中山大學

Principles of Compiler Construction

Lecture 6 Syntax Analysis (II)

Lecturer: CHANG HUIYOU

Note that most of these slides were created by:

Prof. Wen-jun LI (School of Software)

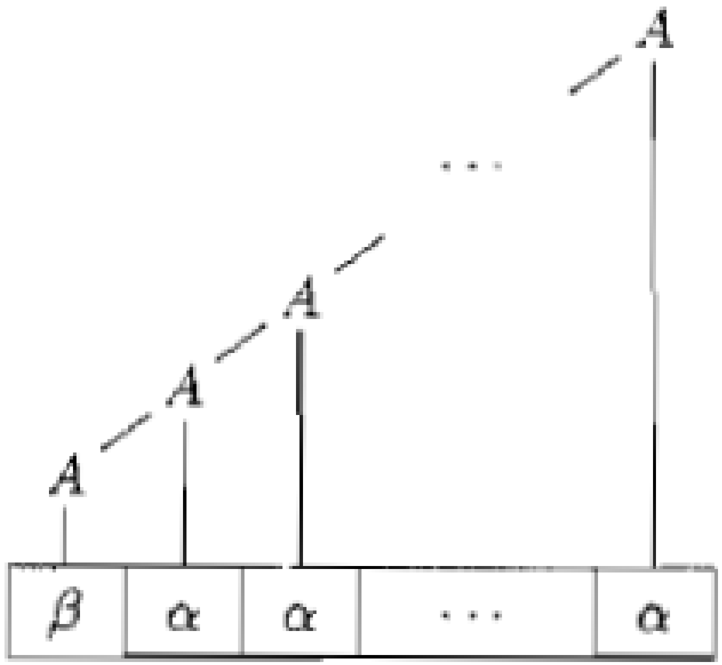
问题

给出一个CFG，如何判断一个字符串是否属于它定义的语言?

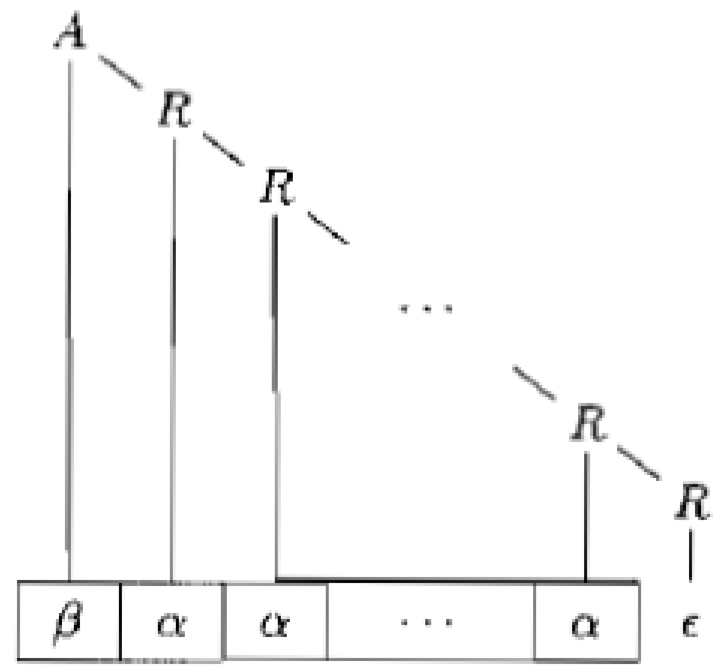
Recursive-Descent Parsing

```
boolean A(){
    Choose an A-production A → X_1 X_2 ··· X_k
    for (i=1 to k) {
        if ( X_i is a nonterminal  &&  ! X_i ())
            return false;
        else if (X_i  is a terminal){
            if (X_i equals the current input token t)
                get the next token t;
            else return false;
        }
    }
    return true;
}
```

