One of the earliest published parsing algorithms is the essentially left-corner parsing algorithm of Irons [1961]. Surveys of early parsing techniques are given by Floyd [1964b], Cheatham and Sattley [1963], and Griffiths and Petrick [1965].

Unger [1968] describes a top-down algorithm in which the initial and final symbols derivable from a nonterminal are used to reduce backtracking. Nondeterministic algorithms are discussed by Floyd [1967b].

One implementation of Domolki's algorithm is described by Hext and Roberts [1970].

The survey article by Cohen and Gotlieb [1970] describes the use of list structure representations for context-free grammars in backtrack and nonbacktrack parsing algorithms.

## 4.2. TABULAR PARSING METHODS

We shall study two parsing methods that work for all context-free grammars, the Cocke–Younger–Kasami algorithm and Earley's algorithm. Each algorithm requires $n^3$ time and $n^2$ space, but the latter requires only $n^2$ time when the underlying grammar is unambiguous. Moreover, Earley's algorithm can be made to work in linear time and space for most of the grammars which can be parsed in linear time by the methods to be discussed in subsequent chapters.

### 4.2.1. The Cocke–Younger–Kasami Algorithm

In the last section we observed that the top-down and bottom-up backtracking methods may take an exponential amount of time to parse according to an arbitrary grammar. In this section, we shall give a method guaranteed to do the job in time proportional to the cube of the input length. It is essentially a "dynamic programming" method and is included here because of its simplicity. It is doubtful, however, that it will find practical use, for three reasons:

(1) $n^3$ time is too much to allow for parsing.

(2) The method uses an amount of space proportional to the square of the input length.

(3) The method of the next section (Earley's algorithm) does at least as well in all respects as this one, and for many grammars does better.

The method works as follows. Let $G = (N, \Sigma, P, S)$ be a Chomsky normal form CFG with no $e$-production. A simple generalization works for non-CNF grammars as well, but we leave this generalization to the reader. Since a cycle-free CFG can be left- or right-covered by a CFG in Chomsky normal form, the generalization is not too important.

Let $w = a_1 a_2 \cdots a_n$ be the input string which is to be parsed according to $G$. We assume that each $a_i$ is in $\Sigma$ for $1 \leq i \leq n$. The essence of the

algorithm is the construction of a triangular *parse table* $T$, whose elements we denote $t_{ij}$ for $1 \leq i \leq n$ and $1 \leq j \leq n - i + 1$. Each $t_{ij}$ will have a value which is a subset of N. Nonterminal $A$ will be in $t_{ij}$ if and only if $A \overset{+}{\Longrightarrow} a_i a_{i+1} \cdots a_{i+j-1}$, that is, if $A$ derives the $j$ input symbols beginning at position $i$. As a special case, the input string $w$ is in $L(G)$ if and only if $S$ is in $t_{1n}$.

Thus, to determine whether string $w$ is in $L(G)$, we compute the parse table $T$ for $w$ and look to see if $S$ is in entry $t_{1n}$. Then, if we want one (or all) parses of $w$, we can use the parse table to construct these parses. Algorithm 4.4 can be used for this purpose.

We shall first give an algorithm to compute the parse table and then the algorithm to construct the parses from the table.

ALGORITHM 4.3

Cocke–Younger–Kasami parsing algorithm.

*Input.* A Chomsky normal form CFG $G = (\mathrm{N}, \Sigma, P, S)$ with no $e$-production and an input string $w = a_1 a_2 \cdots a_n$ in $\Sigma^+$.

*Output.* The parse table $T$ for $w$ such that $t_{ij}$ contains $A$ if and only if $A \overset{+}{\Longrightarrow} a_i a_{i+1} \cdots a_{i+j-1}$.

*Method.*

(1) Set $t_{i1} = \{A \mid A \longrightarrow a_i \text{ is in } P\}$ for each $i$. After this step, if $t_{i1}$ contains $A$, then clearly $A \overset{+}{\Longrightarrow} a_i$.

