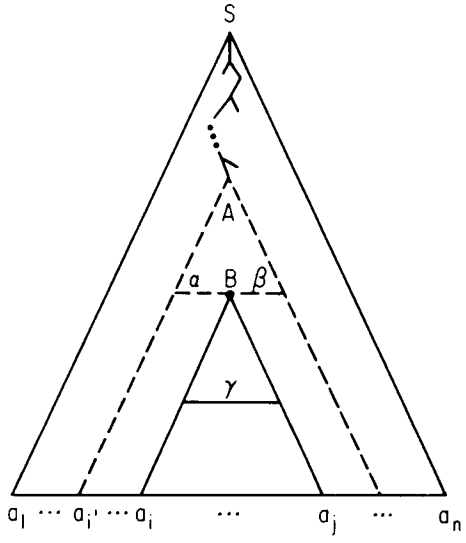

Fig. 10. r in $t_{i,j}$ is $B \rightarrow \gamma \cdot$.

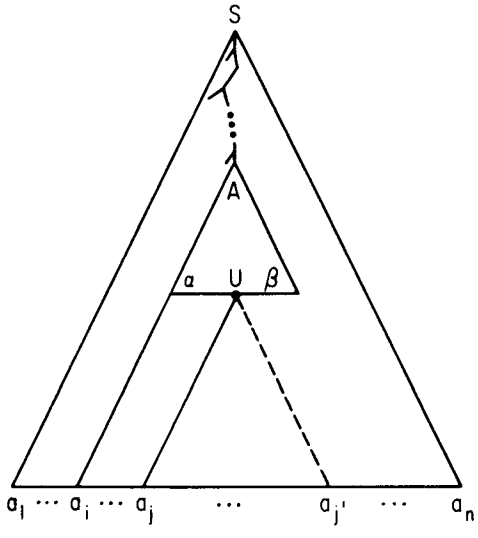

Fig. 11. r in $t_{i,j}$ is $A \rightarrow \alpha \cdot U\beta$.

interesting than the others. In the interest of simplicity we assume a Λ- and chain-free grammar, although it is not hard to handle the general case.

With each bit vector, associate another bit vector of equal length, initially all 0's, which will indicate which elements have been marked. Denote the vector corresponding to the dotted rule $A \rightarrow \alpha \cdot \beta$ in column j by $t_j(A \rightarrow \alpha \cdot \beta)$, and the associated mark vector by $m_j(A \rightarrow \alpha \cdot \beta)$. First, for each $S \rightarrow \alpha \cdot$ in $t_{0,n}$ set the bit in row 0 of $m_n(S \rightarrow \alpha \cdot)$ (i.e., mark $S \rightarrow \alpha \cdot$ in $t_{0,n}$). Process each $t_{i,j}$ in the order described above (3.10) as follows. For each U in FINAL($t_{i,j}$) (and $U = a_j$ if $i = j - 1$) and each $A \rightarrow \alpha \cdot U\beta$, let temp := $m_j(A \rightarrow \alpha U \cdot \beta) \wedge t_i(A \rightarrow \alpha \cdot U\beta)$. If temp

is not empty, mark all $U \rightarrow \gamma \cdot$ in $t_{i,j}$ and set $m_i(A \rightarrow \alpha \cdot U\beta) := m_i(A \rightarrow \alpha \cdot U\beta) \vee$ temp. Notice that $(A \rightarrow \alpha \cdot U\beta) \times (U) = A \rightarrow \alpha U \cdot \beta$, and $U \in \text{FINAL}(t_{i,j})$, so for each dotted rule $A \rightarrow \alpha \cdot U\beta$ in column i there will be an entry $A \rightarrow \alpha U \cdot \beta$ in the same row of column j. If any of the dotted rules $A \rightarrow \alpha U \cdot \beta$ in column j have previously been marked useful, then the corresponding entries $A \rightarrow \alpha \cdot U\beta$ in column i should be marked useful also. The bit vector temp $(= m_j(A \rightarrow \alpha U \cdot \beta) \wedge t_i(A \rightarrow \alpha \cdot U\beta))$ indicates exactly which marked entries $A \rightarrow \alpha U \cdot \beta$ in column j correspond to entries $A \rightarrow \alpha \cdot U\beta$ in column i. Thus if temp is not empty, we mark the useful entries in column i by setting $m_i(A \rightarrow \alpha \cdot U\beta) := m_i(A \rightarrow \alpha \cdot U\beta) \vee$ temp. Further, we also mark all the entries in $t_{i,j}$ which cause U to be in $\text{FINAL}(t_{i,j})$, namely, all dotted rules $U \rightarrow \gamma \cdot$ in $t_{i,j}$.

Figure 12 illustrates the operation of this algorithm on a simple example. We consider the entries marked when processing $t_{2,4}$. The three entries marked previously (when $t_{0,4}$ and $t_{1,4}$ were processed) are flagged by "*S" in the figure. $\text{FINAL}(t_{2,4}) = \{S\}$, so the only triple $(U, A \rightarrow \alpha \cdot U\beta, A \rightarrow \alpha U \cdot \beta)$ which is relevant when processing $t_{2,4}$ is the triple $(S, S \rightarrow A \cdot S, S \rightarrow AS \cdot)$. Thus we form temp $:= m_4(S \rightarrow AS \cdot) \wedge t_2(S \rightarrow A \cdot S)$, which is not zero, so $S \rightarrow ab \cdot$ in $t_{2,4}$ is marked and $m_2(S \rightarrow A \cdot S) := m_2(S \rightarrow A \cdot S) \vee$ temp. (The three entries marked as a result of these steps are flagged by "†S" in the figure. The entries marked in subsequent steps are flagged with "‡S".) In this example more than half of the dotted rules remain unmarked and thus are not part of any derivation of the input string.

The processing described above takes at most a bounded number of vector ANDs and ORs for each $t_{i,j}$, or $\mathcal{O}(n^2)$ in total. This time is comparable to the time needed to construct the matrix initially.

Parse pointers can be efficiently generated in the bit-vector implementation. Simply scan "temp" for "1" bits; whenever one is found in, say, the kth row, add to $A \rightarrow \alpha U \cdot \beta$ in $t_{k,j}$ a pointer to $A \rightarrow \alpha \cdot U\beta$ in $t_{k,i}$ and all $U \rightarrow \gamma \cdot$ in $t_{i,j}$. We can roughly estimate the efficiency of this method as follows. Suppose we want to end up with a matrix with parse pointers (reduced or not). Using one of the list representations, we get the full matrix in time proportional to the total number of parse pointers in it. Alternatively, we could use the bit-vector recognizer (which is perhaps 5 or 10 times as fast) and then reduce the matrix (equally as fast) while simultaneously constructing a list representation with parse pointers. Using special fast instructions (such as floating-point normalize) available on most computers for scanning the "temp" bit vector for 1 bits, we can expect the construction of the list representation to take about the same time as constructing a recognition matrix with the same number of entries. Thus if the reduced matrix is no bigger than about 60 percent of the size of the original matrix, this method will be faster than directly constructing the list-form recognition matrix. If the bit-vector recognizer is 10 times as fast as the list method, this approach is worthwhile if the reduction process removes only 20 percent of the items from the matrix.