Elimination of Left Recursion



$$A \rightarrow A\alpha$$



$$A \rightarrow \beta R$$
$$R \rightarrow \alpha R \mid \epsilon$$

Elimination of Left Recursion

A grammar is *left recursive* if it has a nonterminal A such that there is a derivation $A \stackrel{+}{\Rightarrow} A\alpha$ for some string α.

$$A \rightarrow A\alpha \mid \beta \quad \Longrightarrow \quad \begin{aligned} A &\rightarrow \beta A' \\ A' &\rightarrow \alpha A' \mid \epsilon \end{aligned}$$

Example

$$
\begin{aligned}
E &\rightarrow E + T \mid T \\
T &\rightarrow T * F \mid F \\
F &\rightarrow (E) \mid \mathbf{id}
\end{aligned}
\quad\Longrightarrow\quad
\begin{aligned}
E &\rightarrow T\ E' \\
E' &\rightarrow +\ T\ E' \mid \epsilon \\
T &\rightarrow F\ T' \\
T' &\rightarrow *\ F\ T' \mid \epsilon \\
F &\rightarrow (E) \mid \mathbf{id}
\end{aligned}
$$

Generally, ⋯

$$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \cdots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \cdots \mid \beta_n$$

$$\Downarrow$$

$$\begin{array}{l} A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \cdots \mid \beta_n A' \\ A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \cdots \mid \alpha_m A' \mid \epsilon \end{array}$$

However, ⋯

This procedure eliminates immediate left recursion, but does not eliminate left recursion involving derivations of two or more steps.

Example:

$$\begin{aligned} S &\rightarrow A\,a \mid b \\ A &\rightarrow A\,c \mid S\,d \mid \epsilon \end{aligned}$$

The nonterminal S is left recursive because $S \Rightarrow Aa \Rightarrow Sda$

The Algorithm for Elimination of Left Recursion

Algorithm 4.19: Eliminating left recursion.

INPUT: Grammar G with no cycles or ϵ-productions.

OUTPUT: An equivalent grammar with no left recursion.

METHOD: Apply the algorithm in Fig. 4.11 to G. Note that the resulting non-left-recursive grammar may have ϵ-productions. □

```
arrange the nonterminals in some order A1, A2, ..., An.
for ( each i from 1 to n ) {
        for ( each j from 1 to i − 1 ) {
                replace each production of the form Ai → Aj γ by the
                    productions Ai → δ1 γ | δ2 γ | ··· | δk γ, where
                    Aj → δ1 | δ2 | ··· | δk are all current Aj-productions
        }
        eliminate the immediate left recursion among the Ai-productions
}
```

Example

Dragon, p213

$$S \to A\,a \;|\; b$$
$$A \to A\,c \;|\; S\,d \;|\; \epsilon$$

$$\Longrightarrow$$

$$S \to A\,a \;|\; b$$
$$A \to b\,d\,A' \;|\; A'$$
$$A' \to c\,A' \;|\; a\,d\,A' \;|\; \epsilon$$

Discussion

Why is the algorithm correct?

What will happen if the input grammar has a cycle?

What will happen if the input grammar has an ϵ-productions?

Challenges

Give an algorithm to convert a grammar into an equivalent grammar that has no ϵ-productions.

Give an algorithm to convert a grammar into an equivalent grammar that has no cycles.

See Exercise 4.4.6 and 4.4.7 (DragonBook p232)

Recursive-Descent Parsing

```
boolean A(){
    Choose an A-production A → X1 X2 ··· Xk
    for (i=1 to k) {
        if ( Xi is a nonterminal && ! Xi ())
            return false;
        else if (Xi is a terminal){
            if (Xi equals the current input token t)
                get the next token t;
            else return false;
        }
    }
    return true;
}
```