(2) Assume that $t_{ij'}$ has been computed for all $i$, $1 \leq i \leq n$, and all $j'$, $1 \leq j' < j$. Set

$$t_{ij} = \{A \mid \text{for some } k,\ 1 \leq k < j,\ A \longrightarrow BC \text{ is in } P,\ B \text{ is in } t_{ik}, \text{ and } C \text{ is in } t_{i+k,j-k}\}.\dagger$$

Since $1 \leq k < j$, both $k$ and $j - k$ are less than $j$. Thus both $t_{ik}$ and $t_{i+k,j-k}$ are computed before $t_{ij}$ is computed. After this step, if $t_{ij}$ contains $A$, then

$$A \Longrightarrow BC \overset{+}{\Longrightarrow} a_i \cdots a_{i+k-1} C \overset{+}{\Longrightarrow} a_i \cdots a_{i+k-1} a_{i+k} \cdots a_{i+j-1}.$$

(3) Repeat step (2) until $t_{ij}$ is known for all $1 \leq i \leq n$, and $1 \leq j \leq n - i + 1$. □

**Example 4.8**

Consider the CNF grammar $G$ with productions

†Note that we are not discussing in detail how this is to be done. Obviously, the computation involved can be done by computer. When we discuss the time complexity of Algorithm 4.3, we shall give details of this step that enable it to be done efficiently.

$$S \longrightarrow AA \mid AS \mid b$$
$$A \longrightarrow SA \mid AS \mid a$$

Let *abaab* be the input string. The parse table $T$ that results from Algorithm 4.3 is shown in Fig. 4.8. From step (1), $t_{11} = \{A\}$ since $A \longrightarrow a$ is in $P$ and $a_1 = a$. In step (2) we add $S$ to $t_{32}$, since $S \longrightarrow AA$ is in $P$ and $A$ is in both $t_{31}$ and $t_{41}$. Note that, in general, if the $t_{ij}$'s are displayed as shown, we can

| $j\uparrow$ | | $i\rightarrow$ 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| | 5 | $A, S$ | | | | |
| | 4 | $A, S$ | $A, S$ | | | |
| | 3 | $A, S$ | $S$ | $A, S$ | | |
| | 2 | $A, S$ | $A$ | $S$ | $A, S$ | |
| | 1 | $A$ | $S$ | $A$ | $A$ | $S$ |

**Fig. 4.8** Parse table $T$.

compute $t_{ij}$, $i > 1$, by examining the nonterminals in the following pairs of entries:

$$(t_{i1}, t_{i+1,j-1}), (t_{i2}, t_{i+2,j-2}), \ldots, (t_{i,j-1}, t_{i+j-1,1})$$

Then, if $B$ is in $t_{ik}$ and $C$ is in $t_{i+k,j-k}$ for some $k$ such that $1 \leq k < j$ and $A \longrightarrow BC$ is in $P$, we add $A$ to $t_{ij}$. That is, we move up the $i$th column and down the diagonal extending to the right of cell $t_{ij}$ simultaneously, observing the nonterminals in the pairs of cells as we go.

Since $S$ is in $t_{15}$, *abaab* is in $L(G)$. □

THEOREM 4.6

If Algorithm 4.3 is applied to CNF grammar $G$ and input string $a_1 \cdots a_n$, then upon termination, $A$ is in $t_{ij}$ if and only if $A \overset{+}{\Longrightarrow} a_i \cdots a_{i+j-1}$.