Two schemes for removing useless entries while the matrix is being built are presented in [7, pp. 105–108] and further developed in [35]. Here, after the jth column is built, we retain a dotted rule $A \rightarrow \alpha \cdot \beta$ in $t_{i,k}$, $k \leq j$, only if there are derivations $S \Rightarrow^* a_1 \cdots a_i A\theta$, $\alpha \Rightarrow^* a_{i+1} \cdots a_k$, and $\beta\theta \Rightarrow^* a_{k+1} \cdots a_j\theta'$ for some

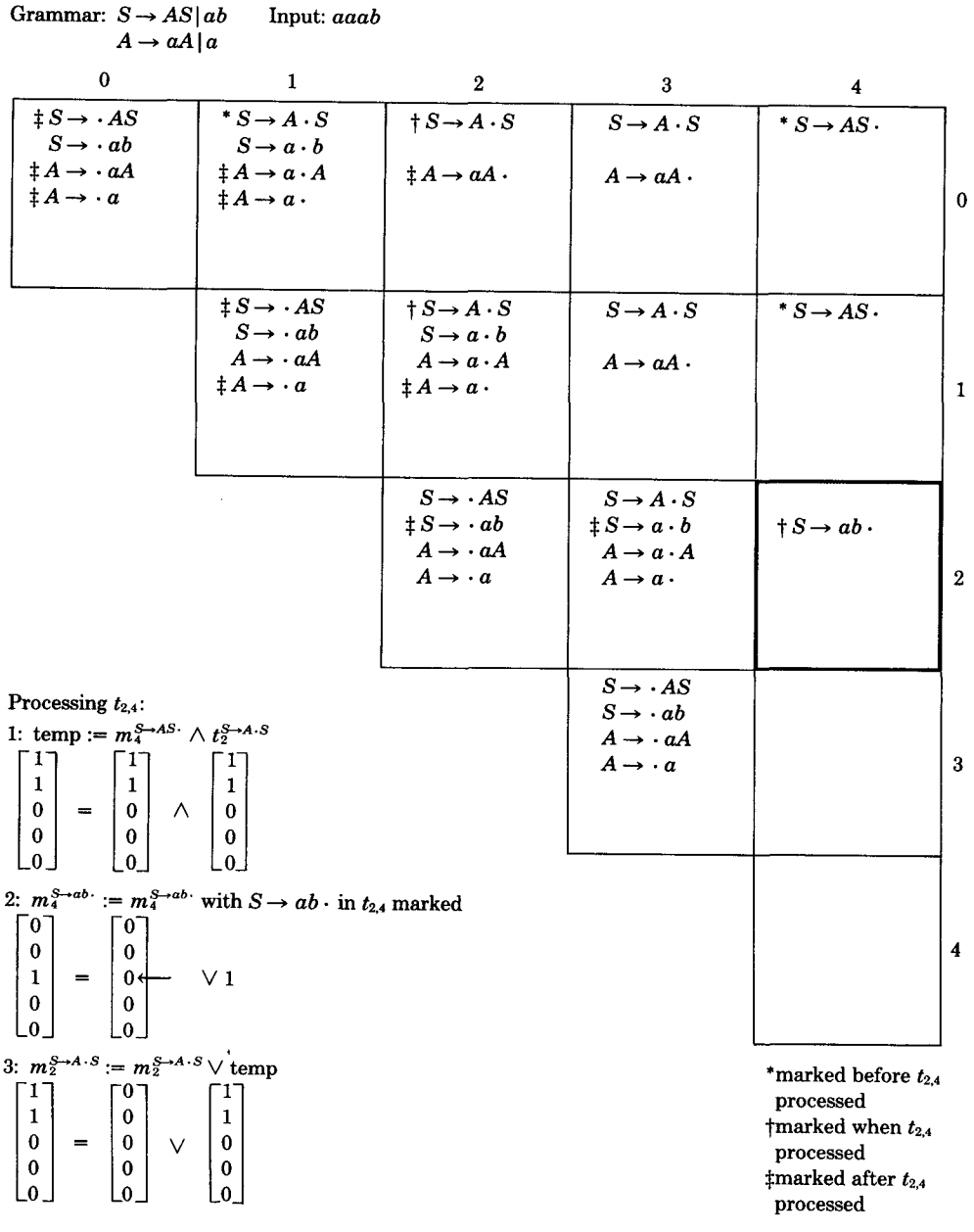

Fig. 12. Reducing the recognition matrix.

θ, θ' in V^*. One of Townley's methods proceeds by "losing" all pointers to useless items so that a garbage collector can reclaim the space occupied by them. Townley's other method uses a reference-count-type of garbage collection scheme tailored to the recognition algorithm. This scheme requires more programming

but may be much faster. It uses total time $\mathcal{O}(n^3)$, whereas the first method may take time $\Omega(n^3)$ for *each* call on the garbage collector.

4. CONNECTIONS AND COMPARISONS WITH OTHER RECOGNITION ALGORITHMS

There are some close connections between the apparently unrelated methods of Cocke–Kasami–Younger and Earley, which we can show by relating both to our algorithm. It is easy to see that the CKY and Earley algorithms use essentially a "dynamic programming" method: A derivation covering a larger portion of the input string is built by combining previously computed derivations of smaller portions. The most obvious difference between the two algorithms is that Earley's will work with any grammar, while CKY is restricted to grammars in Chomsky normal form. We show that this difference is actually quite superficial. The fundamental difference between the two algorithms turns out to be Earley's "predictor," which imposes an additional constraint on the allowable partial derivations, namely, that they be consistent with the beginning of the input string.

To explain the correspondence of our algorithm to the CKY method, we show that the predictor can be "weakened" or eliminated without affecting the correctness of the recognizer. Suppose we add to $t_{j,j}$ some "extra" dotted rule which would not ordinarily be there, say, $A \rightarrow \cdot\alpha$. Subsequent operations may introduce "extra" dotted rules into columns to the right of j, but *not* above row j. (Even if $A \rightarrow \alpha \cdot$ were entered somewhere on row j, the pasting step applied to $A \rightarrow \alpha \cdot$ would have no effect since column j would not have any entries with a dot in front of an A. Similar reasoning applies to other "extra" rules.). In particular, this addition would have no effect on $t_{0,n}$, so we still would have a correct recognizer. In fact, we could replace the predictor by the statement "$t_{j,j}$:= PREDICT(N);" and still have a correct recognizer. This gives the following algorithm.

Algorithm 4.1. This algorithm is identical to Algorithm 2.2, except that lines 1 ($t_{0,0}$:= PREDICT($\{S\}$)) and 11 ($t_{j,j}$:= PREDICT($\cup_{0 \le i \le j-1} t_{i,j}$)) become $t_{0,0}$:= PREDICT(N) and $t_{j,j}$:= PREDICT(N), respectively.

The characterization theorem for this algorithm is the following, which is analogous to the one for CKY.

LEMMA 4.1. *After running Algorithm* 4.1, $A \rightarrow \alpha \cdot \beta$ *is in* $t_{i,j}$ *if and only if* α *matches* $w_{i,j}$, *i.e.*, $\alpha \Rightarrow^* a_{i+1} \cdots a_j$.

PROOF. It follows directly from the definition that PREDICT(N) is the set of all dotted rules matching Λ, so the lemma holds for all $t_{j,j}$, $j \ge 0$. For the sets $t_{i,j}$, $0 \le i < j$, the proof is analogous to the proof of Theorem 2.1. □

Notice that we could set $t_{j,j}$ to any value such that

$$\text{PREDICT}\Big(\bigcup_{i<j} t_{i,j}\Big) \subseteq t_{j,j} \subseteq \text{PREDICT}(N)$$

without affecting the correctness of the recognizer. Adding "extra" items to $t_{j,j}$ in

this way seems to correspond to characteristic LR parsing [11] in roughly the same way that Earley's method [8] corresponds to LR(k) parsing [21].