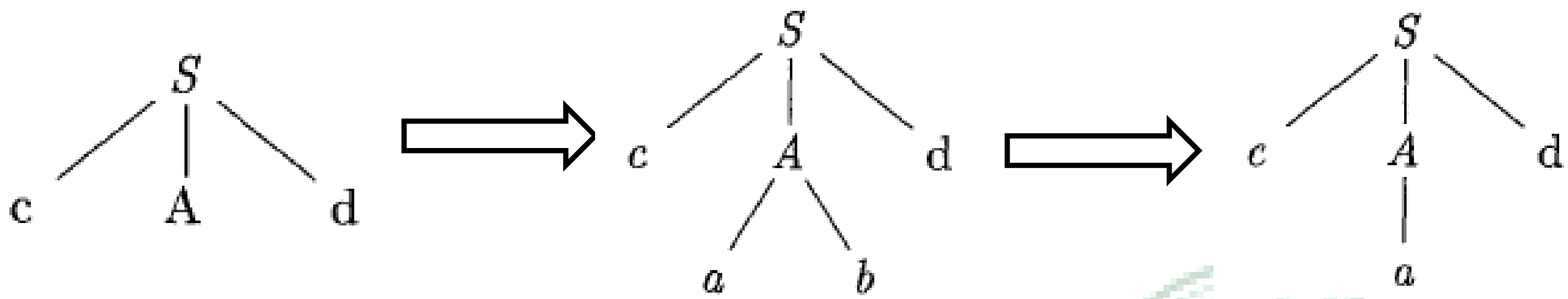
Which production should be chosen?

Naive idea: backtracking

Example

$$
\begin{aligned}
S &\rightarrow c\ A\ d \\
A &\rightarrow a\ b \mid a
\end{aligned}
$$

对于字符串cad



cabd != cad, 回溯

More Efficient Approaches

Backtracking is inefficient!

We need more efficient algorithms.

Recursive-Descent Parsing

```
boolean A(){
    Choose an A-production A → X1 X2 ⋯ Xk
    for (i=1 to k) {
        if ( Xi is a nonterminal  &&  ! Xi ())
            return false;
        else if (Xi is a terminal){
            if (Xi equals the current input token t)
                get the next token t;
            else return false;
        }
    }
    return true;
}
```

It would be fine if there were only one choice!

Lookahead & Left Factoring

Example

$$S \to cAd \mid bAd$$

$$A \to ab \mid a$$

对于输入cad

Left Factoring

INPUT: Grammar G.

OUTPUT: An equivalent left-factored grammar.

METHOD: For each nonterminal A, find the longest prefix α common to two or more of its alternatives. If $\alpha \neq \epsilon$ — i.e., there is a nontrivial common prefix — replace all of the A-productions $A \to \alpha\beta_1 \mid \alpha\beta_2 \mid \cdots \mid \alpha\beta_n \mid \gamma$, where γ represents all alternatives that do not begin with α, by

$$\begin{aligned} A &\to \alpha A' \mid \gamma \\ A' &\to \beta_1 \mid \beta_2 \mid \cdots \mid \beta_n \end{aligned}$$

Here A' is a new nonterminal. Repeatedly apply this transformation until no two alternatives for a nonterminal have a common prefix. □

Example

$$S \rightarrow i\,E\,t\,S \mid i\,E\,t\,S\,e\,S \mid a$$
$$E \rightarrow b$$

$$\Downarrow \text{ left factoring}$$

$$S \rightarrow i\,E\,t\,S\,S' \mid a$$
$$S' \rightarrow e\,S \mid \epsilon$$
$$E \rightarrow b$$

Question

Is left factoring enough for predictive parsing?

No!

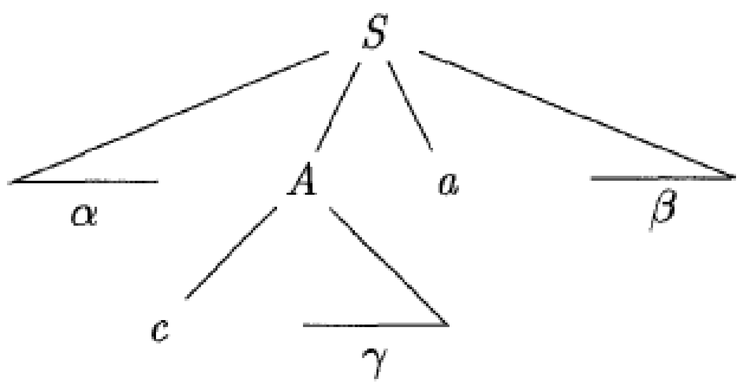
E.g.

