Syntactic Analysis
(Top Down Parsing)

Dr. P K Singh

Top-Down Parsing

- The parse tree is created top to bottom.
- Top-down parser
  - Recursive-Descent Parsing
    - Backtracking is needed (If a choice of a production rule does not work, we backtrack to try other alternatives.)
    - It is a general parsing technique, but not widely used.
    - Not efficient
  - Predictive Parsing
    - no backtracking
    - efficient
    - needs a special form of grammars (LL(1) grammars).
    - Recursive Predictive Parsing is a special form of Recursive Descent parsing without backtracking.
    - Non-Recursive (Table Driven) Predictive Parser is also known as LL(1) parser.

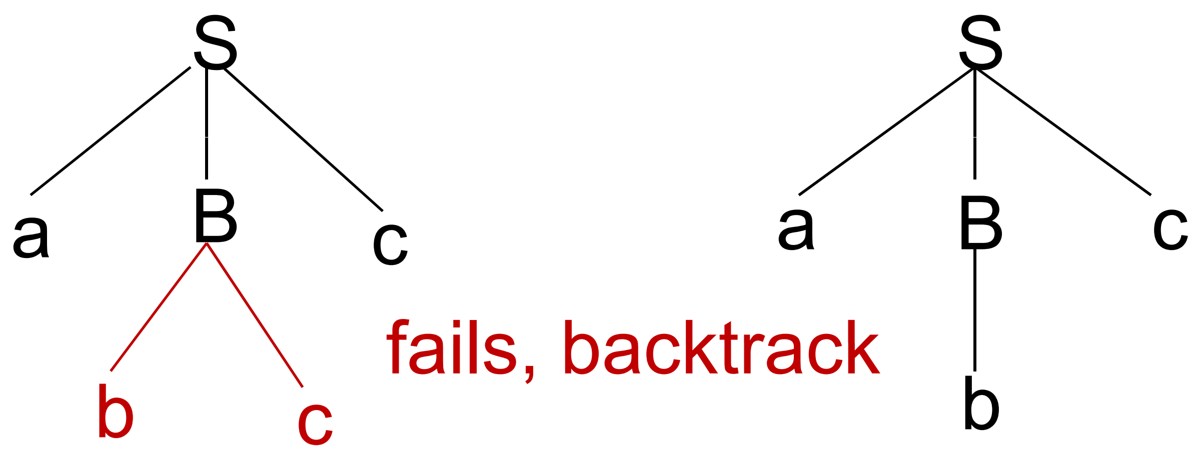
Recursive-Descent Parsing (uses Backtracking)

- Backtracking is needed.
- It tries to find the left-most derivation.

$S \rightarrow aBc$

$B \rightarrow bc \mid b$

Input: abc



Predictive Parser

a grammar ➔ (eliminate left recursion) ➔ (left factor) a grammar suitable for predictive parsing (a LL(1) grammar) no %100 guarantee.

- When re-writing a non-terminal in a derivation step, a predictive parser can uniquely choose a production rule by just looking the current symbol in the input string.

$A \to \alpha_1 \mid ... \mid \alpha_n$ input: ... a ....

↑ current token

Left Recursion

- A grammar is ***left recursive*** if it has a non-terminal A such that there is a derivation.

$$A \overset{+}{\Rightarrow} A\alpha \qquad \text{for some string } \alpha$$

- Top-down parsing techniques **cannot** handle left-recursive grammars.
- So, we have to convert our left-recursive grammar into an equivalent grammar which is not left-recursive.
- The left-recursion may appear in a single step of the derivation (*immediate left-recursion*), or may appear in more than one step of the derivation.

Immediate Left-Recursion

$A \rightarrow A\,\alpha \mid \beta$ where β does not start with A

$\Downarrow$ eliminate immediate left recursion

$A \rightarrow \beta\, A'$

$A' \rightarrow \alpha\, A' \mid \varepsilon$ an equivalent grammar

In general,

$A \rightarrow A\,\alpha_1 \mid \ldots \mid A\,\alpha_m \mid \beta_1 \mid \ldots \mid \beta_n$ where $\beta_1 \ldots \beta_n$ do not start with A

