19Z602 COMPILER DESIGN

Unit-3

SYNTAX ANALYSIS

SYNTAX ANALYSIS : Need and Role of the Parser - Context Free Grammars

Top Down Parsing: Recursive Descent Parser - Predictive Parser.

Bottom Up Parsers: Shift Reduce Parser - LR Parser - LR (0) Item - Construction Of SLR Parsing Table - CLR Parser - LALR Parser.

Error Handling and Recovery in Syntax Analyzer

YACC Tool: Structure of YACC Program – Communication between LEX and YACC - Design of a Syntax Analyzer for a Sample Language

Outline

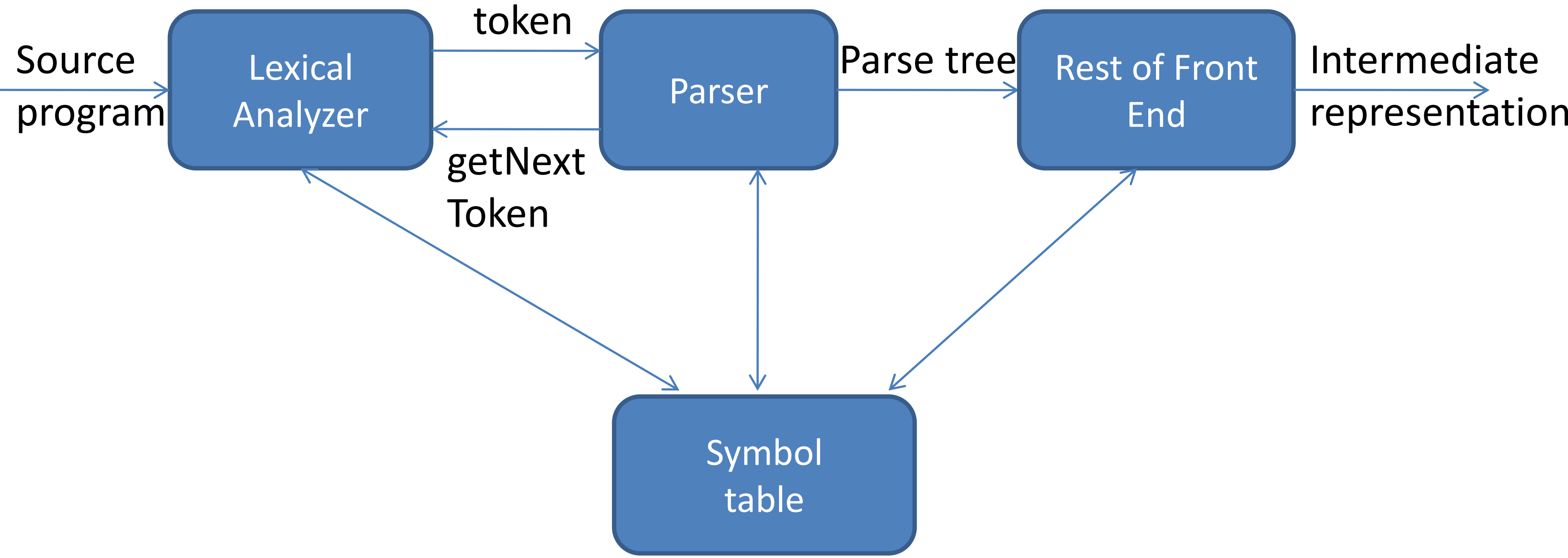
- Role of parser
- Context free grammars
- Top down parsing
- Bottom up parsing
- Parser generators

Need and Role of the parser

Syntactic analysis, or parsing, is needed to analyse the **syntactical** structure and checks if the given input is in the correct **syntax** of the programming language or not. The **syntax** analyser also checks whether a given program fulfills the rules implied by a context-free **grammar**. If it satisfies, the **parser** then creates the **parse** tree of that source program.

In the **syntax analysis** phase, a compiler verifies whether or not the tokens generated by the **lexical analyzer** are grouped according to the **syntactic** rules of the language. It detects and reports any **syntax** errors and produces a **parse** tree from which intermediate code can be generated.

The role of parser



Context Free Grammars

- A context free grammar consists of terminals, nonterminals, a start symbol, and productions.
- Terminals are the basic symbols from which strings are formed.
- Nonterminals are syntactic variables that denote sets of strings.
- One nonterminal is distinguished as the start symbol.
- The productions of a grammar specify the manner in which the terminal and nonterminals can be combined to form strings.
- A language that can be generated by a grammar is said to be a context-free language.

Notational Conventions

- Example

  $E \rightarrow EAE \mid (E) \mid -E \mid id$

  $A \rightarrow + \mid - \mid * \mid / \mid \uparrow$

Derivations

- Productions are treated as rewriting rules to generate a string
- Rightmost and leftmost derivations

  E -> E + E | E * E | -E | (E) | **id**

  Derivations for **–(id+id)**

  - E => -E => -(E) => -(E+E) => -(**id**+E)=>-(**id**+**id**)

Derivations

- $E \Rightarrow -E$ is read "*E derives -E*"
- $E \Rightarrow -E \Rightarrow -(E) = -(\mathbf{id})$ is called a derivation of $-(\mathbf{id})$ from E.
- If $A \rightarrow \gamma$ is a production and α and β are arbitrary strings of grammar symbols, we say $\alpha A \beta \Rightarrow \alpha \gamma \beta$.
- If $\alpha_1 \Rightarrow \alpha_2 \Rightarrow \ldots \Rightarrow \alpha_n$, we say α_1 *derives* α_n.

Derivations (II)

- $\Rightarrow$ means "derives in one step."
- $\overset{*}{\Rightarrow}$ means "derives in zero or more steps."
  - $\alpha \Rightarrow \alpha$
  - if $\alpha \overset{*}{\Rightarrow} \beta$ and $\beta \Rightarrow \gamma$ then $\alpha \overset{*}{\Rightarrow} \gamma$
- $\overset{+}{\Rightarrow}$ means "derives in one or more steps."
- If $S \overset{*}{\Rightarrow} \alpha$, where α may contain nonterminals, then we say that α is a sentential form.

Derivations (III)

- G: grammar, S: start symbol, L(G): the language generated by G.
- Strings in L(G) may contain only terminal symbols of G.
- A string of terminal *w* is said to be in L(G) if and only if $S \overset{+}{\Rightarrow} w$.
- The string *w* is called a sentence of G.
- A language that can be generated by a grammar is said to be a context-free language.
- If two grammars generate the same language, the grammars are said to be equivalent.

Derivations (IV)

$E \rightarrow EAE \mid (E) \mid -E \mid id$

$A \rightarrow + \mid - \mid * \mid / \mid \uparrow$

- The string $-(id+id)$ is a sentence of the above grammar because

$E \Rightarrow -E \Rightarrow -(E+E) \Rightarrow -(\mathbf{id}+E) \Rightarrow -(\mathbf{id}+\mathbf{id})$

We write $E \overset{*}{\Rightarrow} -(\mathbf{id}+\mathbf{id})$

Left-most Derivation

- If the sentential form of an input is scanned and replaced from left to right, it is called left-most derivation. The sentential form derived by the left-most derivation is called the left-sentential form.

Right-most Derivation

- If we scan and replace the input with production rules, from right to left, it is known as right-most derivation. The sentential form derived from the right-most derivation is called the right-sentential form.