*Proof.* The proof is a straightforward induction on $j$ and is left for the Exercises. The most difficult step occurs in the "if" portion, where one must observe that if $j > 1$ and $A \overset{+}{\Longrightarrow} a_i \cdots a_{i+j-1}$, then there exist nonterminals $B$ and $C$ and integer $k$ such that $A \longrightarrow BC$ is in $P$, $B \overset{+}{\Longrightarrow} a_i \cdots a_{i+k-1}$, and $C \overset{+}{\Longrightarrow} a_{i+k} \cdots a_{i+j-1}$. □

Next, we show that Algorithm 4.3 can be executed on a random access computer in $n^3$ suitably defined elementary operations. For this purpose, we

shall assume that we have several integer variables available, one of which is $n$, the input length. An *elementary operation*, for the purposes of this discussion, is one of the following:

(1) Setting a variable to a constant, to the value held by some variable, or to the sum or difference of the value of two variables or constants;

(2) Testing if two variables are equal,

(3) Examining and/or altering the value of $t_{ij}$, if $i$ and $j$ are the current values of two integer variables or constants, or

(4) Examining $a_i$, the $i$th input symbol, if $i$ is the value of some variable.

We note that operation (3) is a finite operation if the grammar is known in advance. As the grammar becomes more complex, the amount of space necessary to store $t_{ij}$ and the amount of time necessary to examine it both increase, in terms of reasonable steps of a more elementary nature. However, here we are interested only in the variation of time with input length. It is left to the reader to define some more elementary steps to replace (3) and find the functional variation of the computation time with the number of nonterminals and productions of the grammar.

CONVENTION

We take the notation "$f(n)$ is $0(g(n))$" to mean that there exists a constant $k$ such that for all $n \geq 1$, $f(n) \leq kg(n)$. Thus, when we say that Algorithm 4.3 operates in time $0(n^3)$, we mean that there exists a constant $k$ for which it never takes more than $kn^3$ elementary operations on a word of length $n$.

THEOREM 4.7

Algorithm 4.3 requires $0(n^3)$ elementary operations of the type enumerated above to compute $t_{ij}$ for all $i$ and $j$.

*Proof.* To compute $t_{i1}$ for all $i$ merely requires that we set $i = 1$ [operation(1)], then repeatedly set $t_{i1}$ to $\{A \mid A \longrightarrow a_i \text{ is in } P\}$ [operations (3) and (4)], test if $i = n$ [operation (2)], and if not, increment $i$ by 1 [operation (1)]. The total number of elementary operations performed is $0(n)$.

Next, we must perform the following steps to compute $t_{ij}$:

(1) Set $j = 1$.

(2) Test if $j = n$. If not, increment $j$ by 1 and perform **line**($j$), a procedure to be defined below.

(3) Repeat step (2) until $j = n$.

Exclusive of operations required for **line**($j$), this routine involves $2n - 2$ elementary operations. The total number of elementary operations required for Algorithm 4.3 is thus $0(n)$ plus $\sum_{j=2}^{n} l(j)$, where $l(j)$ is the number of elementary operations used in **line**($j$). We shall show that $l(j)$ is $0(n^2)$ and thus that the total number of operations is $0(n^3)$.

The procedure **line**($j$) computes all entries $t_{ij}$ such that $1 \leq i < n - j + 1$. It embodies the procedure outlined in Example 4.8 to compute $t_{ij}$. It is defined as follows (we assume that all $t_{ij}$ initially have value $\varnothing$):

(1) Let $i = 1$ and $j' = n - j + 1$.
(2) Let $k = 1$.
(3) Let $k' = i + k$ and $j'' = j - k$.
(4) Examine $t_{ik}$ and $t_{k'j''}$. Let

$$t_{ij} = t_{ij} \cup \{A \mid A \longrightarrow BC \text{ is in } P,\ B \text{ in } t_{ik}, \text{ and } C \text{ in } t_{k'j''}\}.$$

(5) Increment $k$ by 1.
(6) If $k = j$, go to step (7). Otherwise, go to step (3).
(7) If $i = j'$, halt. Otherwise do step (8).
(8) Increment $i$ by 1 and go to step (2).