$\Downarrow$ eliminate immediate left recursion

$A \rightarrow \beta_1 A' \mid \ldots \mid \beta_n A'$

$A' \rightarrow \alpha_1 A' \mid \ldots \mid \alpha_m A' \mid \varepsilon$ an equivalent grammar

Immediate Left-Recursion -- Example

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

$\Downarrow$ eliminate immediate left recursion

$$E \rightarrow T\,E'$$
$$E' \rightarrow +T\,E' \mid \varepsilon$$
$$T \rightarrow F\,T'$$
$$T' \rightarrow *F\,T' \mid \varepsilon$$
$$F \rightarrow id \mid (E)$$

Left-Recursion -- Problem

- A grammar cannot be immediately left-recursive, but it still can be left-recursive.
- By just eliminating the immediate left-recursion, we may not get a grammar which is not left-recursive.

$S \rightarrow Aa \mid b$
$A \rightarrow Sc \mid d$ This grammar is not immediately left-recursive, but it is still left-recursive.

$\underline{S} \Rightarrow Aa \Rightarrow \underline{S}ca$ or
$\underline{A} \Rightarrow Sc \Rightarrow \underline{A}ac$ causes to a left-recursion

- So, we have to eliminate all left-recursions from our grammar

Eliminate Left-Recursion -- Algorithm

- Arrange non-terminals in some order: $A_1 \ldots A_n$

- **for** i **from** 1 **to** n **do** {

  - **for** j **from** 1 **to** i-1 **do** {

    replace each production

    $A_i \rightarrow A_j \gamma$

    by

    $A_i \rightarrow \alpha_1 \gamma \mid \ldots \mid \alpha_k \gamma$

    where $A_j \rightarrow \alpha_1 \mid \ldots \mid \alpha_k$

    }

  - eliminate immediate left-recursions among A_i productions

}

Eliminate Left-Recursion -- Example

$S \rightarrow Aa \mid b$
$A \rightarrow Ac \mid Sd \mid f$

- Order of non-terminals: S, A

for S:

- we do not enter the inner loop.
- there is no immediate left recursion in S.

for A:

- Replace $A \rightarrow Sd$ with $A \rightarrow Aad \mid bd$
  So, we will have $A \rightarrow Ac \mid Aad \mid bd \mid f$
- Eliminate the immediate left-recursion in A
  $A \rightarrow bdA' \mid fA'$
  $A' \rightarrow cA' \mid adA' \mid \varepsilon$

So, the resulting equivalent grammar which is not left-recursive is:

$S \rightarrow Aa \mid b$
$A \rightarrow bdA' \mid fA'$
$A' \rightarrow cA' \mid adA' \mid \varepsilon$

Eliminate Left-Recursion – Example2

$S \rightarrow Aa \mid b$
$A \rightarrow Ac \mid Sd \mid f$

- Order of non-terminals: A, S

for A:

- we do not enter the inner loop.
- Eliminate the immediate left-recursion in A

$A \rightarrow SdA' \mid fA'$
$A' \rightarrow cA' \mid \varepsilon$

for S:

- Replace $S \rightarrow Aa$ with $S \rightarrow SdA'a \mid fA'a$

  So, we will have $S \rightarrow SdA'a \mid fA'a \mid b$
- Eliminate the immediate left-recursion in S

$S \rightarrow fA'aS' \mid bS'$
$S' \rightarrow dA'aS' \mid \varepsilon$

So, the resulting equivalent grammar which is not left-recursive is:

$S \rightarrow fA'aS' \mid bS'$
$S' \rightarrow dA'aS' \mid \varepsilon$
$A \rightarrow SdA' \mid fA'$
$A' \rightarrow cA' \mid \varepsilon$

Left-Factoring

- A predictive parser (a top-down parser without backtracking) insists that the grammar must be *left-factored*.

stmt → `if` expr `then` stmt `else` stmt |

`if` expr `then` stmt

- when we see `if`, we cannot now which production rule to choose to re-write *stmt* in the derivation.