Production rules:

E → E + E

E → E * E

E → id

Input string: id + id * id

The left-most derivation is:

E → E * E

E → E + E * E

E → id + E * E

E → id + id * E

E → id + id * id

Notice that the left-most side non-terminal is always processed first.

The right-most derivation is:

E → E + E

E → E + E * E

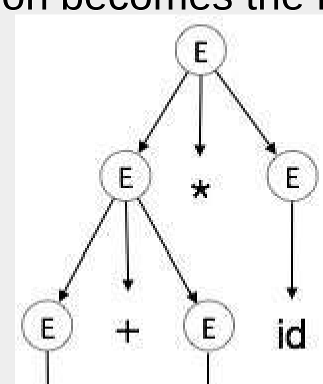
E → E + E * id

E → E + id * id

E → id + id * id

Parse Tree

A parse tree is a graphical depiction of a derivation. It is convenient to see how strings are derived from the start symbol. The start symbol of the derivation becomes the root of the parse tree.



left-most derivation of a + b * c

The left-most derivation is:

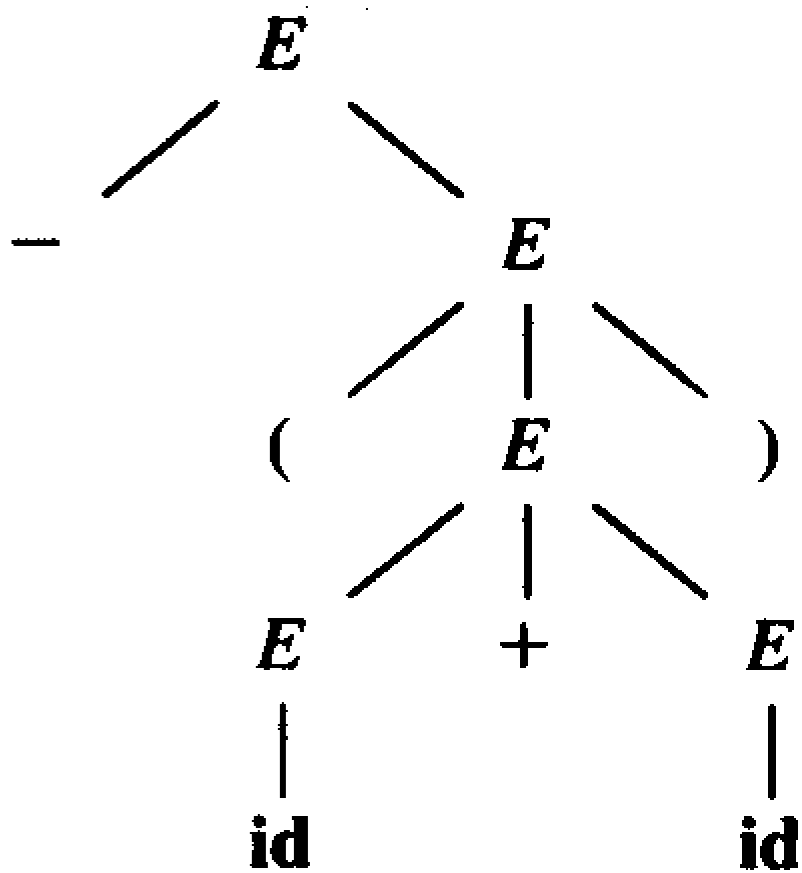
E → E * E

E → E + E * E

E → id + E * E

E → id + id * E

Parse Tree



E→E+E|E*E|(E)|-E|id

Fig. 4.2. Parse tree for −(**id** + **id**).

Parse Tree (II)

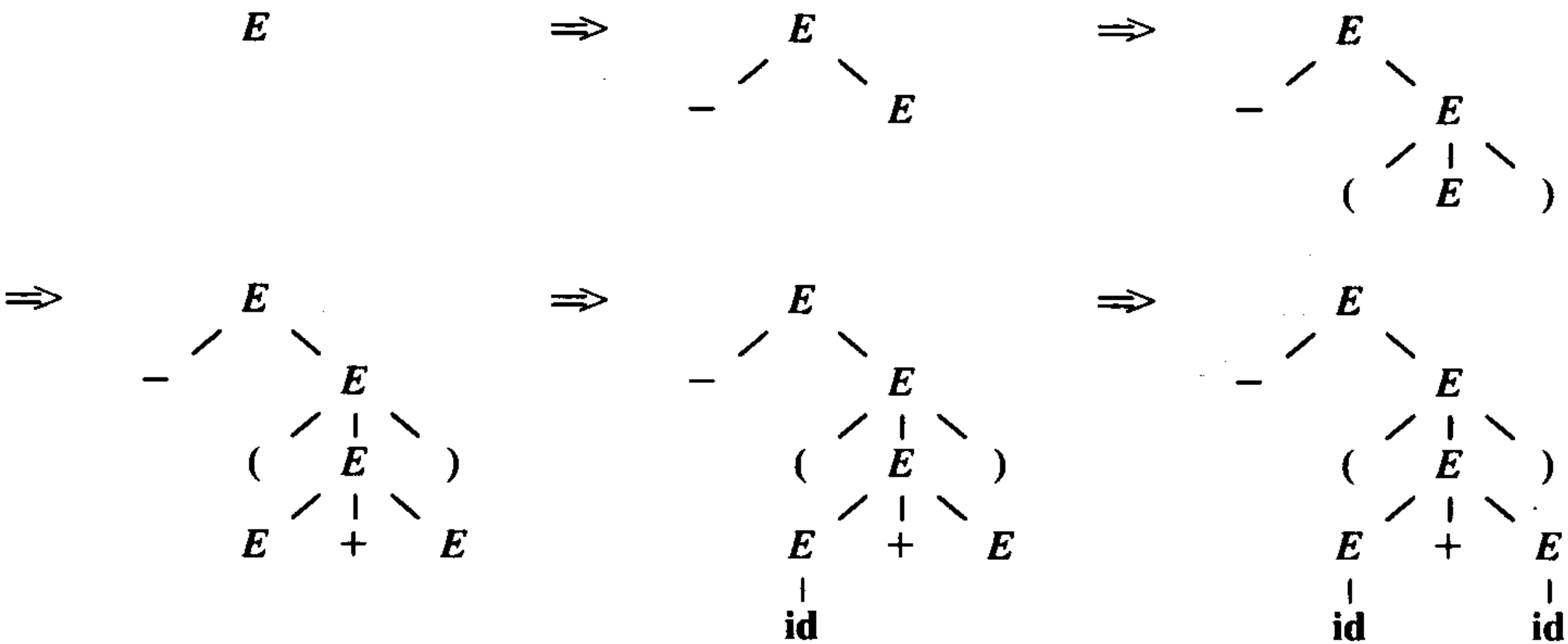


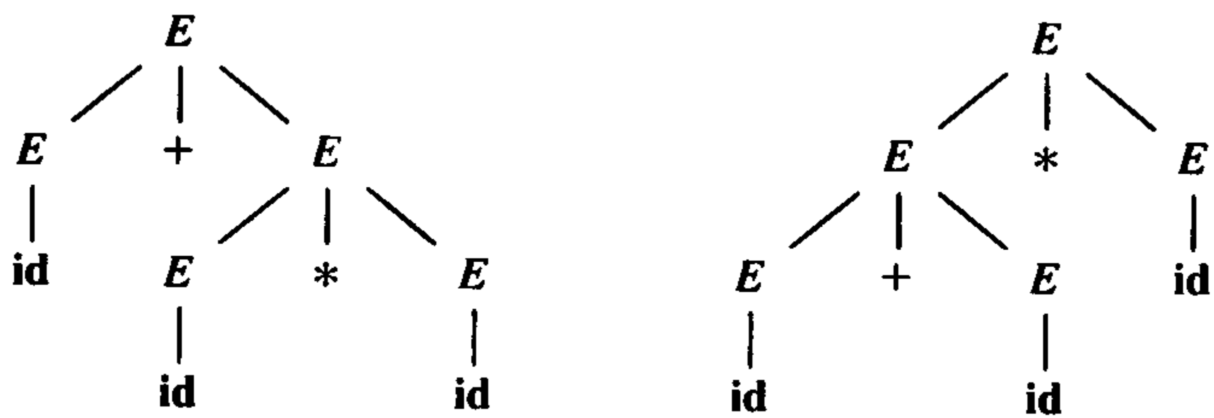
Fig. 4.3. Building the parse tree from derivation (4.4).