We observe that the above routine consists of an inner loop, (3)–(6), and an outer loop, (2)–(8). The inner loop is executed $j - 1$ times (for values of $k$ from 1 to $j - 1$) each time it is entered. At the end, $t_{ij}$ has the value defined in Algorithm 4.3. It consists of seven elementary operations, and so the inner loop uses $0(j)$ elementary operations each time it is entered.

The outer loop is entered $n - j + 1$ times and consists of $0(j)$ elementary operations each time it is entered. Since $j \leq n$, each computation of **line**($j$) takes $0(n^2)$ operations.

Since **line**($j$) is computed $n$ times, the total number of elementary operations needed to execute Algorithm 4.3 is thus $0(n^3)$. □

We shall now describe how to find a left parse from the parse table. The method is given by Algorithm 4.4.

ALGORITHM 4.4

Left parse from parse table.

*Input.* A Chomsky normal form CFG $G = (N, \Sigma, P, S)$ in which the productions in $P$ are numbered from 1 to $p$, an input string $w = a_1 a_2 \cdots a_n$, and the parse table $T$ for $w$ constructed by Algorithm 4.3.

*Output.* A left parse for $w$ or the signal "error."

*Method.* We shall describe a recursive routine **gen**($i, j, A$) to generate a left parse corresponding to the derivation $A \underset{\text{lm}}{\overset{+}{\Longrightarrow}} a_i a_{i+1} \cdots a_{i+j-1}$. The routine **gen**($i, j, A$) is defined as follows:

(1) If $j = 1$ and the $m$th production in $P$ is $A \longrightarrow a_i$, then emit the production number $m$.

(2) If $j > 1$, $k$ is the smallest integer, $1 \leq k < j$, such that for some $B$ in $t_{ik}$ and $C$ in $t_{i+k,j-k}$, $A \longrightarrow BC$ is a production in $P$, say the $m$th. (There

may be several choices for $A \longrightarrow BC$ here. We can arbitrarily choose the one with the smallest $m$.) Then emit the production number $m$ and execute **gen**$(i, k, B)$, followed by **gen**$(i + k, j - k, C)$.

Algorithm 4.4, then, is to execute **gen**$(1, n, S)$, provided that $S$ is in $t_{1n}$. If $S$ is not in $t_{1n}$, emit the message "error." □

We shall extend the notion of an elementary operation to include the writing of a production number associated with a production. We can then show the following result.

THEOREM 4.8

If Algorithm 4.4 is executed with input string $a_1 \cdots a_n$, then it will terminate with some left parse for the input if one exists. The number of elementary steps taken by Algorithm 4.4 is $0(n^2)$.

*Proof.* An induction on the order in which **gen** is called shows that whenever **gen**$(i, j, A)$ is called, then $A$ is in $t_{ij}$. It is thus straightforward to show that Algorithm 4.4 produces a left parse.

To show that Algorithm 4.4 operates in time $0(n^2)$, we prove by induction on $j$ that for all $j$ a call of **gen**$(i, j, A)$ takes no more than $c_1 j^2$ steps for some constant $c_1$. The basis, $j = 1$, is trivial, since step (1) of Algorithm 4.4 applies and uses one elementary operation.

For the induction, a call of **gen**$(i, j, A)$ with $j > 1$ causes step (2) to be executed. The reader can verify that there is a constant $c_2$ such that step (2) takes no more than $c_2 j$ elementary operations, exclusive of calls. If **gen**$(i, k, B)$ and **gen**$(i + k, j - k, C)$ are called, then by the inductive hypothesis, no more than $c_1 k^2 + c_1(j - k)^2 + c_2 j$ steps are taken by **gen**$(i, j, A)$. This expression reduces to $c_1(j^2 + 2k^2 - 2kj) + c_2 j$. Since $1 \leq k < j$ and $j \geq 2$, we know that $2k^2 - 2kj \leq 2 - 2j \leq -j$. Thus, if we chose $c_1$ to be $c_2$ in the inductive hypothesis, we would have $c_1 k^2 + c_1(j - k)^2 + c_2 j \leq c_1 j^2$. Since we are free to make this choice of $c_1$, we conclude the theorem. □