Left Factoring

- Rewriting productions to delay decisions
- Helpful for predictive parsing
- Not guaranteed to remove ambiguity

$$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$$

$$\Downarrow$$

$$A \rightarrow \alpha A'$$
$$A' \rightarrow \beta_1 \mid \beta_2$$

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 – Example1

$A \rightarrow \underline{a}bB \mid \underline{a}B \mid cdg \mid cdeB \mid cdfB$

$\Downarrow$

$A \rightarrow aA' \mid \underline{cd}g \mid \underline{cd}eB \mid \underline{cd}fB$

$A' \rightarrow bB \mid B$

$\Downarrow$

$A \rightarrow aA' \mid cdA''$

$A' \rightarrow bB \mid B$

$A'' \rightarrow g \mid eB \mid fB$

Left-Factoring – Example2

$A \rightarrow ad \mid a \mid ab \mid abc \mid b$

$\Downarrow$

$A \rightarrow aA' \mid b$

$A' \rightarrow d \mid \varepsilon \mid b \mid bc$

$\Downarrow$

$A \rightarrow aA' \mid b$

$A' \rightarrow d \mid \varepsilon \mid bA''$

$A'' \rightarrow \varepsilon \mid c$

Top Down Parsing

- Can be viewed two ways:
  - Attempt to find leftmost derivation for input string
  - Attempt to create parse tree, starting from at root, creating nodes in preorder
- General form is recursive descent parsing
  - May require backtracking
  - Backtracking parsers not used frequently because not needed

Predictive Parsing

- A special case of recursive-descent parsing that does not require backtracking
- Must always know which production to use based on current input symbol
- Can often create appropriate grammar:
  - removing left-recursion
  - left factoring the resulting grammar

Predictive Parser (example)

stmt → `if` ...... |

`while` ...... |

`begin` ...... |

`for` .....

- When we are trying to write the non-terminal *stmt*, if the current token is `if` we have to choose first production rule.
- When we are trying to write the non-terminal *stmt*, we can uniquely choose the production rule by just looking the current token.
- We eliminate the left recursion in the grammar, and left factor it. But it may not be suitable for predictive parsing (not LL(1) grammar).

Recursive Predictive Parsing

- Each non-terminal corresponds to a procedure.

Ex: $A \rightarrow aBb$ (This is only the production rule for A)

proc A {

- match the current token with a, and move to the next token;
- call 'B';
- match the current token with b, and move to the next token;

}

Recursive Predictive Parsing (cont.)

$A \rightarrow aBb \mid bAB$

```
proc A {
    case of the current token {
        'a':   - match the current token with a, and move to the next token;
               - call 'B';
               - match the current token with b, and move to the next token;
        'b':   - match the current token with b, and move to the next token;
               - call 'A';
               - call 'B';
    }
}
```

Recursive Predictive Parsing (cont.)

- When to apply ε-productions.

  $A \rightarrow aA \mid bB \mid \varepsilon$

- If all other productions fail, we should apply an ε-production. For example, if the current token is not a or b, we may apply the ε-production.
- Most correct choice: We should apply an ε-production for a non-terminal A when the current token is in the follow set of A (which terminals can follow A in the sentential forms).

Transition Diagrams

- For parser:
  - One diagram for each nonterminal
  - Edge labels can be tokens or nonterminal
    - A transition on a token means we should take that transition if token is next input symbol
    - A transition on a nonterminal can be thought of as a call to a procedure for that nonterminal
- As opposed to lexical analyzers:
  - One (or more) diagrams for each token
  - Labels are symbols of input alphabet

Creating Transition Diagrams

- First eliminate left recursion from grammar
- Then left factor grammar
- For each nonterminal A:
  - Create an initial and final state
  - For every production $A \rightarrow X_1X_2\ldots X_n$, create a path from initial to final state with edges labeled X_1, X_2, …, X_n.