Two Parse Trees

Example 4.6. Let us again consider the arithmetic expression grammar (4.3). The sentence **id+id*id** has the two distinct leftmost derivations:

$$\begin{aligned} E &\Rightarrow E+E \\ &\Rightarrow \mathbf{id}+E \\ &\Rightarrow \mathbf{id}+E*E \\ &\Rightarrow \mathbf{id}+\mathbf{id}*E \\ &\Rightarrow \mathbf{id}+\mathbf{id}*\mathbf{id} \end{aligned} \qquad\qquad \begin{aligned} E &\Rightarrow E*E \\ &\Rightarrow E+E*E \\ &\Rightarrow \mathbf{id}+E*E \\ &\Rightarrow \mathbf{id}+\mathbf{id}*E \\ &\Rightarrow \mathbf{id}+\mathbf{id}*\mathbf{id} \end{aligned}$$

with the two corresponding parse trees shown in Fig. 4.4. □



(a) (b)

Fig. 4.4. Two parse trees for **id+id*id**.

If E1 then S1 else if E2 then S2 else S3
If E1 then if E2 then S1 else S2

$$stmt \rightarrow \textbf{if } expr \textbf{ then } stmt \mid \textbf{if } expr \textbf{ then } stmt \textbf{ else } stmt \mid \textbf{other}$$

Ambiguity

- A grammar that produces more than one parse tree for some sentence is said to be *ambiguous*.

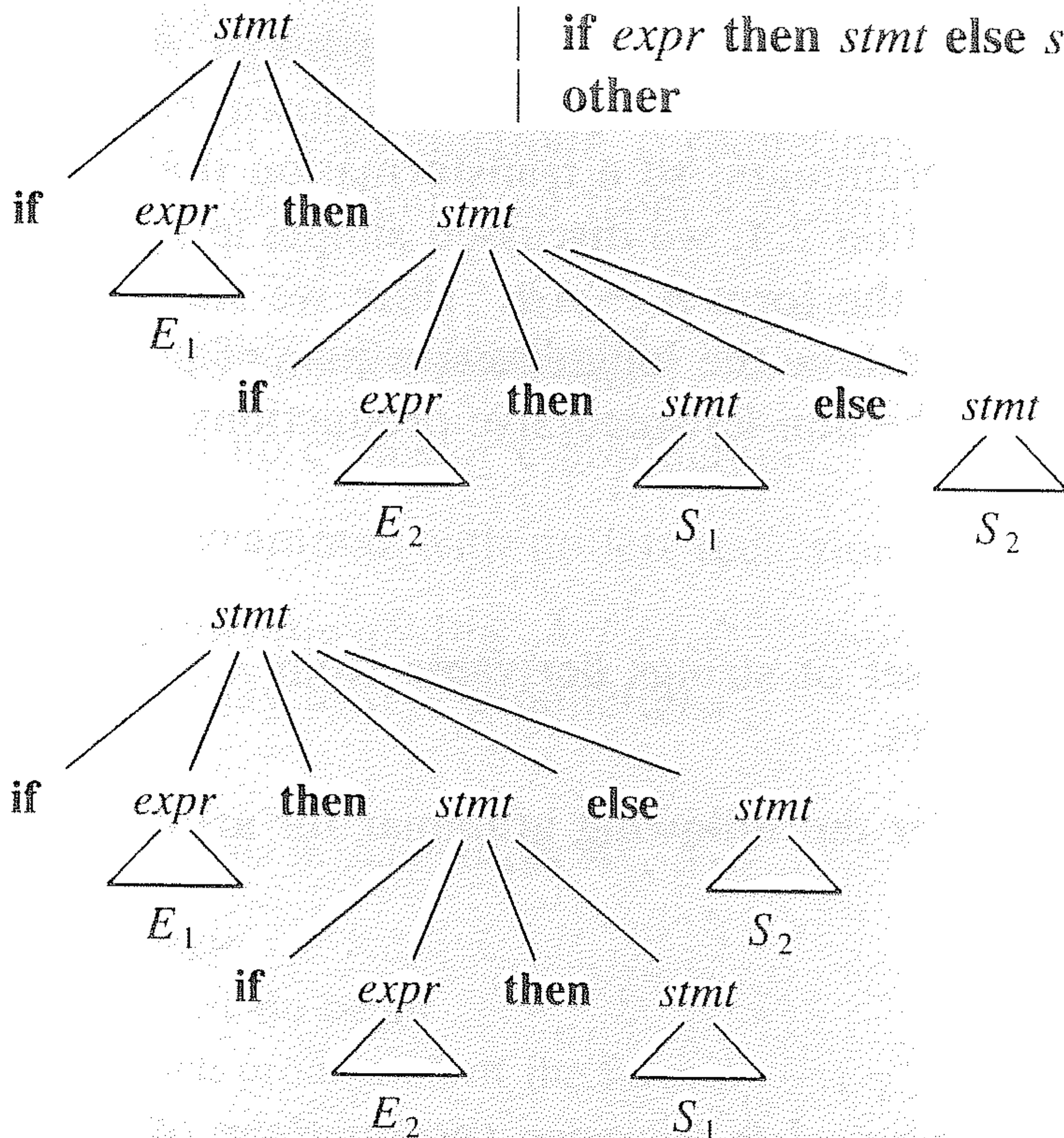


Fig. 4.6. Two parse trees for an ambiguous sentence.

Eliminating Ambiguity

- Sometimes an ambiguous grammar can be rewritten to eliminate the ambiguity.
- Stmt appearing between a 'then' and 'else' must be 'matched' ie - interior stmt must not end with unmatched or open 'then'

$$
\begin{array}{rcl}
\mathit{stmt} & \rightarrow & \mathit{matched\_stmt} \\
& | & \mathit{unmatched\_stmt} \\
\mathit{matched\_stmt} & \rightarrow & \textbf{if}\ \mathit{expr}\ \textbf{then}\ \mathit{matched\_stmt}\ \textbf{else}\ \mathit{matched\_stmt} \\
& | & \textbf{other} \\
\mathit{unmatched\_stmt} & \rightarrow & \textbf{if}\ \mathit{expr}\ \textbf{then}\ \mathit{stmt} \\
& | & \textbf{if}\ \mathit{expr}\ \textbf{then}\ \mathit{matched\_stmt}\ \textbf{else}\ \mathit{unmatched\_stmt}
\end{array}
$$

Eliminating 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.
- If we have the left-recursive pair of productions-
- $A \rightarrow A\alpha \mid \beta$

(Left Recursive Grammar)

where β does not begin with an A.