**Example 4.9**

Let $G$ be the grammar with the productions

(1) $S \longrightarrow AA$
(2) $S \longrightarrow AS$
(3) $S \longrightarrow b$
(4) $A \longrightarrow SA$
(5) $A \longrightarrow AS$
(6) $A \longrightarrow a$

Let $w = abaab$ be the input string. The parse table for $w$ is given in Example 4.8.

Since $S$ is in $T_{1,5}$, $w$ is in $L(G)$. To find a left parse for *abaab* we call routine **gen**(1, 5, $S$). We find $A$ in $t_{11}$ and in $t_{24}$ and the production $S \longrightarrow AA$ in the set of productions. Thus we emit 1 (the production number for $S \longrightarrow AA$) and then call **gen**(1, 1, $A$) and **gen**(2, 4, $A$). **gen**(1, 1, $A$) gives the production number 6. Since $S$ is in $t_{21}$ and $A$ is in $t_{33}$ and $A \longrightarrow SA$ is the fourth production, **gen**(2, 4, $A$) emits 4 and calls **gen**(2, 1, $S$) followed by **gen**(3, 3, $A$).

Continuing in this fashion we obtain the left parse 164356263.

Note that $G$ is ambiguous; in fact, *abaab* has more than one left parse. It is not in general possible to obtain all parses of the input from a parse table in less than exponential time, as there may be an exponential number of left parses for the input. □

We should mention that Algorithm 4.4 can be made to run faster if, when we construct the parse table and add a new entry, we place pointers to those entries which cause the new entry to appear (see Exercise 4.2.21).

### 4.2.2. The Parsing Method of Earley

In this section we shall present a parsing method which will parse an input string according to an arbitrary CFG using time $0(n^3)$ and space $0(n^2)$, where $n$ is the length of the input string. Moreover, if the CFG is unambiguous, the time variation is quadratic, and on most grammars for programming languages the algorithm can be modified so that both the time and space variations are linear with respect to input length (Exercise 4.2.18). We shall first give the basic algorithm informally and later show that the computation can be organized in such a manner that the time bounds stated above can be obtained.

The central idea of the algorithm is the following. Let $G = (N, \Sigma, P, S)$ be a CFG and let $w = a_1 a_2 \cdots a_n$ be an input string in $\Sigma^*$. An object of the form $[A \longrightarrow X_1 X_2 \cdots X_k \cdot X_{k+1} \cdots X_m, i]$ is called an *item* for $w$ if $A \longrightarrow X_1 \cdots X_m$ is a production in $P$ and $0 \leq i \leq n$. The dot between $X_k$ and $X_{k+1}$ is a metasymbol not in N or $\Sigma$. The integer $k$ can be any number including 0 (in which case the $\cdot$ is the first symbol) or $m$ (in which case it is the last).†

For each integer $j$, $0 \leq j \leq n$, we shall construct a list of items $I_j$ such that $[A \longrightarrow \alpha \cdot \beta, i]$ is in $I_j$ for $0 \leq i \leq j$ if and only if for some $\gamma$ and $\delta$, we have $S \overset{*}{\Longrightarrow} \gamma A \delta$, $\gamma \overset{*}{\Longrightarrow} a_1 \cdots a_i$, and $\alpha \overset{*}{\Longrightarrow} a_{i+1} \cdots a_j$. Thus the second component of the item and the number of the list on which it appears bracket the portion of the input derived from the string $\alpha$. The other conditions on the item merely assure us of the possibility that the production $A \longrightarrow \alpha\beta$

†If the production is $A \longrightarrow e$, then the item is $[A \longrightarrow \cdot, i]$.