It will be more convenient to work with a slightly different version of CKY—a version we call CKY1, which allows the addition of chain rules to Chomsky normal form grammars. (This normal form is called "canonical two form" [15].) The modification is very simple: Whenever a variable A is added to some $t_{i,j}$, we also add A' if $A' \to A$ (and likewise A'' if $A'' \to A'$, etc.). In the product notation we have been using, we just extend the definition of $\otimes$ as follows:

$$Q \otimes R = \{A \mid A \Rightarrow^* BC \text{ for some } B \in Q, C \in R\}.$$

We relate our algorithm to CKY1 by using a particular transformation which converts any grammar G into a grammar $G1$ which is in canonical two form. (For simplicity we assume that $\Lambda \notin L(G)$.) We show that our algorithm without the predictor working on G is the "same" as CKY1 working on the transformed grammar $G1$. We also give a transformation from an arbitrary grammar G to a grammar $G2$ in Chomsky form. This transformation can be used to relate our algorithm directly to CKY (rather than the variant of CKY), but the correspondence is more clearly visible with the simpler transformation. (Instead of removing the predictor from our algorithm, it is possible to add one to CKY; [6, 27] give such an algorithm. The same grammar transformations can be used to show the connections between the algorithms.)

The first transformation we want is the following: The nonterminals of $G1$ consist of symbols $\langle A \to \alpha \cdot \beta \rangle$ for every dotted rule $A \to \alpha \cdot \beta$ of G and symbols $\langle a \rangle$ for every terminal a of G. The productions of $G1$ are, for every pair of rules $A \to \alpha B \beta$ and $B \to \gamma$, every rule $A \to \alpha a \beta$ of G, and every a in Σ,

(1) $\langle A \to \alpha B \cdot \beta \rangle \to \langle A \to \alpha \cdot B \beta \rangle \langle B \to \gamma \cdot \rangle$,
(2) $\langle A \to \alpha B \cdot \beta \rangle \to \langle A \to \alpha \cdot B \beta \rangle$ if $B \Rightarrow_G^* \Lambda$,
(3) $\langle A \to \alpha B \cdot \beta \rangle \to \langle B \to \gamma \cdot \rangle$ if $\alpha \Rightarrow_G^* \Lambda$,
(4) $\langle A \to \alpha a \cdot \beta \rangle \to \langle A \to \alpha \cdot a \beta \rangle \langle a \rangle$,
(5) $\langle A \to \alpha a \cdot \beta \rangle \to \langle a \rangle$ if $\alpha \Rightarrow^* \Lambda$, and
(6) $\langle a \rangle \to a$.

It is obvious that $G1$ is in canonical two form. Further, it is easy to see that this grammar is equivalent to G. Rule 6 arises from the obvious transformation used to restrict terminals to rules of the form $A \to a$; rules 1 and 4 are a simple way to reduce all right-hand sides to length ≤ 2; and rules 2, 3, and 5 come from rules 1 and 4 by the obvious transformation used to eliminate Λ-rules. The exact correspondence between G and $G1$ is captured by the following lemma.

LEMMA 4.2. *Let $G = (V, \Sigma, P, S)$ be an arbitrary cfg such that $\Lambda \notin L(G)$. Let $G1$ be formed from G by the above construction. Then for any rule $A \to \alpha\beta$ of G and any $x \in \Sigma^+$, $\langle A \to \alpha \cdot \beta \rangle \Rightarrow_{G1}^* x$ if and only if $\alpha \Rightarrow_G^* x$.*

The proof is omitted.

The equivalence of the grammars G and $G1$ follows, since

$$x \in L(G) \qquad \text{iff} \quad \langle S \to \alpha \cdot \rangle \Rightarrow_{G1}^* x$$

for some rule $S \to \alpha$ for G. (The set of nonterminals $\langle S \to \alpha \cdot \rangle$ functions as the

"start symbol" of $G1$. To be more formal, one should introduce a new start symbol $\langle S \rangle$ and rules $\langle S \rangle \rightarrow \langle S \rightarrow \alpha \cdot \rangle$.) Finally, we are ready to state precisely the correspondence between the algorithms.

THEOREM 4.1. *Let $G = (V, \Sigma, P, S)$ be an arbitrary cfg with $\Lambda \notin L(G)$, and let $G1$ be constructed from G as described above. If Algorithm* 4.1 *and CKY*1 *(described above) are executed for the same input using grammars G and $G1$, respectively, then the nonterminal $\langle A \rightarrow \alpha \cdot \beta \rangle$ is in $t_{i,j}$ (CKY*1*) if and only if the dotted rule $A \rightarrow \alpha \cdot \beta$ is in $t_{i,j}$ (Algorithm* 4.1*).*

PROOF. By Lemmas 4.1 and 4.2 we have

$$\begin{aligned} \langle A \rightarrow \alpha \cdot \beta \rangle \in t_{i,j} \quad \text{(CKY1)} \quad & \text{iff} \quad \langle A \rightarrow \alpha \cdot \beta \rangle \Rightarrow_{G1}^{*} a_{i+1} \cdots a_j \\ & \text{iff} \quad \alpha \Rightarrow_{G}^{*} a_{i+1} \cdots a_j \\ & \text{iff} \quad A \rightarrow \alpha \cdot \beta \in t_{i,j} \quad \text{(Algorithm 4.1).} \end{aligned}$$ □

Not only are the two algorithms computing the same information, but we can show that their methods of computing it are the "same" also. CKY1 puts A in $t_{i,j}$ whenever it finds B in $t_{i,k}$ and C in $t_{i,j}$ with $A \rightarrow BC$. Similarly, our algorithm puts $A \rightarrow \alpha B \cdot \beta$ into $t_{i,j}$ whenever it finds $A \rightarrow \alpha \cdot B\beta$ in $t_{i,k}$ and $B \rightarrow \gamma \cdot$ in $t_{k,j}$. We can view this as a disguised version of the CKY method applied to the rule $\langle A \rightarrow \alpha B \cdot \beta \rangle \rightarrow \langle A \rightarrow \alpha \cdot B\beta \rangle \langle B \rightarrow \gamma \cdot \rangle$. Furthermore, whenever CKY1 adds A, it will also add A' if $A' \rightarrow A$ (and A'' if $A'' \rightarrow A'$, etc.). Our algorithm is very similar if we look closely at our definition of the $\times$- and $*$-products. Suppose we find some dotted rule $A \rightarrow \alpha \cdot BCD\beta$ and also find $B \rightarrow \gamma \cdot$. Naturally, we add $A \rightarrow \alpha B \cdot CD\beta$ since it is in $\{A \rightarrow \alpha \cdot BCD\beta\} \times \{B \rightarrow \gamma \cdot\}$. Further, if $C \Rightarrow^{*} \Lambda$ and $D \Rightarrow^{*} \Lambda$, then $A \rightarrow \alpha BC \cdot D\beta$ and $A \rightarrow \alpha BCD \cdot \beta$ are also in $\{A \rightarrow \alpha \cdot BCD\beta\} \times \{B \rightarrow \gamma \cdot\}$, so we will add them also. This action is the direct counterpart of CKY1 handling the chain rules $\langle A \rightarrow \alpha BC \cdot D\beta \rangle \rightarrow \langle A \rightarrow \alpha B \cdot CD\beta \rangle$ and $\langle A \rightarrow \alpha BCD \cdot \beta \rangle \rightarrow \langle A \rightarrow \alpha BC \cdot D\beta \rangle$ in $G1$. There is a similar correspondence for the other chain rules in $G1$.

One further modification of our algorithm is of interest. If we eliminate the predictor (as in Algorithm 4.1), all of the diagonal elements would be the same, independent of the input, so they could be eliminated by changing the limits on the loops and using a suitably modified "product" operation in place of both $\times$ and $*$, e.g., $\circledast$, where

$$Q \circledast R = (Q \times R) \cup \text{PREDICT}(N) * (Q \times R).$$

We then get an algorithm which is structurally identical to CKY but can be used for any grammar, not just one in Chomsky normal form.[10] It is also possible to apply Valiant's method to compute the matrix in time proportional to that for Boolean matrix multiplication. The modified product implicitly defines a transformation to Chomsky form which is similar to the one we discussed above but somewhat more complex. In the interest of completeness we present the grammar ($G2$) corresponding to this variation below.