$A \rightarrow A\alpha \mid \beta$ can be replaced by

$A \rightarrow \beta A'$

$A' \rightarrow \alpha A' \mid \varepsilon$

- $A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \dots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \dots \mid \beta_n \mid$

$A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \dots \mid \beta_n A' \mid$

$A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \dots \ \alpha_m A' \mid \varepsilon$

Algorithm: Eliminating Left Recursion

Input. Grammar G with no cycles or ϵ-productions.

Output. An equivalent grammar with no left recursion.

Method. Apply the algorithm ' to G. Note that the resulting non-left-recursive grammar may have ϵ-productions. □

1. Arrange the nonterminals in some order $A_1, A_2, \ldots, A_n$.

2. **for** i := 1 **to** n **do begin**
    for j := 1 **to** $i-1$ **do begin**
        replace each production of the form $A_i \rightarrow A_j\gamma$
            by the productions $A_i \rightarrow \delta_1\gamma \mid \delta_2\gamma \mid \cdots \mid \delta_k\gamma$,
            where $A_j \rightarrow \delta_1 \mid \delta_2 \mid \cdots \mid \delta_k$ are all the current A_j-productions;
    end
    eliminate the immediate left recursion among the A_i-productions
end

Examples

$$E \rightarrow E + T \mid T$$
$$T \rightarrow T * F \mid F$$
$$F \rightarrow (E) \mid \mathbf{id}$$

$$E \rightarrow TE'$$
$$E' \rightarrow +TE' \mid \epsilon$$
$$T \rightarrow FT'$$
$$T' \rightarrow *FT' \mid \epsilon$$
$$F \rightarrow (E) \mid \mathbf{id}$$

$S \rightarrow Aa \mid b$

$A \rightarrow Ac \mid Sd \mid \varepsilon$

$A \rightarrow Ac \mid Aad \mid bd \mid \varepsilon$

$S \rightarrow Aa \mid b$

$A \rightarrow bdA' \mid A'$

$A' \rightarrow cA' \mid adA' \mid \varepsilon$

Examples

- $A \rightarrow ABd \ / \ Aa \ / \ a$

  $B \rightarrow Be \ / \ b$

- $S \rightarrow (L) \ / \ a$

  $L \rightarrow L \ , \ S \ / \ S$

Examples

- A → ABd / Aa / a

  B → Be / b

- A → aA'

  A' → BdA' / aA' / ∈

  B → bB'

  B' → eB' / ∈

- S → (L) / a

  L → L , S / S

- S → (L) / a

  L → SL'

  L' → ,SL' / ∈

Left Factoring

- Left factoring is a grammar transformation that is useful for producing a grammar suitable for **predictive parsing.**
- The basic idea is that when it is not clear which of two alternative productions to use to expand a nonterminal A, be able to rewrite the A-productions to defer the decision until we have seen enough of the input to make the right choice.
- `Stmt --> ` **`if`** ` expr ` **`then`** ` stmt ` **`else`** ` stmt`

  `| ` **`if`** ` expr ` **`then`** ` stmt`

Algorithm: Left Factoring

Algorithm 4.2. Left factoring a grammar.

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 the A productions $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \cdots \mid \alpha\beta_n \mid \gamma$ where γ represents all alternatives that do not begin with α by

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

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

Left Factoring (example)

- $A \rightarrow \alpha\beta_1 | \alpha\beta_2$

- The following grammar abstracts the dangling-else problem:
  - $S \rightarrow iEtS | iEtSeS | a$
  - $E \rightarrow b$

  - $S \rightarrow iEtSS' | a$
  - $S' \rightarrow eS | \varepsilon$
  - $E \rightarrow b$

Error handling

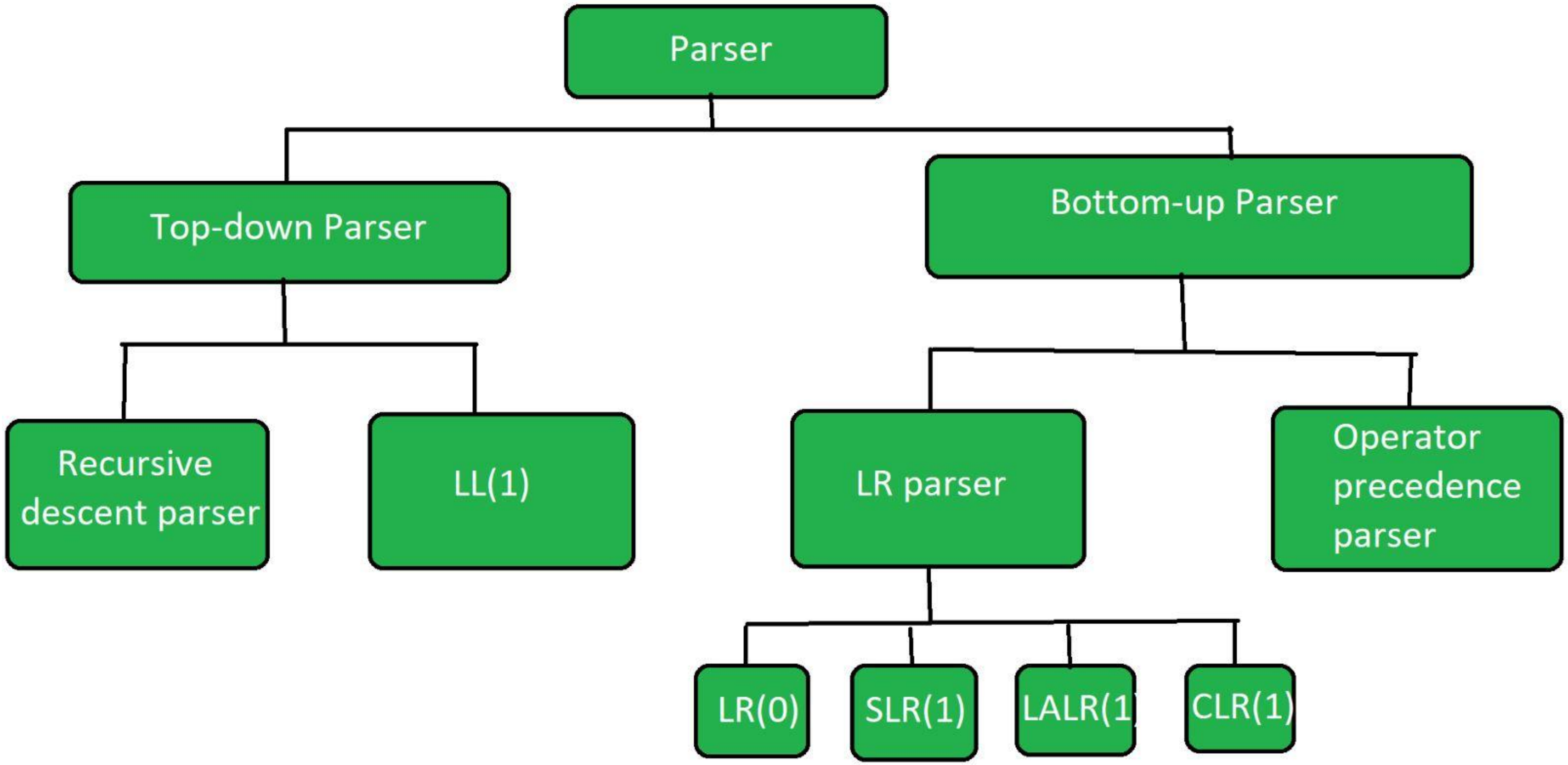
- Common programming errors
  - Lexical errors: misspellings of identifiers, keywords, or operators
  - Syntactic errors: misplaced semicolons, extra or missing braces, case without switch, … .
  - Semantic errors: type mismatches between operators and operands
  - Logical errors
- Error handler goals
  - Report the presence of errors clearly and accurately
  - Recover from each error quickly enough to detect subsequent errors
  - Add minimal overhead to the processing of correct programs