Using Transition Diagrams

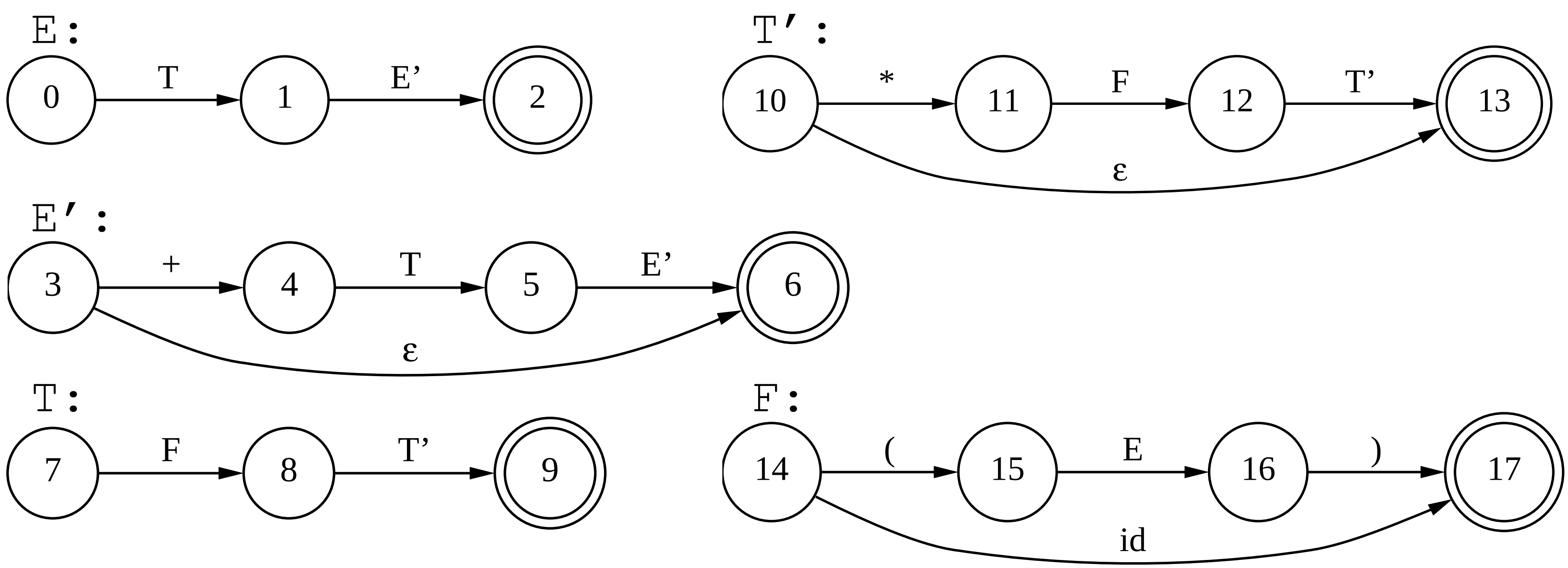
- Predictive parsers:
  - Start at start symbol of grammar
  - From state s with edge to state t labeled with token a, if next input token is a:
    - State changes to t
    - Input cursor moves one position right
  - If edge labeled by nonterminal A:
    - State changes to start state for A
    - Input cursor is not moved
    - If final state of A reached, then state changes to t
  - If edge labeled by ε, state changes to t
- Can be recursive or non-recursive using stack

Transition Diagram Example

E → E + T | T
T → T * F | F
F → (E) | **id**

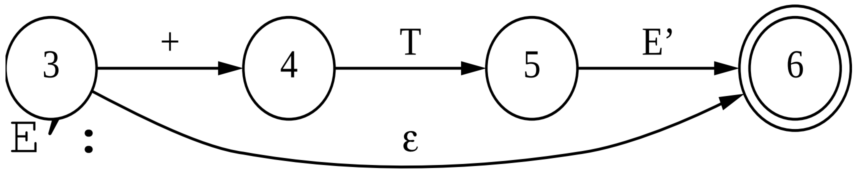
⟹

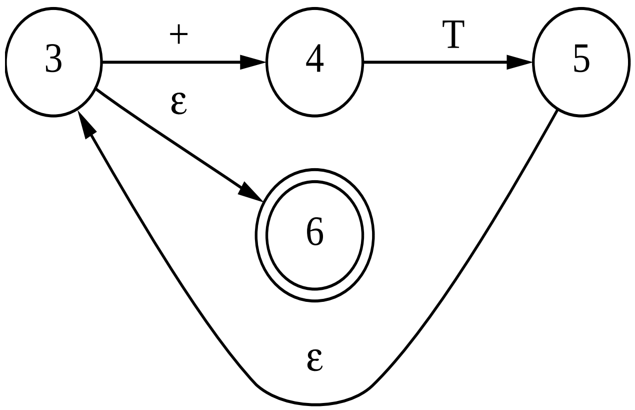
E → TE′
E′ → +TE′ | ε
T → FT′
T′ → *FT′ | ε
F → (E) | **id**