For all dotted rules $A \rightarrow \alpha \cdot \beta$ in G we have

[10] A similar idea is suggested in problem 4.2.15 of [2].

(i) If $\alpha \Rightarrow^* a \in \Sigma$,

$$\langle A \to \alpha \cdot \beta \rangle \to a$$

(ii) if $\alpha \Rightarrow^* B \in N$, then for every rule $B \to \gamma U \delta$ with $\delta \Rightarrow^* \Lambda$ we have either

(a) if $U \in \Sigma$,

$$\langle A \to \alpha \cdot \beta \rangle \to \langle B \to \gamma \cdot U\delta \rangle \langle U \rangle$$

or (b) if $U \in N$, then for all $U \to \sigma$,

$$\langle A \to \alpha \cdot \beta \rangle \to \langle B \to \gamma \cdot U\delta \rangle \langle U \to \sigma \cdot \rangle$$

(iii) if $\alpha = \alpha_1 U \alpha_2$ with $\alpha_2 \Rightarrow^* \Lambda$, we have either

(a) if $U \in \Sigma$,

$$\langle A \to \alpha \cdot \beta \rangle \to \langle A \to \alpha_1 \cdot U\alpha_2\beta \rangle \langle U \rangle$$

or

(b) if $U \in N$, for all $U \to \sigma$,

$$\langle A \to \alpha \cdot \beta \rangle \to \langle A \to \alpha_1 \cdot U\alpha_2\beta \rangle \langle U \to \sigma \cdot \rangle$$

(iv) for every $a \in \Sigma$,

$$\langle a \rangle \to a$$

(Note that the first three cases are not mutually exclusive, and that the grammar is not necessarily reduced.) The correspondence between G and $G2$ is presented in the following proposition. The proof is omitted.

PROPOSITION 4.1. *Let $G = (V, \Sigma, P, S)$ be an arbitrary cfg with $\Lambda \notin L(G)$, and let $G2$ be constructed from G as described above. Then for all dotted rules $A \to \alpha \cdot \beta$ and nonnull strings $x \in \Sigma^+$,*

$$\alpha \Rightarrow_G^* x \quad \textit{iff} \quad \langle A \to \alpha \cdot \beta \rangle \Rightarrow_{G2}^* x.$$

Finally we consider the differences between the CKY algorithm and ours. As mentioned previously, the main difference is the predictor, which allows certain left context to be considered. From our characterization theorems, it is clear that

$$t_{i,j} \quad \text{with predictor (i.e., Algorithm 2.2)}$$

$$\subseteq t_{i,j} \quad \text{without (i.e., Algorithm 4.1),}$$

and it is easy to find examples where the containment is proper. Figure 13 gives such an example. Since fewer matrix entries usually means savings in both time and space, it is clear that the predictor usually does not hurt. However, the worst-case performance for both algorithms is $\mathcal{O}(n^3)$ time and $\mathcal{O}(n^2)$ space. For certain restricted grammar classes the bounds are also the same; for example, for unambiguous or linear grammars both may need $\mathcal{O}(n^2)$ time. There are other cases where our algorithm is asymptotically superior to CKY. It is not hard to see how this situation can arise. The predictor has a left-right bias which may be ideal for some grammars but useless for others. For example, our algorithm takes time $\mathcal{O}(n)$ on $S \to Sa \mid a$ while CKY takes $\Omega(n^2)$, but on the equally simple grammar $S \to aS \mid a$, both take $\Omega(n^2)$. A less artificial comparison is reported in

Grammar: $S \to Sa \mid a$ Input: aaa

t (with predictor, Algorithm 2.2)

$S \to \cdot Sa$ $S \to \cdot a$	$S \to a \cdot$ $S \to S \cdot a$	$S \to Sa \cdot$ $S \to S \cdot a$	$S \to Sa \cdot$ $S \to S \cdot a$

t (without predictor, Algorithm 4.1)

$S \to \cdot Sa$ $S \to \cdot a$	$S \to a \cdot$ $S \to S \cdot a$	$S \to Sa \cdot$ $S \to S \cdot a$	$S \to Sa \cdot$ $S \to S \cdot a$
	$S \to \cdot Sa$ $S \to \cdot a$	$S \to a \cdot$ $S \to S \cdot a$	$S \to Sa \cdot$ $S \to S \cdot a$
		$S \to \cdot Sa$ $S \to \cdot a$	$S \to a \cdot$ $S \to S \cdot a$
			$S \to \cdot Sa$ $S \to \cdot a$

Fig. 13. $t_{i,j}$ with predictor $\subseteq t_{i,j}$ without predictor.

[27]. Pratt states that use of the predictor gave a factor-of-5 reduction in the number of matrix entries generated. In his application (natural language processing) the semantic routines executed for each entry may be very expensive, so the savings may be appreciable. However, it should be emphasized that the savings achieved by the predictor are strongly dependent on the grammar and input being processed.

Next we consider the connections between our algorithm and Earley's as described in [8]. The notation used below follows the discussion of Earley's algorithm given in [2], which is closer to our notation than that originally used by Earley.

Earley's algorithm and ours are very similar in the information they gather and in the way it is gathered. Earley's algorithm constructs lists I_j, $0 \leq j \leq n$, of ordered pairs $(A \rightarrow \alpha \cdot \beta, i)$, where $A \rightarrow \alpha \cdot \beta$ is a dotted rule of G and $0 \leq i \leq n$. The entry $(A \rightarrow \alpha \cdot \beta, i)$ will appear on list I_j if and only if $A \rightarrow \alpha \cdot \beta$ follows w_i and matches $w_{i,j}$. Consequently, the set of entries in list I_j with second component i corresponds exactly to the set $t_{i,j}$ constructed by Algorithm 2.2 (and list I_j corresponds to the jth column of $\mathbf{t}$). One of the main steps of Earley's algorithm (the "completer") is to add $(A \rightarrow \alpha B \cdot \beta, i)$ to list I_j when some $(B \rightarrow \gamma \cdot, k)$ is found on list I_j and $(A \rightarrow \alpha \cdot B\beta, i)$ is on list I_k. This step is analogous to line 8 of Algorithm 2.2, noting that $A \rightarrow \alpha B \cdot \beta$ is in $t_{i,k} \times t_{k,j}$ (since $A \rightarrow \alpha \cdot B\beta \in t_{i,k}$ and $B \rightarrow \gamma \cdot \in t_{k,j}$).

Our algorithm and Earley's differ in three respects. One is the handling of Λ- and chain-derivations. Our algorithm precomputes this information, whereas Earley's will find a Λ- or chain-derivation by doing a series (equal in number to the length of the derivation) of predictor and completer steps. Two more fundamental differences between the algorithms are the choice of data structure (matrix versus list) and the order of computation. Earley's processes items added to list I_j in the order of their addition, whereas ours processes items added to column j in order of decreasing row number.

As we have seen, a variety of implementations of our algorithm is possible. Usually these implementations will be more efficient than comparable versions of Earley's algorithm. Some of our implementations have no counterpart at all within Earley's framework. While this efficiency and flexibility are a significant advantage over Earley's method, we feel that it really represents only one expression of the main advantage of our method, which is its simplicity and clarity. Our method also exposes certain otherwise obscure characteristics of the two algorithms. Several examples will be presented to emphasize these points.