Error-recover strategies

- Panic mode recovery
  - Discard input symbol one at a time until one of designated set of synchronization tokens is found
- Phrase level recovery
  - Replacing a prefix of remaining input by some string that allows the parser to continue
- Error productions
  - Augment the grammar with productions that generate the erroneous constructs (production rules for common errors)
- Global correction
  - Choosing minimal sequence of changes to obtain a globally least-cost correction

TOP DOWN PARSING

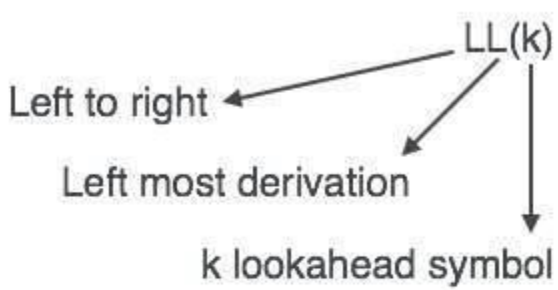
Types of Parser

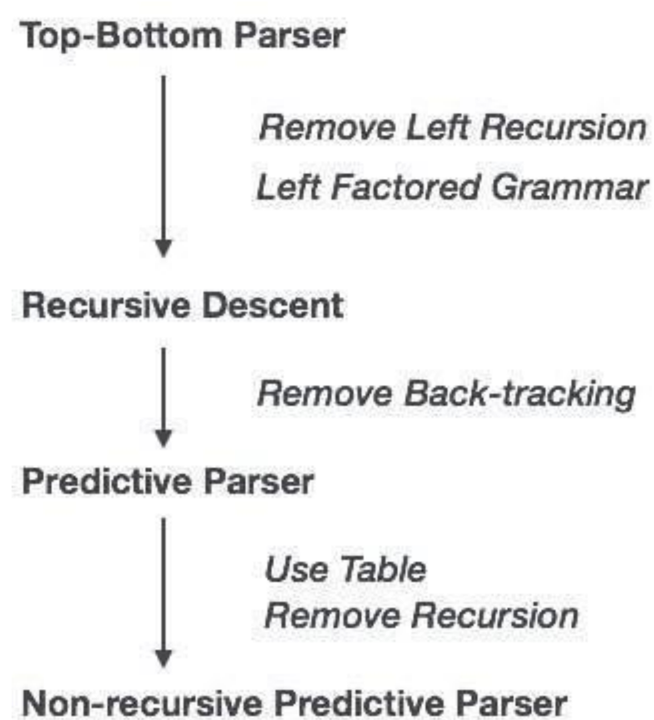


Top-down parser

Top-down parser is the parser which generates parse for the given input string with the help of grammar productions by expanding the non-terminals i.e. it starts from the start symbol and ends on the terminals. It uses left most derivation.

Top-down parser is a parser for LL class of grammars





Recursive Descent Parsing

- Recursive descent parsing is a top-down method of syntax analysis in which a set recursive procedures to process the input is executed.
- A procedure is associated with each nonterminal of a grammar.
- Top-down parsing can be viewed as an attempt to find a leftmost derivation for an input string.
- Equivalently, it attempts to construct a parse tree for the input starting from the root and creating the nodes of the parse tree in preorder.
- Recursive descent parsing involves backtracking.

Recursive descent parsing

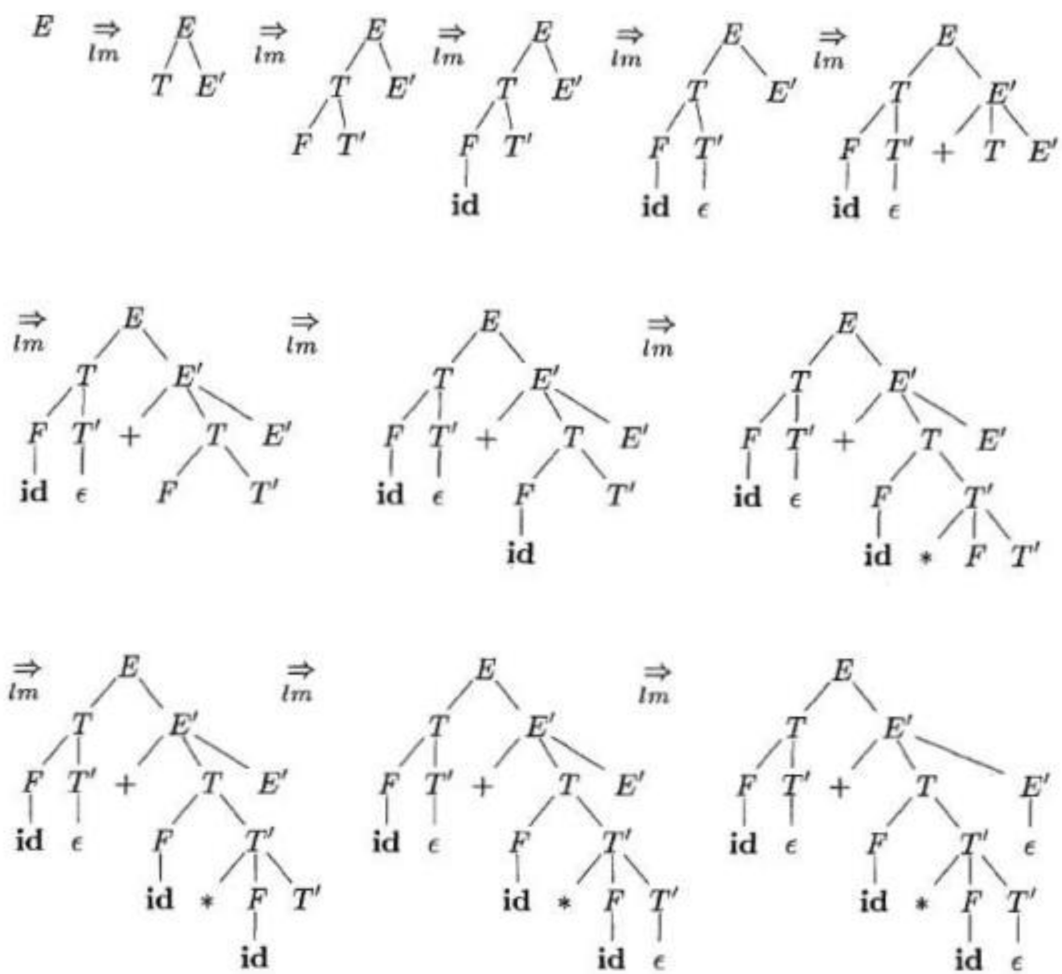
- Consists of a set of procedures, one for each nonterminal
- Execution begins with the procedure for start symbol
- A typical procedure for a non-terminal

```
void A() {
        choose an A-production, A->X1X2..Xk
        for (i=1 to k) {
                if (Xi is a nonterminal
                        call procedure Xi();
                else if (Xi equals the current input symbol a)
                        advance the input to the next symbol;
                else /* an error has occurred */
        }
}
```