S→Abc|Bda …

If we only lookahead only one character, we do not know which production should be chosen.

FIRST(α)

Define *FIRST*(α), where α is any string of grammar symbols, to be the set of terminals that begin strings derived from α. If $\alpha \stackrel{*}{\Rightarrow} \epsilon$, then ϵ is also in FIRST(α). For example, in Fig. 4.15, $A \stackrel{*}{\Rightarrow} c\gamma$, so c is in FIRST(A).



Consider two A-productions A→α|β, where FIRST(α) and FIRST(β) are disjoint. We can then choose between these A-productions by looking at the next input symbol a, since a can be in at most one of FIRST(α) and FIRST(β), not both.

Computing FIRST(X)

To compute FIRST(X) for all grammar symbols X, apply the following rules until no more terminals or ϵ can be added to any FIRST set.

1. If X is a terminal, then FIRST(X) $= \{X\}$.

2. If X is a nonterminal and $X \rightarrow Y_1 Y_2 \cdots Y_k$ is a production for some $k \geq 1$, then place a in FIRST(X) if for some i, a is in FIRST(Y_i), and ϵ is in all of FIRST(Y_1), … , FIRST(Y_{i-1}); that is, $Y_1 \cdots Y_{i-1} \stackrel{*}{\Rightarrow} \epsilon$. If ϵ is in FIRST(Y_j) for all $j = 1, 2, \ldots, k$, then add ϵ to FIRST(X). For example, everything in FIRST(Y_1) is surely in FIRST(X). If Y_1 does not derive ϵ, then we add nothing more to FIRST(X), but if $Y_1 \stackrel{*}{\Rightarrow} \epsilon$, then we add FIRST($Y_2$), and so on.

3. If $X \rightarrow \epsilon$ is a production, then add ϵ to FIRST(X).

Example

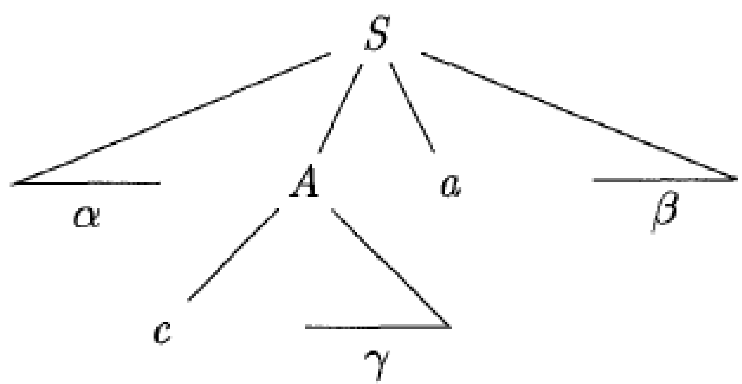
$$
\begin{array}{lcl}
E & \rightarrow & T\ E' \\
E' & \rightarrow & +\ T\ E' \mid \epsilon \\
T & \rightarrow & F\ T' \\
T' & \rightarrow & *\ F\ T' \mid \epsilon \\
F & \rightarrow & (\ E\ ) \mid \mathbf{id}
\end{array}
$$

Compute the FIRST(X) for each nonterminal X.

FOLLOW(A)

S→aA|є, if the next input symbol is not 'a', then will will choose S→ є; otherwise both productions can be chosen.

Define $FOLLOW(A)$, for nonterminal A, to be the set of terminals a that can appear immediately to the right of A in some sentential form; that is, the set of terminals a such that there exists a derivation of the form $S \stackrel{*}{\Rightarrow} \alpha A a \beta$, for some α and β



if A can be the rightmost symbol in some sentential form, then $ is in FOLLOW(A).