One example of a feature of the algorithm made visible in our version is the relationship between the CKY and Earley methods, discussed above. As a second example, we consider the recognition of linear languages. A *linear* context-free grammar is one whose rules are all of the form

$$A \rightarrow x \qquad \text{or} \qquad A \rightarrow xBy$$

where $x, y \in \Sigma^*$, $A, B \in N$. It is known that both the CKY and Earley algorithms can be made to run in time $\mathcal{O}(n^2)$ on linear grammars [12], but the proof for Earley's algorithm is much less obvious than for CKY. Our algorithm also can work in time $\mathcal{O}(n^2)$ for linear grammars, and the proof is very simple. Suppose we

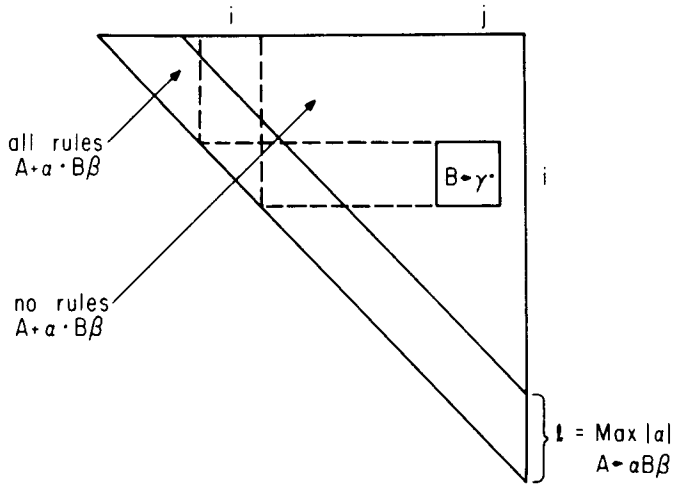

Fig. 14. Time n^2 for linear grammars.

find some rule $B \rightarrow \gamma \cdot$ in $t_{i,j}$; we must search column i for all rules of the form $A \rightarrow \alpha \cdot B\beta$. Since the grammar is linear, α and $\beta \in \Sigma^*$, so $A \rightarrow \alpha \cdot B\beta$ can be in $t_{k,i}$ if and only if $\alpha \Rightarrow^* w_{k,i}$ if and only if $\alpha = w_{k,i}$. If $l = \max_{A \rightarrow \alpha B\beta} \lg(\alpha)$, then $i - l \leq k \leq i$, i.e., all entries of the form $A \rightarrow \alpha \cdot B\beta$ are in a narrow band above the diagonal (see Figure 14). Thus we must search only a bounded portion of column i to find all $A \rightarrow \alpha \cdot B\beta$. The total processing time is then easily seen to be $\mathcal{O}(n^2)$. Furthermore, we can take advantage of the "shape" of the matrix to choose a better representation.

There are several examples of implementations of our algorithm for which there is either no counterpart, or only a less efficient one for Earley's algorithm. One method, using the representation shown in Figure 4, scans each column to the left of column j exactly once. An equivalently simple implementation of Earley's algorithm would have to scan list I_i, $i < j$, once for *each* dotted rule of the form $B \rightarrow \gamma \cdot$ found in $t_{i,j}$. This could easily be five to ten times more work than with our method. Furthermore, our method scans the columns to the left of j in the order $j - 1, j - 2, \ldots$, whereas Earley's method accesses the columns at "random" (i.e., in whatever order items are entered onto list I_j). The scan order allows our method to be done in time n^3 on a multitape Turing machine, whereas Earley's method takes time n^4 on a Turing machine [7]. Performance of the algorithm on a Turing machine may not seem of much practical interest. However, the much more regular pattern of memory accesses which allows the improved performance on Turing machines may also give greatly improved performance on virtual memory or paging systems, a factor which is of significant practical interest. A third example is the bit-vector implementation shown in Figure 8, for which there is no analog with Earley's method. Other examples include the subcubic algorithms discussed in the next section.

Two less well-known general context-free parsing methods are closely related to ours. The *nodal span* method of [6] is an adaptation of Earley's predictor to the CKY algorithm. A few years later a similar version appeared in [27]. This

algorithm requires a grammar in Chomsky normal form, so it suffers from the problem mentioned in Section 2.2—converting to Chomsky form may square the size of the grammar. Pratt's algorithm allows Chomsky form plus chain rules (i.e., canonical two form) as in CKY1 discussed above. It is easy to convert an arbitrary grammar to canonical two form with only a linear increase in size, so this aspect of Pratt's algorithm is preferable.

The use of a Chomsky or canonical two form grammar in these algorithms is attractive, since it simplifies the notation needed to state and program the algorithms. However, there is a hidden cost to this approach. If the algorithm stores information corresponding to dotted rules of the form $A \rightarrow B \cdot C$, then more space may be used than with our method; if such dotted rules are not stored, the algorithm may use more time, but less space, than ours. To see how this can happen, suppose that the original grammar G has a rule $A \rightarrow B_1 B_2 \cdots B_k$. In the equivalent canonical two form grammar $G1$ there will be $k - 1$ rules,

$$A \rightarrow C_{k-1} B_k$$

$$C_{k-1} \rightarrow C_{k-2} B_{k-1}$$

$$\vdots$$

$$C_2 \rightarrow B_1 B_2.$$

When some portion of the input matches $A \rightarrow B_1 \cdots B_k \cdot$, our algorithm will make, among others, the k entries $A \rightarrow B_1 \cdot B_2 \cdots B_k, \ldots, A \rightarrow B_1 B_2 \cdots B_k \cdot$. (We ignore entries $A \rightarrow \cdot B_1 \cdots B_k$ for reasons explained below.) An algorithm working with $G1$ must store $k - 1$ dotted rules, namely, $C_2 \rightarrow B_1 B_2 \cdot$, $C_3 \rightarrow C_2 B_3 \cdot, \ldots, A \rightarrow C_{k-1} B_k \cdot$. If it also stores the intermediate $k - 1$ entries $C_2 \rightarrow B_1 \cdot B_2$, $C_3 \rightarrow C_2 \cdot B_3, \ldots, A \rightarrow C_{k-1} \cdot B_k$, it will have used $(2k - 2)/k$ times as much space as our algorithm. (Our dotted rule $A \rightarrow B_1 B_2 \cdots B_i \cdot B_{i+1} \cdots B_k$ serves two purposes: it notes that $B_1 \cdots B_i$ has been matched, which is noted by $C_i \rightarrow C_{i-1} B_i \cdot$ in $G1$; and it notes that a B_{i+1} would be useful, which is noted by the separate dotted rule $C_{i+1} \rightarrow C_i \cdot B_{i+1}$ in $G1$.) Of course, there is only a small increase in space if the average length of a rule in G is nearly 2 (averaged with respect to frequency of occurrence in the recognition matrix). Alternatively, if the algorithm does not store the intermediate items $C_i \rightarrow C_{i-1} \cdot B_i$, then it will save space, namely, use only $(k - 1)/k$ times as much space as our algorithm. However, it will use more time since it must do extra searching to discover that, say, a C_{i-1} and a B_i can be combined to give a C_i. Note that these space estimates are only approximate, since a given dotted rule may be paired with another dotted rule to generate a third dotted rule several times (or no times). The published versions of both the Cocke–Schwartz [6] and Pratt algorithms use the space-saving form, but Pratt's implementation of his LINGOL system uses the faster form [27].