Given:

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

and: $\mathbf{id}+\mathbf{id}*\mathbf{id}$



```
procedure E()
        T();
        E'();
    if NextInputChar = END  then /* done */
   else print("syntax error")

procedureE'();
    if NextInputChar = "+"  then
     read(NextInputChar);
     T() ;
     E'();

 procedure T()
        F();
        T'();

 procedure T '()
        if NextInputChar = "*" then
        read(NextInputChar);
        F();
        T'();
 procedure F()
        if NextInputChar = "(" then
                     read(NextInputChar);
                     E();
                     if NextInputChar = ")" then
                     read(NextInputChar)
                     else print("syntax error");
        else if NextInputChar = identifier then
        read(NextInputChar)
        else print("syntax error");
```

Consider the Grammar

$E \rightarrow TE'$

$E' \rightarrow +TE' \mid \varepsilon$

$T \rightarrow FT'$

$T' \rightarrow *FT' \mid \varepsilon$

$F \rightarrow (E) \mid id$

Example (backtracking)

- Consider the grammar

  $S \rightarrow cAd$

  $A \rightarrow ab|a$

  and the input string w = cad
- To construct a parse tree for this string using top-down approach, initially create a tree consisting of a single node labeled S.

Procedure S

```
procedure S()
begin
    if input symbol = 'c' then
        begin
            ADVANCE( );
             if A( ) then
                if input symbol = 'd' then
                    begin ADSVANCE( ); return true
end
        end;
    return false
end
```

Procedure A

```
procedure A( )
begin
    isave := input-pointer;
    if input symbol = 'a' then
        begin
            ADVANCE( );
            if input symbol = 'b' then
                begin ADVANCE( ); return true end
        end
    input-pointer := isave;
        /* failure to find ab */
    if input symbol = 'a' then
        begin ADVANCE( ); return true end
    else return false
end
```

Grammar:

$$S \rightarrow cAd$$
$$A \rightarrow ab \mid a$$

Input string

`w = cad`

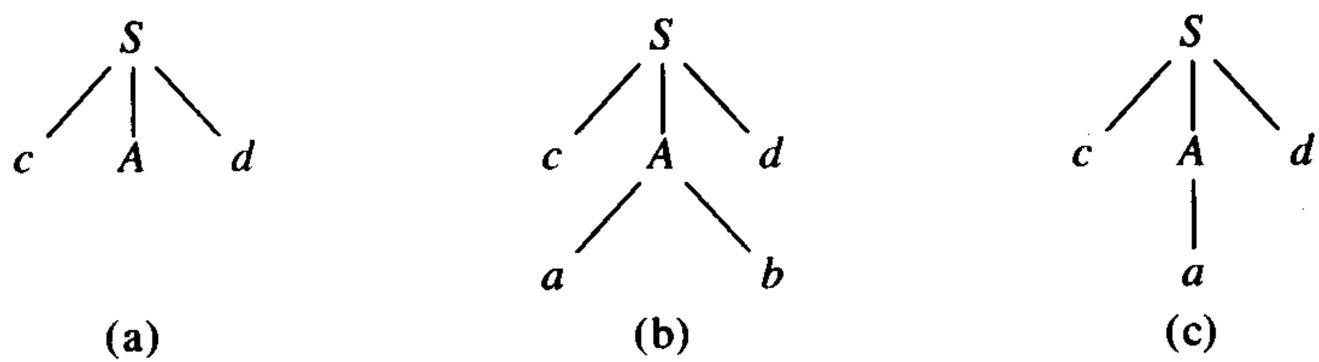


Fig. 4.9. Steps in top-down parse.

Predictive Parsers

- Eliminating left recursion, and left factoring the resulting grammar, can obtain a grammar that can be parsed by a recursive-descent parser that needs **no backtracking**, i.e., a predictive parser.

$$
\begin{aligned}
S &\rightarrow cAd \\
A &\rightarrow aA' \\
A' &\rightarrow b \mid \varepsilon
\end{aligned}
$$

$$
\begin{aligned}
\textit{stmt} \rightarrow\ & \textbf{if}\ \textit{expr}\ \textbf{then}\ \textit{stmt}\ \textbf{else}\ \textit{stmt} \\
\mid\ & \textbf{while}\ \textit{expr}\ \textbf{do}\ \textit{stmt} \\
\mid\ & \textbf{begin}\ \textit{stmt\_list}\ \textbf{end}
\end{aligned}
$$

The construction of a predictive parser is aided by two functions associated with a grammar G :

- FIRST()-First(α) is set of terminals that begins strings derived from α.

  In predictive parsing when we have A-> α|β, if First(α) and First(β) are disjoint sets then we can select appropriate A-production by looking at the next input

- FOLLOW()- for any nonterminal A, is set of terminals **a** that can appear immediately after A in some sentential form

  If we have $S \overset{*}{\Rightarrow} \alpha A a \beta$ for some α and β then **a** is in Follow(A)

Rules for FIRST( ):

- If X is terminal, then FIRST(X) is {X}.
- If $X \rightarrow \varepsilon$ is a production, then add ε to FIRST(X).
- If X is non-terminal and $X \rightarrow a\alpha$ is a production then add a to FIRST(X).
- If X is non-terminal and $X \rightarrow Y_1 Y_2 \ldots Y_k$ is a production, then place ***a*** in **FIRST(*X*)** if for some i, a is in FIRST(Yi), and ε is in all of FIRST($Y1$),…,FIRST($Yi\text{-}1$); that is, $Y1, \ldots . Yi\text{-}1 \overset{*}{=>} \varepsilon$. If ε is in FIRST($Y_j$) for all j=1,2,..,k, then add ε to FIRST(X).

Rules for FOLLOW ( ):

- If S is a start symbol, then FOLLOW(S) contains \$.
- If there is a production $A \rightarrow \alpha B\beta$, then everything in FIRST(β) except ε is placed in follow(B).
- If there is a production $A \rightarrow \alpha B$, or a production $A \rightarrow \alpha B\beta$ where FIRST(β) contains ε, then everything in FOLLOW(A) is in FOLLOW(B).

Consider the following grammar :

E → E+T | T

T→T*F | F F → (E) | id

After eliminating left-recursion the grammar is

E → TE'

E' → +TE' |ε

T → FT'

T' → *FT' | ε

F → (E) | id

FIRST( ) :

FIRST(E) = { (, id}

FIRST(E') ={+ ,ε}

FIRST(T) = { (, id}

FIRST(T') = {*, ε }

FIRST(F) = { (, id }

FOLLOW( ):

FOLLOW(E) = { $,) }

FOLLOW(E') = { $,) }

FOLLOW(T) = { +, $,) }

FOLLOW(T') = { +, $,) }

FOLLOW(F) = {+, * , $,) }

Consider this following grammar:

S → iEtS | iEtSeS | a

E → b

After eliminating left factoring,

S→iEtSS' | a

S'→eS |ε

E→b

FIRST(S) = { i, a }

FIRST(S') = {e,ε}

FIRST(E) = { b}

FOLLOW(S) = { $,e }

FOLLOW(S') = { $,e }

FOLLOW(E) = {t}

FIRST(X)

For a production rule $X \rightarrow Y_1Y_2Y_3$,

- If $\in \notin First(Y_1)$, then $First(X) = First(Y_1)$
- If $\in \in First(Y_1)$, then $First(X) = \{ First(Y_1) - \in \} \cup First(Y_2Y_3)$