Computing FOLLOW(A)

To compute FOLLOW(A) for all nonterminals A, apply the following rules until nothing can be added to any FOLLOW set.

1. Place \$ in FOLLOW(S), where S is the start symbol, and \$ is the input right endmarker.
2. If there is a production $A \to \alpha B \beta$, then everything in FIRST(β) except ϵ is in FOLLOW(B).
3. If there is a production $A \to \alpha B$, or a production $A \to \alpha B \beta$, where FIRST(β) contains ϵ, then everything in FOLLOW(A) is in FOLLOW(B).

Practice

$$
\begin{array}{lcl}
E & \rightarrow & T\ E' \\
E' & \rightarrow & +\ T\ E' \mid \epsilon \\
T & \rightarrow & F\ T' \\
T' & \rightarrow & *\ F\ T' \mid \epsilon \\
F & \rightarrow & (\ E\ ) \mid \mathbf{id}
\end{array}
$$

Compute the FOLLOW(X) for each nonterminal X.

Construction of a Predictive Parsing Table

INPUT: Grammar G.

OUTPUT: Parsing table M.

METHOD: For each production $A \rightarrow \alpha$ of the grammar, do the following:

1. For each terminal a in $\text{FIRST}(\alpha)$, add $A \rightarrow \alpha$ to $M[A, a]$.

2. If ϵ is in $\text{FIRST}(\alpha)$, then for each terminal b in $\text{FOLLOW}(A)$, add $A \rightarrow \alpha$ to $M[A, b]$. If ϵ is in $\text{FIRST}(\alpha)$ and \$ is in $\text{FOLLOW}(A)$, add $A \rightarrow \alpha$ to $M[A, \$]$ as well.

If, after performing the above, there is no production at all in $M[A, a]$, then set $M[A, a]$ to **error** (which we normally represent by an empty entry in the table). □

Predictive Parsing Table

NON - TERMINAL	INPUT SYMBOL					
	id	+	*	(	)	$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
F	$F \rightarrow \mathbf{id}$			$F \rightarrow (E)$		

LL(1) Grammar

A grammar G is LL(1) if and only if whenever $A \to \alpha \mid \beta$ are two distinct productions of G, the following conditions hold:

1. For no terminal a do both α and β derive strings beginning with a.
2. At most one of α and β can derive the empty string.
3. If $\beta \stackrel{*}{\Rightarrow} \epsilon$, then α does not derive any string beginning with a terminal in FOLLOW(A). Likewise, if $\alpha \stackrel{*}{\Rightarrow} \epsilon$, then β does not derive any string beginning with a terminal in FOLLOW(A).

第一个L：输入字符串从左边开始扫描
第二个L：得到的推导是最左推导
(1)：向前看1个输入符号（或单词）

Example

$$
\begin{aligned}
S &\rightarrow iEtSS' \mid a \\
S' &\rightarrow eS \mid \epsilon \\
E &\rightarrow b
\end{aligned}
$$

Non - terminal	Input Symbol					
	a	b	e	i	t	$\$$
S	$S \rightarrow a$			$S \rightarrow iEtSS'$		
S'			$S' \rightarrow \epsilon$ $S' \rightarrow eS$			$S' \rightarrow \epsilon$
E		$E \rightarrow b$				

Since M[S', e] contains more than one production, the grammar is not LL(1).

作业

Week07.pdf

Implementing the Parser

Two ways:

- **Recursive**
- **Non-recursive， table-driven**

Coding for Recursive Predictive Parser

A→aBb

```
boolean  A(){
    return (match(‘a’) && B() && match(‘b’));
}
```

```
boolean match(Token t){
    if (lookahead == t){
        lookahead = lexer.nextToken();
        return true;
    else return false;
}
```

Coding for Recursive Predictive Parser

A→aBb|bAC

```
boolean  A(){
    if (lookahead == 'a')
        return (match('a') && B() && match('b'));
    else if (lookahead == 'b')
        return (match('b') && A() && C());
    else return false;
}
```