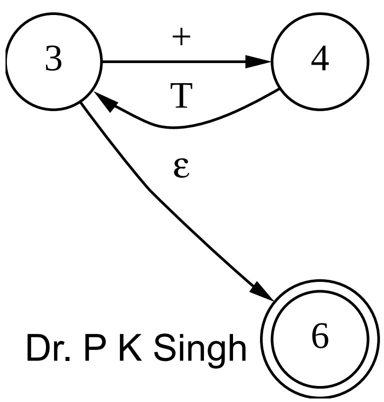


Simplifying Transition Diagrams

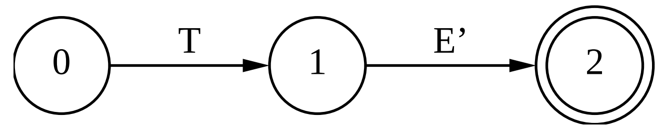
E':

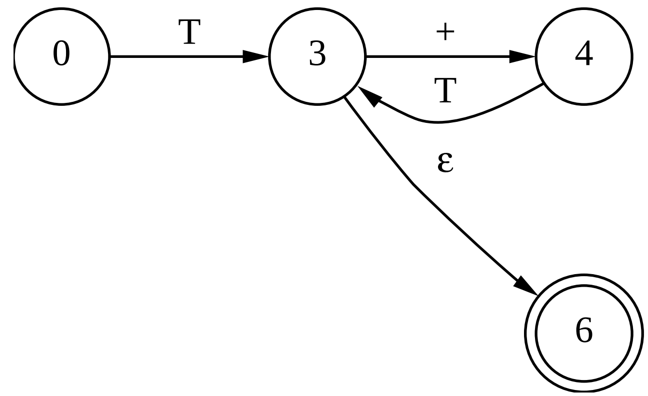


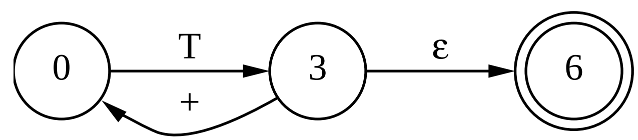




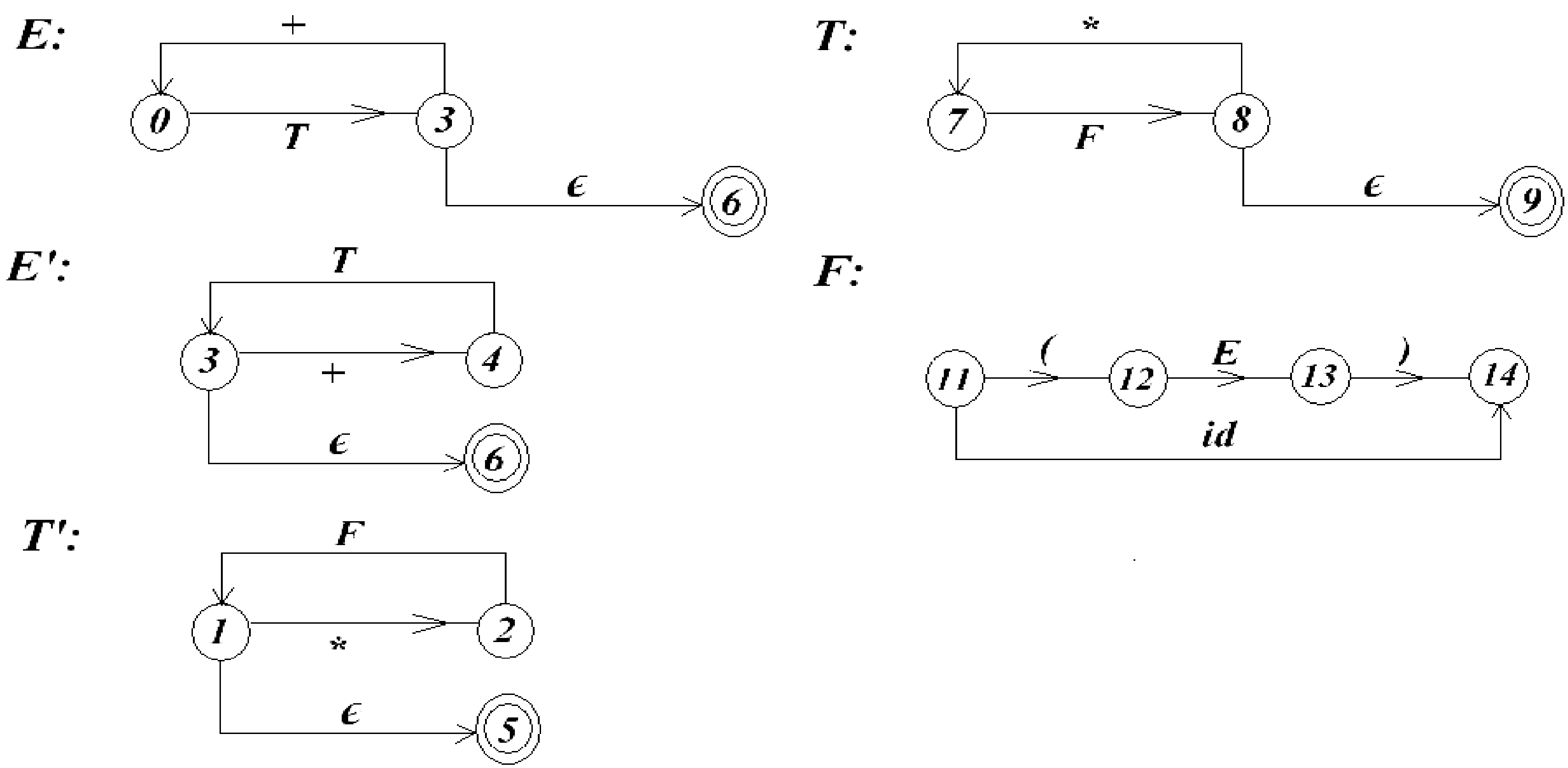
E:







Simplified Transition Diagrams



Recursive Predictive Parsing (Example)

$A \rightarrow aBe \mid cBd \mid C$

$B \rightarrow bB \mid \varepsilon$

$C \rightarrow f$

```
proc A {
    case of the current token {
      a:  - match the current token with a,
            and move to the next token;
          - call B;
          - match the current token with e,
            and move to the next token;
      c:  - match the current token with c,
            and move to the next token;
          - call B;
          - match the current token with d,
            and move to the next token;
      f:  - call C
    }
}
```

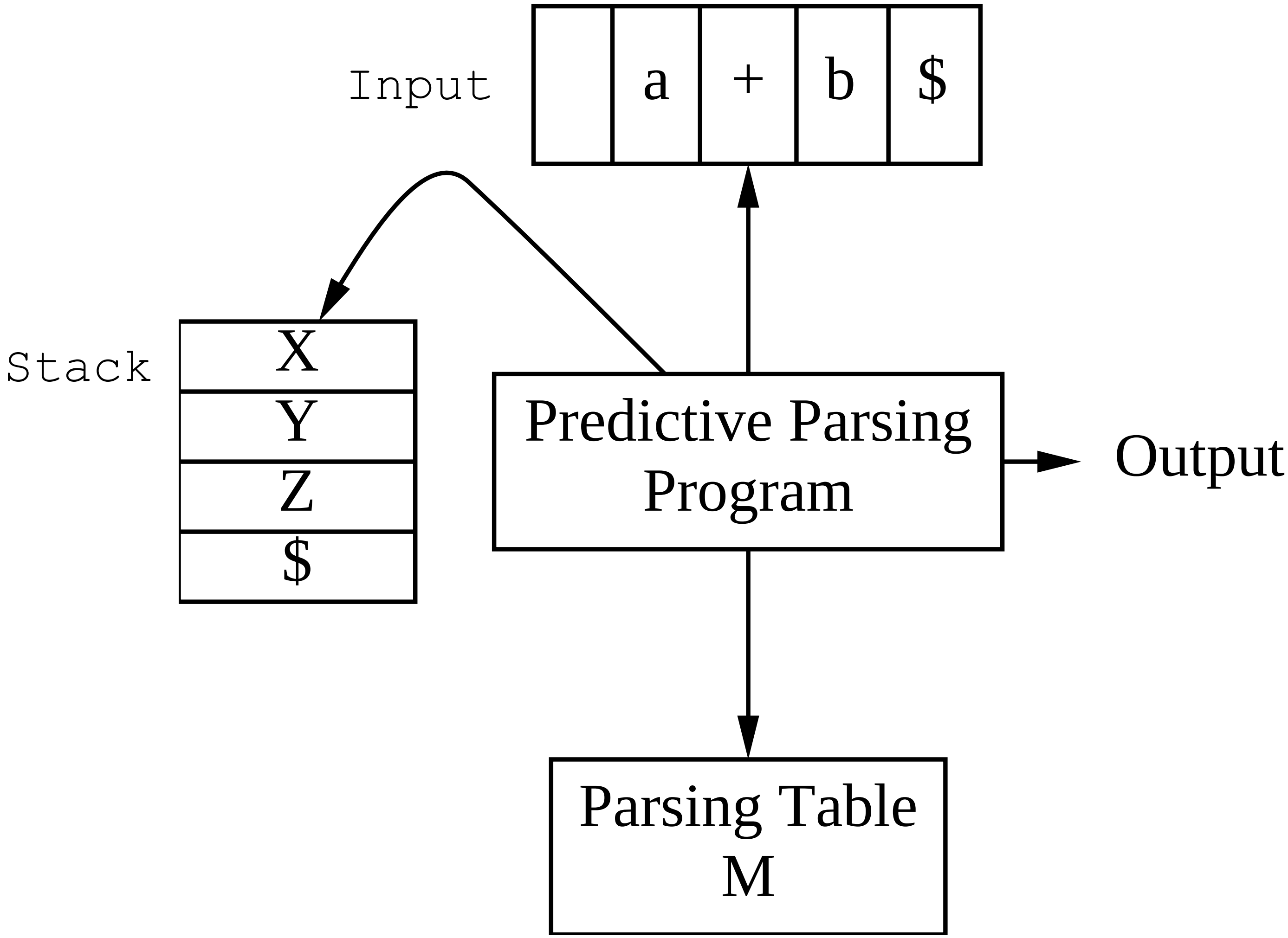
first set of C

```
proc C {   match the current token with f,
           and move to the next token; }

proc B {
    case of the current token {
        b:- match the current token with b,
            and move to the next token;
            - call B
        e,d:  do nothing
    }
}
```

follow set of B

Nonrecursive Predictive Parsing (1)



Nonrecursive Predictive Parsing (2)

- The symbol at the top of the stack (say X) and the current symbol in the input string (say a) determine the parser action.
- There are four possible parser actions.

1. If X and a are \$ ➔ parser halts (successful completion)

2. If X and a are the same terminal symbol (different from \$)