Three different methods of storing the predictor data are used by Pratt's method, the Cocke–Schwartz method, and ours. Pratt stores for the jth column,

$$\text{Goal}_j = \{C \mid A \rightarrow B \cdot C \text{ is in } t_{i,j} \text{ for some } i\}.$$

Cocke and Schwartz store

$$\text{Goal}_j^* = \{D \mid C \Rightarrow^* D\delta \text{ for some } \delta \in V^* \text{ and some } C \in \text{Goal}_j\}.$$

Our method stores

$$\text{PREDICT}(\text{Goal}_j) = \{D \rightarrow \cdot\,\alpha \mid D \in \text{Goal}_j^* \text{ and } D \rightarrow \alpha \text{ is a rule}\}.$$

(For simplicity, we assume that the same grammar is used for all three algorithms.) Pratt's method requires the least computation to generate the predictor information, and ours needs the most. For all subsequent tests using the predictor data, the situation is reversed: our method takes the least work and Pratt's takes the most. Our method also has the advantage of uniform representation: we do not need a special data structure for recording the predictor data. However, Pratt argues that PREDICT may generate a large number of irrelevant items (at least with the sort of grammars used for processing English). As an example, Pratt considers a grammar having rules $S \rightarrow NV \mid WV \mid V$ where S stands for "sentence," N for "noun phrase," V for "verb phrase," and W for "interrogatory phrase." For a sentence beginning "Why . . . ," only the second of the rules is applicable, but PREDICT will also generate entries for the dead ends $S \rightarrow \cdot\, NV$, $S \rightarrow \cdot\, V$, $N \rightarrow \cdot \cdots$, $V \rightarrow \cdot \cdots$, etc. The prediction certainly wastes space, and the time spent constructing these dotted rules could easily exceed the savings due to quicker tests, so Pratt's method may be faster. (Mere presence of the dotted rules does not slow subsequent operations if appropriate data structures are used.) If so, the Cocke–Schwartz form may be better still. Some experimentation with realistic grammars and inputs would be required to settle these issues.

One shortcoming of Pratt's method is that like the simple implementation of Earley's algorithm discussed above, it will scan column i once for *each* variable C found in $t_{i,j}$ (searching for items $A \rightarrow B \cdot C$). This search can be considerably slower than our algorithm which scans column i only once.

Several other related contributions have been made. Lyon [24] independently observed that Earley's algorithm could be modified to operate with its lists sorted by row number, and that this modification allowed an $\mathcal{O}(n^3)$ multitape Turing machine[11] version. Weicker [38] also gives a modification of Earley's algorithm which orders the columns as we have done and uses this ordering to get an $\mathcal{O}(n^2)$ bit-vector version which is somewhat different from ours: each column is represented by a *single* bit vector of length $n \cdot |G|$. Weicker extends this method to an $\mathcal{O}(n^2 \log n)$ algorithm for a uniform cost RAM[11] by encoding the bit vectors into integers having $|G| \cdot n \log n$ bits.

An unpublished paper by Floyd [9] describes another clever $\mathcal{O}(n^2)$ bit-vector recognizer, in this case a version of CKY. Floyd's method stores the transpose of the (upper triangular) recognition matrix in (the lower triangle of) the same matrix, so that a row–column product may be formed by ANDing the row with the transpose of the column. (The column-oriented bit vector algorithm described in Section 3 is probably faster, since it does not need to maintain the second copy of the matrix.) Another related work is by Townley [35], who also considered the possibility of precomputing information about Λ-derivations.

[11] See, e.g., [1] for definitions.

5. SUBCUBIC VERSIONS

All versions of the algorithm presented to this point have had a worst-case running time of $\Omega(n^3)$. Two versions of our algorithm running in less than n^3 time are presented here. These algorithms are theoretically interesting, but are not of practical utility, since they are faster than the n^3 methods only for unrealistically large values of n.

The first subcubic version uses Valiant's method to give an algorithm which runs in the same time bound as matrix multiplication. In Section 4 it was noted that our algorithm could be modified so that Valiant's method could be used to produce a matrix $t_{i,j} = \{A \to \alpha \cdot \beta \mid \alpha \Rightarrow^* a_{i+1} \cdots a_j\}$ for $i < j$ (note that $i \neq j$). Having done this in time $\mathcal{O}(n^{2.55})$, we can in time $\mathcal{O}(n^2)$ add the correct diagonal and "prune" the other entries to get a matrix identical to the one produced by Algorithm 2.2 as follows.

```
(1) t_{0,0} := PREDICT({S});
    for i := 0 to n − 1 do
    begin
(3)   for j = i + 1 to n do
        t_{i,j} := {A → α · β in t_{i,j} | some A rule A → α' · β' is in t_{i,i}};
(2)   t_{i+1,i+1} := PREDICT(∪_{k<i+1} t_{k,i+1})
    end;
```

The proof is simple. If $t_{i,i}$ is set correctly (i.e., as in Earley's algorithm) at (1) (which it clearly is) and at (2) (which will follow by induction), then the **for** loop at (3) will set the entire ith row correctly, since

$$A \to \alpha \cdot \beta \in t_{i,j} \qquad \text{(after (3))}$$

implies

$$A \to \alpha \cdot \beta \in t_{i,j} \qquad \text{(before (3))},$$

which implies

$$\alpha \Rightarrow^* a_{i+1} \cdots a_j. \tag{5.1}$$

Further,

$$A \to \alpha' \cdot \beta' \text{ is in } t_{i,i} \qquad \text{for some rule} \quad A \to \alpha'\beta'$$

which implies

$$S \Rightarrow^* a_1 \cdots a_i A\gamma \qquad \text{for some} \quad \gamma. \tag{5.2}$$

Conditions (5.1) and (5.2) characterize $t_{i,j}$ in our algorithm, of course.

Conversely, if (5.1) holds, then $A \to \alpha \cdot \beta$ must be in $t_{i,j}$ before (3), and if (5.2) holds, some $A \to \alpha' \cdot \beta'$ must be in $t_{i,i}$ (in particular, $A \to \cdot\, \gamma$ is in $t_{i,i}$ for all rules $A \to \gamma$). Thus $A \to \alpha \cdot \beta$ will be in $t_{i,j}$ after (3). By induction, all of column $i + 1$ will be correct before (2) is done, so $t_{i+1,i+1}$ will be correct.

If recognition alone is the goal, it makes no sense to prune the matrix, since Valiant's algorithm alone is a recognizer. However, for parsing it might be very useful to remove many of the "useless" items found in the CKY-type scheme we

started with. It would be interesting to know if *all* useless items (in the sense defined in Section 2.4) could be removed within the same time bound.

Valiant's method was based on fast general matrix multiplication methods [1]. As we show next, it is also possible to use the Boolean matrix multiplication scheme of [3]. This approach gives a recognizer requiring $\mathcal{O}(n^2/\log n)$ operations on bit vectors of length n, $\mathcal{O}(n^3/\log n)$ steps on a uniform cost RAM,[12] or $\mathcal{O}(n^3)$ steps on a logarithmic cost RAM.[12] Further, this is an on-line recognizer, in contrast to the algorithm given above, or Valiant's method.

Before giving the next algorithm, we mention several related results. Bit-vector machines defined in [30] allow unit time "AND," "NOT," and "SHIFT" operations on bit vectors of arbitrary length. The results in [30] give an $\mathcal{O}(\log^4 n)$ context-free language recognizer. Pratt [28] reports a method using time $\mathcal{O}(\log^3 n)$. Ruzzo [34] gives an $\mathcal{O}(\log^2 n)$ algorithm. These methods use very long vectors ($\Omega(n^{\log n})$ bits) and make essential use of the "shift" instruction. The method presented here uses vectors of length n and does not use shifts. Our method is the fastest known to us for machines using vectors of length $\mathcal{O}(n)$, even if shifts are allowed. The fastest known method for a uniform-cost RAM is the $\mathcal{O}(n^2 \log n)$ algorithm of Weicker [38]. The uniform-cost RAM is often used as a model of modern computers. It captures the fact that most operations take about the same time, regardless of the size of the numbers involved, provided that the numbers fit in one "word." However, the uniform-cost model deviates from reality in allowing arbitrarily large numbers, i.e., arbitrarily large "words." Weicker's algorithm exploits this feature of the uniform-cost model to achieve a fast algorithm by encoding an entire column of the recognition matrix as one long integer—$|G|n \log n$ bits. It has been suggested that a modification of the uniform-cost model which restricts integers to $k \log n$ bits (for some constant k) would be more consistent with the characteristics of current computers. Our $n^3/\log n$ algorithm falls within this restricted model. Results in [17] show that any algorithm running in time $T(n) \geq n \log n$ on a multitape Turing machine can be simulated in time $T(n)/\log T(n)$ on a uniform-cost RAM or in time $T(n)$ on a logarithmic-cost RAM. Galil [10] extends their result by showing that the simulating RAM can be on line if the Turing machine was on line. Thus our $n^3/\log n$ uniform-cost and n^3 log-cost results can now be obtained as corollaries of these more general theorems. However, our direct construction of these results and our $n^2/\log n$ bit-vector algorithm are still of interest. They are presented below, but only a sketch of the proofs will be given.