Coding for Recursive Predictive Parser

A→aBb|bAC|Da

```
boolean  A(){
     if (lookahead == ‘a’)
          return (match(‘a’) && B() && match(‘b’));
     else if (lookahead == ‘b’)
          return (match(‘b’) && A() && C());
     else if (lookahead in FIRST(D))
          return (D() && match(a));
     else return false;
}
```

Coding for Recursive Predictive Parser

A→aBb|bAC|ε

```
boolean  A(){
     if (lookahead == 'a')
          return (match('a') && B() && match('b'));
     else if (lookahead == 'b')
          return (match('b') && A() && C());
     else if (lookahead in FOLLOW (A))
          return true;
     else return false;
}
```

Model of a Table-Driven Predictive Parser

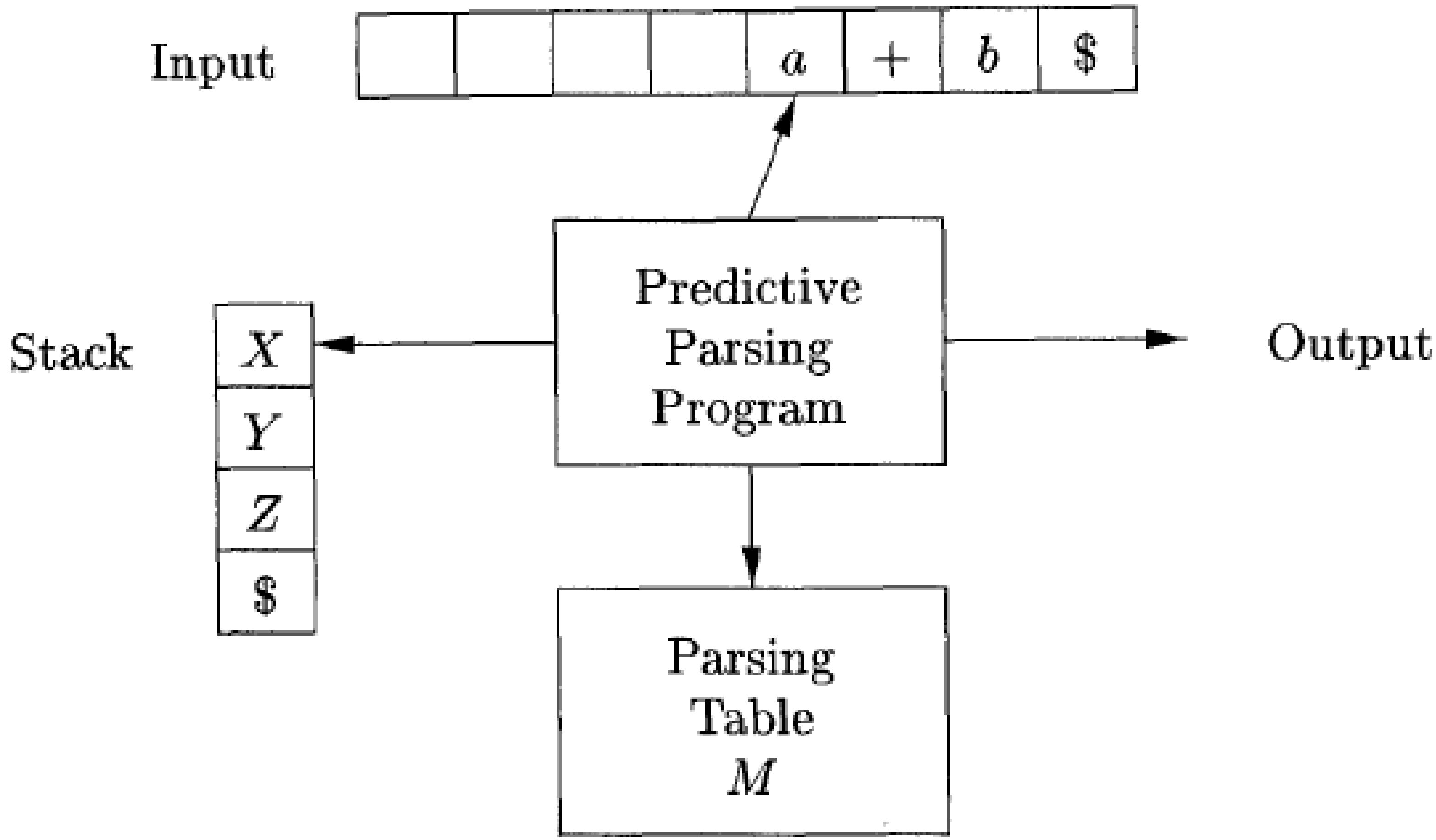


Table-Driven Predictive Parsing

Algorithm 4.34: Table-driven predictive parsing.

INPUT: A string w and a parsing table M for grammar G.

OUTPUT: If w is in $L(G)$, a leftmost derivation of w; otherwise, an error indication.

Table-Driven Predictive Parsing

METHOD: Initially, the parser is in a configuration with $w\$$ in the input buffer and the start symbol S of G on top of the stack, above $\$$. The program in Fig. 4.20 uses the predictive parsing table M to produce a predictive parse for the input. □

the current input symbol

set *ip* to point to the first symbol of w;
set X to the top stack symbol;
while ($X \neq \$$) { /* stack is not empty */
  if (X is a) pop the stack and advance *ip*;
  else if (X is a terminal) *error*();
  else if ($M[X, a]$ is an error entry) *error*();
  else if ($M[X, a] = X \rightarrow Y_1 Y_2 \cdots Y_k$) {