- If $\in \notin First(Y_2)$, then $First(Y_2Y_3) = First(Y_2)$
- If $\in \in First(Y_2)$, then $First(Y_2Y_3) = \{ First(Y_2) - \in \} \cup First(Y_3)$

FOLLOW(B)

For any production rule $A \rightarrow \alpha B \beta$,

- If $\in \notin$ First(β), then Follow(B) = First(β)
- If $\in \in$ First(β), then Follow(B) = { First(β) – $\in$ } $\cup$ Follow(A)

Calculate the first and follow functions for the given grammar-

$S \rightarrow aBDh$

$B \rightarrow cC$

$C \rightarrow bC$ / $\in$

$D \rightarrow EF$

$E \rightarrow g$ / $\in$

$F \rightarrow f$ / $\in$

- $First(S) = \{ a \}$
- $First(B) = \{ c \}$
- $First(C) = \{ b , \in \}$
- $First(D) = \{ First(E) – \in \} \cup First(F) = \{ g , f , \in \}$
- $First(E) = \{ g , \in \}$
- $First(F) = \{ f , \in \}$

$S \rightarrow aBDh$

$B \rightarrow cC$

$C \rightarrow bC / \in$

$D \rightarrow EF$

$E \rightarrow g / \in$

$F \rightarrow f / \in$

- $Follow(S) = \{ \$ \}$
- $Follow(B) = \{ First(D) – \in \} \cup First(h) = \{ g , f , h \}$
- $Follow(C) = Follow(B) = \{ g , f , h \}$
- $Follow(D) = First(h) = \{ h \}$
- $Follow(E) = \{ First(F) – \in \} \cup Follow(D) = \{ f , h \}$
- $Follow(F) = Follow(D) = \{ h \}$

$S \rightarrow A$

$A \rightarrow aB \mid Ad$

$B \rightarrow b$

$C \rightarrow g$

$S \rightarrow A$

$A \rightarrow aBA'$

$A' \rightarrow dA' \mid \epsilon$

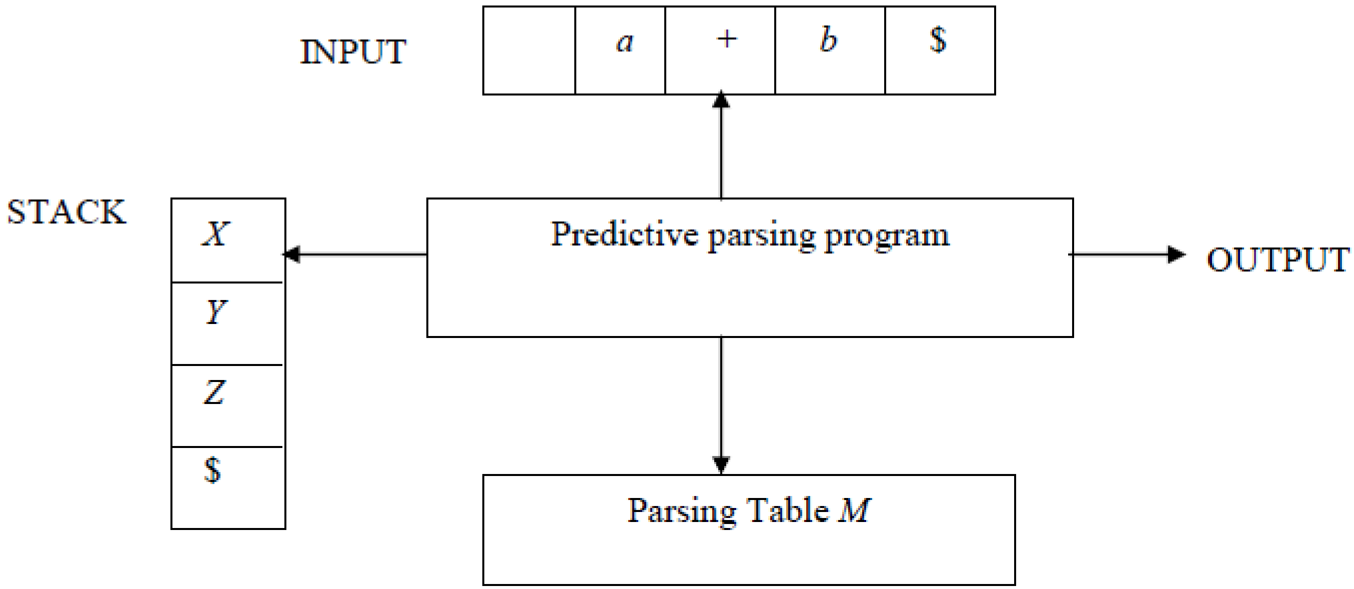
$B \rightarrow b$

$C \rightarrow g$

LL(1) Grammars

- Predictive parsers are those recursive descent parsers needing no backtracking
- Grammars for which we can create predictive parsers are called LL(1)
  - The first L means scanning input from left to right
  - The second L means leftmost derivation
  - And 1 stands for using one input symbol for lookahead
- A grammar G is LL(1) if and only if whenever A-> $\alpha \mid \beta$ are two distinct productions of G, the following conditions hold:
  - For no terminal a do α and β both derive strings beginning with a
  - At most one of α or β can derive empty string
  - If $\alpha \overset{*}{=>} \varepsilon$ then β does not derive any string beginning with a terminal in Follow(A).

Non-recursive predictive parser



The table-driven predictive parser has an input buffer, stack, a parsing table and an output stream.

Input buffer:

It consists of strings to be parsed, followed by $ to indicate the end of the input string.

Stack:

It contains a sequence of grammar symbols preceded by $ to indicate the bottom of the stack. Initially, the stack contains the start symbol on top of $.

Parsing table:

It is a two-dimensional array $M[A, a]$, where ***'A'*** is a non-terminal and ***'a'*** is a terminal.

Construction of predictive parsing table

Algorithm for construction of predictive parsing table:

Input : Grammar *G*

Output : Parsing table *M*

Method :

1. For each production $A \rightarrow \alpha$ of the grammar, do steps 2 and 3.
2. For each terminal a in FIRST(α), add $A \rightarrow \alpha$ to $M[A, a]$.
3. If ε is in FIRST(α), add $A \rightarrow \alpha$ to $M[A, b]$ for each terminal b in FOLLOW(A). If ε is in FIRST(α) and $ is in FOLLOW(A) , add $A \rightarrow \alpha$ to $M[A, \$]$.
4. Make each undefined entry of *M* be **error**.

E → TE'
E' → +TE' |ε
T → FT'
T' → *FT' | ε
F → (E) | id

FIRST( ) :

FIRST(E) = { (, id}
FIRST(E') ={+ ,ε}
FIRST(T) = { (, id}
FIRST(T') = {*, ε }
FIRST(F) = { (, id }

FOLLOW( ):

FOLLOW(E) = { $,) }
FOLLOW(E') = { $,) }
FOLLOW(T) = { +, $,) }
FOLLOW(T') = { +, $,) }
FOLLOW(F) = {+, * , $,) }

Algorithm for construction of predictive parsing table:

Input : Grammar *G*

Output : Parsing table *M*

Method :

1. For each production *A* → α of the grammar, do steps 2 and 3.
2. For each terminal *a* in FIRST(α), add *A* → α to *M*[*A*, *a*].
3. If ε is in FIRST(α), add A → α to *M*[*A*, *b*] for each terminal *b* in FOLLOW(*A*). If ε is in FIRST(α) and $ is in FOLLOW(*A*) , add *A* → α to *M*[*A*, $].
4. Make each undefined entry of *M* be **error**.