The idea of the construction is quite simple. Consider Algorithm 2.4. Pick some number $q > 0$. For each j from q to n the algorithm computes a specific function of columns $\vec{t}_0 \cdots \vec{t}_{q-1}$ and the sets $t_{0,j}, \ldots, t_{q-1,j}$ (namely, the portion of the inner loop "**for** $k := q - 1$ **downto** 0 **do** $\vec{t}_j := \vec{t}_j \cup \vec{t}_k * t_{k,j}$"). There are only a finite number, say s, of possible sets of dotted rules. If n is very large (greater than s^q), the same q-tuple of sets $t_{0,j}, \ldots, t_{q-1,j}$ must occur for different values of j. We could avoid recomputing those functions if the previous answers had been tabulated. In fact, it is easier simply to tabulate the function for all possible q-tuples of sets of dotted rules and then to compute the function by table look-up

[12] See, e.g., [1] for definitions.

for each q-tuple $t_{0,j}, \ldots, t_{q-1,j}$, $j \leq n$. Instead of taking about $n \cdot q$ steps to compute, we will now need about n (look-ups) + s^q (to build the table). Choosing $q = \log n$, the work is then $\mathcal{O}(n)$ rather than $\mathcal{O}(n \log n)$. Repeating this construction for the other $n/\log n$ groups of columns, the total work is $\mathcal{O}(n \cdot n/\log n) = \mathcal{O}(n^2/\log n)$ instead of $\mathcal{O}(n \log n \cdot n/\log n) = \mathcal{O}(n^2)$.

Our algorithm is given below. $\mathcal{D}$ denotes the set of all dotted rules; R denotes the tabulated function; $\lfloor k \rfloor$ is the floor of k.

Algorithm 5.1

for $q := 1, 2, 3, \ldots$ **until** $n \leq s^q$ **do**
begin (* Throughout, all vectors are s^q-vectors; the second index of R always has q components. *)

$$\vec{t}_0 := \begin{bmatrix} \text{PREDICT}(\{S\}) \\ \emptyset \\ \vdots \\ \emptyset \end{bmatrix};$$

$R := \emptyset$;
for all $v_0 \subset \mathcal{D}$ **do** $R(0, \langle v_0, \emptyset, \ldots, \emptyset \rangle) := \vec{t}_0 * v_0$;
for $j := 1$ **to** s^q **do**
begin
$\vec{t}_j := \vec{t}_{j-1} * \{a_j\}$;
for $k := \lfloor (j-1)/q \rfloor$ **downto** 0 **do**
$\vec{t}_j := \vec{t}_j \cup R(k, \langle t_{kq,j}, t_{kq+1,j}, \ldots, t_{kq+q-1,j} \rangle)$;
$t_{j,j} := \text{PREDICT}(\cup_{0 \leq i \leq j-1} t_{i,j})$;
$j' := \lfloor j/q \rfloor q$;
for all $v_{j'}, v_{j'+1}, \ldots, v_j \subset \mathcal{D}$ **do**
begin

$$\vec{v} := \begin{bmatrix} \left.\begin{matrix} \emptyset \\ \vdots \\ \emptyset \end{matrix}\right\} j' \\ v_{j'} \\ \vdots \\ v_j \\ \emptyset \\ \vdots \\ \emptyset \end{bmatrix};$$

$\vec{w} := \vec{v} \cup \vec{t}_j * v_j$;
$R(\lfloor j/q \rfloor, \langle v_{j'}, \ldots, v_j, \emptyset, \ldots, \emptyset \rangle) :=$
$w \cup R(\lfloor j/q \rfloor, \langle w_{j'}, \ldots, w_{j-1}, \emptyset, \emptyset, \ldots, \emptyset \rangle)$;
end;
end;
end;

Two points are noteworthy. First, an on-line algorithm cannot know n, the length of the input, in advance. We used n in our previous algorithms as a

notational convenience, but only to control termination of a loop; this could be done as easily by some kind of "end of input" indicator. However, in the current algorithm n determines q which determines the table structure, so it would be incorrect to assume it was known in advance. Instead, this algorithm will run with some q until $n > s^q$, then use $q + 1$; at this point it will go back to the beginning of the input and rebuild the tables based on the new value for q. This updating will turn out to affect the execution time only by a constant factor. Second, note that if the accesses to the table R are removed and the following function inserted, the resultant program is equivalent to Algorithm 2.4.

function $R(i, \langle v_{iq}, \ldots, v_{iq+q-1}\rangle)$;
begin

$$\vec{v} := \begin{bmatrix} \begin{matrix}\varnothing \\ \vdots \\ \varnothing\end{matrix} \Big\} iq \\ v_{iq} \\ \vdots \\ v_{iq+q-1} \\ \begin{matrix}\varnothing \\ \vdots \\ \varnothing\end{matrix} \end{bmatrix};$$

for $k := iq + q - 1$ **downto** iq **do**
$\vec{v} := \vec{v} \cup t_k * v_k$
return $(\vec{v})$;
end;

Basically, $R(i, \langle v_{iq}, \ldots\rangle)$ produces the vector reflecting the contribution of the ith group of columns (columns $iq \cdots iq + q - 1$) to some column with $v_{iq} \cdots$ in the ith group of rows (more precisely, $v_{iq} \cdots$ in the ith group of rows just after the contributions from column groups $\lfloor j - 1/q\rfloor$, $\lfloor j - 1/q\rfloor - 1, \ldots, i + 1$ have been added). Since Algorithm 5.1 with the function R is equivalent to Algorithm 2.4, to show correctness of Algorithm 5.1 with the table R, we must merely show that the table and function values are the same. This is fairly easy to show by induction on j. The key statement is the last "**for**" loop, where $R(\lfloor j/q\rfloor, \ldots)$ is extended to include the contribution due to column j. This gives the following proposition.

PROPOSITION 5.1. *Algorithm* 5.1 *correctly computes the recognition matrix; i.e.,* $A \to \alpha \cdot \beta$ *is in* $t_{i,j}$ *if and only if* $S \Rightarrow^* a_1 \cdots a_i A\gamma$ *and* $\alpha \Rightarrow^* a_{i+1} \cdots a_j$ *for some* γ.

Next we consider the time complexity of Algorithm 5.1. First we note that R is represented by a two-dimensional array, with $\langle v_0, \ldots, v_{q-1}\rangle$ considered to be an integer (think of it as a q-digit s-ary number). All conversions to and from this format can easily be done in a constant number of steps on a bit-vector machine. Thus the array references each take constant time. The "**for** k" loop takes $\mathcal{O}(j/q)$ steps. The body of the last **for** loop takes constant time, so the loop takes

$\mathcal{O}(s^{j \bmod q})$ steps. Thus as j runs from 1 to s^q, the "**for** j" loop takes

$$\mathcal{O}\left(\sum_{j=1}^{s_q} \frac{j}{q} + \frac{s^q}{q} \sum_{j=0}^{q-1} s^j\right) = \mathcal{O}\left(\frac{(s^q)^2}{q} + \frac{s^q}{q} \cdot s^q\right) = \mathcal{O}\left(\frac{(s^q)^2}{q}\right).$$

The outermost loop runs for $q = 1, \ldots, \lceil \log_s n \rceil$, so the total time is

$$\mathcal{O}\left(\sum_{q=1}^{\lceil \log_s n \rceil} \frac{(s^q)^2}{q}\right),$$

which is $\mathcal{O}(n^2/\log n)$ (as can be verified by induction). Since the bit vectors are all of length $\mathcal{O}(n)$, any bit-vector operation can be simulated in time $\mathcal{O}(n)$ on a uniform-cost RAM, so the time complexity on a RAM is $\mathcal{O}(n^3/\log n)$.