    output the production $X \rightarrow Y_1 Y_2 \cdots Y_k$;
    pop the stack;
    push $Y_k, Y_{k-1}, \ldots, Y_1$ onto the stack, with Y_1 on top;
  }
  set X to the top stack symbol;
} if (a != '$') error();

Example

MATCHED	STACK	INPUT	ACTION
	$E\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	
	$TE'\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	output $E \to TE'$
	$FT'E'\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	output $T \to FT'$
	$\mathbf{id}\ T'E'\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	output $F \to \mathbf{id}$
$\mathbf{id}$	$T'E'\$$	$+ \mathbf{id} * \mathbf{id}\$$	match $\mathbf{id}$
$\mathbf{id}$	$E'\$$	$+ \mathbf{id} * \mathbf{id}\$$	output $T' \to \epsilon$
$\mathbf{id}$	$+\ TE'\$$	$+ \mathbf{id} * \mathbf{id}\$$	output $E' \to +\ TE'$
$\mathbf{id}\ +$	$TE'\$$	$\mathbf{id} * \mathbf{id}\$$	match $+$
$\mathbf{id}\ +$	$FT'E'\$$	$\mathbf{id} * \mathbf{id}\$$	output $T \to FT'$
$\mathbf{id}\ +$	$\mathbf{id}\ T'E'\$$	$\mathbf{id} * \mathbf{id}\$$	output $F \to \mathbf{id}$
$\mathbf{id} + \mathbf{id}$	$T'E'\$$	$* \mathbf{id}\$$	match $\mathbf{id}$
$\mathbf{id} + \mathbf{id}$	$*\ FT'E'\$$	$* \mathbf{id}\$$	output $T' \to *\ FT'$
$\mathbf{id} + \mathbf{id}\ *$	$FT'E'\$$	$\mathbf{id}\$$	match $*$
$\mathbf{id} + \mathbf{id}\ *$	$\mathbf{id}\ T'E'\$$	$\mathbf{id}\$$	output $F \to \mathbf{id}$
$\mathbf{id} + \mathbf{id} * \mathbf{id}$	$T'E'\$$	$\$$	match $\mathbf{id}$
$\mathbf{id} + \mathbf{id} * \mathbf{id}$	$E'\$$	$\$$	output $T' \to \epsilon$
$\mathbf{id} + \mathbf{id} * \mathbf{id}$	$\$$	$\$$	output $E' \to \epsilon$