Predictive parsing table :

NON-TERMINAL	id	+	*	(	)	$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \varepsilon$	$E' \rightarrow \varepsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \varepsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \varepsilon$	$T' \rightarrow \varepsilon$
F	$F \rightarrow id$			$F \rightarrow (E)$		

Predictive parsing program:

The parser is controlled by a program that considers ***X*, the symbol on top of stack**, and *a*, the current input symbol. These two symbols determine the parser action. There are three possibilities:

- If $X = a = \$$, the parser halts and announces successful completion of parsing.
- If $X = a \neq \$$, the parser pops X off the stack and advances the input pointer to the next input symbol.
- If X is a non-terminal , the program consults entry $M[X, a]$ of the parsing table M. This entry will either be an X-production of the grammar or an error entry. If $M[X, a] = \{X \rightarrow UVW\}$,the parser replaces X on top of the stack by UVW
- If $M[X, a]$ =**error**, the parser calls an error recovery routine.

Algorithm for nonrecursive 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.

Method : Initially, the parser has $*S* on the stack with *S*, the start symbol of *G* on top, and *w*$ in the input buffer. The program that utilizes the predictive parsing table *M* to produce a parse for the input is as follows:

set *ip* to point to the first symbol of *w*$;

repeat

let *X* be the top stack symbol and *a* the symbol pointed to by *ip*;

if *X* is a terminal or $ **then**

if *X* = *a* **then**

pop *X* from the stack and advance *ip*

else *error*()

else/* *X* is a non-terminal */

if *M*[*X*, *a*] = *X* →*Y1Y2* … *Yk* **then begin**

pop *X* from the stack;

push *Yk*, *Yk-1*, … ,*Y1* onto the stack, with *Y1* on top;

output the production *X* → *Y1 Y2* . . . *Yk*

end

else*error*()

until *X* = $

Stack implementation:

stack	Input	Output
$E	id+id*id $	
$E'T	id+id*id $	E → TE'
$E'T'F	id+id*id $	T → FT'
$E'T'id	id+id*id $	F → id
$E'T'	+id*id $	
$E'	+id*id $	T' → ε
$E'T+	+id*id $	E' → +TE'
$E'T	id*id $	
$E'T'F	id*id $	T → FT'
$E'T'id	id*id $	F → id
$E'T'	*id $	
$E'T'F*	*id $	T' → *FT'
$E'T'F	id $	
$E'T'id	id $	F → id
$E'T'	$	
$E'	$	T' → ε
$	$	E' → ε

Stack implementation:

stack	Input	Output
$E	id+id*id $	
$E'T	id+id*id $	E → TE'
$E'T'F	id+id*id $	T → FT'
$E'T'id	id+id*id $	F → id
$E'T'	+id*id $	
$E'	+id*id $	T' → ε
$E'T+	+id*id $	E' → +TE'
$E'T	id*id $	
$E'T'F	id*id $	T → FT'
$E'T'id	id*id $	F → id
$E'T'	*id $	
$E'T'F*	*id $	T' → *FT'
$E'T'F	id $	
$E'T'id	id $	F → id
$E'T'	$	
$E'	$	T' → ε
$	$	E' → ε

Error recovery in predictive parsing

An error is detected during the predictive parsing

- when the terminal on top of the stack does not match the next input symbol **or**
- when nonterminal A on top of the stack, a is the next input symbol, and parsing table entry M[A,a] is empty.

Error recovery in predictive parsing

- **Panic-mode error recovery** is based on the idea of skipping symbols on the input until a token in a selected set of synchronizing tokens. In this method, successive characters from input are removed one at a time until a designated set of synchronizing tokens is found. Synchronizing tokens are deli-meters such as ; or }
- **Phrase Level Recovery -**This involves, defining the blank entries in the table **with pointers to some error routines** which may

❖ Change, delete or insert symbols in the input or

❖ May also pop symbols from the stack

Panic-mode

How to select synchronizing set?

- Place all symbols in FOLLOW(A) into the synchronizing set for nonterminal A. If we skip tokens until an element of FOLLOW(A) is seen and pop A from the stack, it likely that parsing can continue.
- We might add keywords that begins statements to the synchronizing sets for the nonterminals generating expressions.
- If a nonterminal can generate the empty string, then the production deriving ε can be used as a default. This may postpone some error detection, but cannot cause an error to be missed. This approach reduces the number of nonterminals that have to be considered during error recovery.
- If a terminal on top of stack cannot be matched, a simple idea is to pop the terminal, issue a message saying that the terminal was inserted.

Example: error recovery

"synch" indicating synchronizing tokens obtained from FOLLOW set of the nonterminal in question.

If the parser looks up entry M[A,a] and finds that it is blank, the input symbol a is skipped.

If the entry is synch, the nonterminal on top of the stack is popped.

If a token on top of the stack does not match the input symbol, then we pop the token from the stack.

FIRST(E) = FIRST(T) = FIRST(F) = {(, **id**}.

FIRST(E') = {+, ϵ}

FIRST(T') = {*, ϵ}

FOLLOW(E) = FOLLOW(E') = {), \$}

FOLLOW(T) = FOLLOW(T') = {+,), \$}

FOLLOW(F) = {+, *,), \$}

NONTERMINAL	INPUT SYMBOL					
	id	+	*	(	)	\$
E	$E \to TE'$			$E \to TE'$	synch	synch
E'		$E' \to +TE'$			$E' \to \epsilon$	$E' \to \epsilon$
T	$T \to FT'$	synch		$T \to FT'$	synch	synch
T'		$T' \to \epsilon$	$T' \to *FT'$		$T' \to \epsilon$	$T' \to \epsilon$
F	$F \to \mathbf{id}$	synch	synch	$F \to (E)$	synch	synch

Fig. 4.18. Synchronizing tokens added to parsing table of Fig. 4.15.

Example: error recovery (II)

STACK	INPUT	REMARK
$\$E$	$)\,\mathbf{id} * + \mathbf{id}\,\$$	error, skip)
$\$E$	$\mathbf{id} * + \mathbf{id}\,\$$	**id** is in FIRST(E)
$\$E'T$	$\mathbf{id} * + \mathbf{id}\,\$$	
$\$E'T'F$	$\mathbf{id} * + \mathbf{id}\,\$$	
$\$E'T'\mathbf{id}$	$\mathbf{id} * + \mathbf{id}\,\$$	
$\$E'T'$	$* + \mathbf{id}\,\$$	
$\$E'T'F*$	$* + \mathbf{id}\,\$$	
$\$E'T'F$	$+ \mathbf{id}\,\$$	error, $M[F, +]$ = synch
$\$E'T'$	$+ \mathbf{id}\,\$$	F has been popped
$\$E'$	$+ \mathbf{id}\,\$$	
$\$E'T+$	$+ \mathbf{id}\,\$$	
$\$E'T$	$\mathbf{id}\,\$$	
$\$E'T'F$	$\mathbf{id}\,\$$	
$\$E'T'\mathbf{id}$	$\mathbf{id}\,\$$	
$\$E'T'$	$\$$	
$\$E'$	$\$$	
$\$$	$\$$	

Fig. 4.19. Parsing and error recovery moves made by predictive parser.