Note that this algorithm requires the storage of $\mathcal{O}(n^2/\log n)$ bit vectors or $\mathcal{O}(n^3/\log n)$ bits. Most of the other versions we have discussed use only $\mathcal{O}(n)$ bit vectors or $\mathcal{O}(n^2)$ bits.

6. CONCLUSION

A new general context-free language recognition algorithm has been presented, together with its implementation. The algorithm should be of practical utility since it is conceptually simple, can be efficiently implemented in a wide variety of situations, and can be used for any context-free grammar. It is also of theoretical interest since it exposes close connections between a number of previously known recognition methods, notably the method of Earley and that of Cocke, Kasami, and Younger. It is also possible to implement the algorithm in time asymptotically less than n^3.

REFERENCES

1. AHO, A.V., HOPCROFT, J.E., AND ULLMAN, J.D. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, Mass., 1974.
2. AHO, A.V., AND ULLMAN, J.D. *The Theory of Parsing, Translation, and Compiling, Vol. 1: Parsing.* Prentice-Hall, Englewood Cliffs, N.J., 1972.
3. ARLAZAROV, V.L., DINIC, E.A., KRONROD M.A., AND FARADZEV, I.A. On economical construction of the transitive closure of a directed graph. *Dokl. Akad. Nauk SSSR 194* (1970), 487–488 (in Russian). [English translation *Soviet Math. Dokl. 11*, 5 (1970), 1209–1210.]
4. BAYER, P.J. Personal communication, 1977.
5. BOUCKAERT, M., PIROTTE, A., AND SNELLING, M. Efficient parsing algorithms for general context free grammars. *Inf. Sci. 8*, 1 (Jan. 1975), 1–26.
6. COCKE, J., AND SCHWARTZ, J.I. *Programming Languages and Their Compilers.* Courant Institute of Mathematical Sciences, New York University, New York, 1970.
7. EARLEY, J. An efficient context-free parsing algorithm. Ph.D. Dissertation, Carnegie Mellon University, Pittsburgh, Pa., 1968.
8. EARLEY, J. An efficient context-free parsing algorithm. *Commun. ACM 13*, 2 (Feb. 1970), 94–102.
9. FLOYD, R.W. A machine-oriented recognition algorithm for context-free languages. Unpublished manuscript, 1969.
10. GALIL, Z. Two fast simulations which imply some fast string matching and palindrome-recognition algorithms. *Inf. Process. Lett. 4*, 4 (Jan. 1976), 85–87.
11. GELLER, M.M., AND HARRISON, M.A. Characteristic parsing: A framework for producing compact deterministic parsers, I. *J. Comput. Syst. Sci. 14*, 3 (June 1977), 265–317.

12. GRAHAM, S.L., AND HARRISON, M.A. Parsing of general context-free languages. In *Advances in Computers, Vol. 14.* Academic Press, New York, 1976, pp. 77–185.
13. GRAHAM, S.L., HARRISON, M.A., AND RUZZO, W.L. Online context free language recognition in less than cubic time. *Proc. 8th Ann. ACM Symp. on Theory of Computing*, Hershey, Pa., 1976, pp. 112–120.
14. GRIFFITHS, I., AND PETRICK, S. On the relative efficiencies of context-free grammar recognizers. *Commun. ACM 8*, 5 (May 1965), 289–300.
15. HARRISON, M.A. *Introduction to Formal Language Theory.* Addison-Wesley, Reading, Mass., 1978.
16. HAYS, D.G. Automatic language-data processing. In *Computer Applications in the Behavioral Sciences*, H. Borko, Ed., Prentice-Hall, Englewood Cliffs, N.J., 1962, pp. 394–423.
17. HOPCROFT, J.E., PAUL, W., AND VALIANT, L. On time versus space and related problems. Conf. Record IEEE 16th Ann. Symp. on Foundations of Computer Science, Berkeley, Calif., 1975, pp. 57–64.
18. IRONS, E.T. Experience with an extensible language. *Commun. ACM 13*, 1 (Jan. 1970), 31–40.
19. JONES, C.B. Formal development of correct algorithms: An example based on Earley's recognizer. *ACM SIGPLAN Notices 7*, 1 (Jan. 1972), 150–169.
20. KASAMI, T. An efficient recognition and syntax analysis algorithm for context free languages. Sci. Rep. AF CRL-65-758, Air Force Cambridge Research Laboratory, Bedford, Mass., 1965.
21. KNUTH, D.E. On the translation of languages from left to right. *Inf. Control 8* (1965), 607–639.
22. KUNO, S. The predictive analyzer and a path elimination technique. *Commun. ACM 8*, 7 (July 1965), 453–462.
23. LIPTON, R.J., AND SNYDER, L. On the optimal parsing of speech. Res. Rep. No. 37, Dep. Computer Science, Yale Univ., New Haven, Conn., 1974.
24. LYON, G. Syntax-directed least-errors analysis for context-free languages: A practical approach. *Commun. ACM 17*, 1 (Jan. 1974), 3–14.
25. MANACHER, G.K. An improved version of the Cocke-Younger-Kasami algorithm. *Comput. Lang. 3* (1978), 127–133.
26. MARQUE-PUCHEN, G. Analyses des langages algébriques. Thèse de 3° Cycle, Institute de Programmation, IP75-23, Université Paris VI, 1975 (in French).
27. PRATT, V.R., LINGOL—A progress report. *Advance Papers 4th Int. Joint Conf. on Artificial Intelligence*, Tbilisi, Georgia, USSR, 1975, 422–428.
28. PRATT, V.R. Personal communication, 1976.
29. PRATT, V.R. Personal communication, 1977.
30. PRATT, V.R., AND STOCKMEYER, L.J. A characterization of the power of vector machines. *J. Comput. Syst. Sci. 12* (1976), 198–221.
31. RUZZO, W.L. General context free language recognition. Ph.D. Dissertation, Dep. Computer Science, Univ. California, Berkeley, Calif., 1978.
32. RUZZO, W.L. On the complexity of general context-free language parsing and recognition. In *Automata, Languages, and Programming, Sixth Colloquium, Graz, July 1979*, H. Maurer, Ed., Vol. 71, Lecture Notes in Computer Science, Springer Verlag, Berlin, 1979, pp. 489–497.
33. RUZZO, W.L. Tree-size bounded alternation. Proc. 11th Ann. ACM Symp. on Theory of Computing, Atlanta, Ga., 1979, pp. 352–359.
34. RUZZO, W.L. An improved characterization of the power of vector machines. In preparation.
35. TOWNLEY, J.G. The measurement of complex algorithms. Ph.D. Dissertation, Harvard Univ., Cambridge, Mass., 1973 (TR14-73).
36. VALIANT, L. General context free recognition in less than cubic time. *J. Comput. Syst. Sci. 10* (1975), 308–315.
37. WEGBREIT, B. Studies in extensible programming languages. Ph.D. Dissertation, Harvard Univ., Cambridge, Mass., 1972.
38. WEICKER, R. Context free language recognition by a RAM with uniform cost criterion in time $n^2 \log n$. In *Symposium on New Directions and Recent Results in Algorithms and Complexity*, J.F. Traub, Ed., Academic Press, New York, 1976.
39. YOUNGER, D.H. Recognition of context-free languages in time n^3. *Inf. Control 10*, 2 (Feb. 1967), 189–208.

Received September 1979; revised April 1980; accepted April 1980