   ➔ parser pops X from the stack, and moves the next symbol in the input buffer.

3. If X is a non-terminal

   ➔ parser looks at the parsing table entry M[X,a]. If M[X,a] holds a production rule $X \to Y_1 Y_2 ... Y_k$, it pops X from the stack and pushes $Y_k, Y_{k-1}, ..., Y_1$ into the stack. The parser also outputs the production rule $X \to Y_1 Y_2 ... Y_k$ to represent a step of the derivation.

4. none of the above ➔ error
   - all empty entries in the parsing table are errors.
   - If X is a terminal symbol different from a, this is also an error case.

Predictive Parsing Table

Nonter-minal	Input Symbol					
	id	+	*	(	)	$
E	E→TE′			E→TE′		
E′		E′→+TE′			E′→ε	E′→ε
T	T→FT′			T→FT′		
T′		T′→ε	T′→*FT′		T′→ε	T′→ε
F	F→id			F→(E)		

Using a Predictive Parsing Table

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′

Stack	Input	Output
…	…	…
$E′T′id	id*id$	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

FIRST

- FIRST(α) is the set of all terminals that begin any string derived from α

- Computing FIRST:
  - If X is a terminal, FIRST(X) = {X}
  - If X→ε is a production, add ε to FIRST(X)
  - If X is a nonterminal and $X \rightarrow Y_1 Y_2 \ldots Y_n$ is a production:
    - For all terminals a, add a to FIRST(X) if a is a member of any FIRST(Y_i) and ε is a member of FIRST(Y_1), FIRST(Y_2), … FIRST(Y_{i-1})
    - If ε is a member of FIRST(Y_1), FIRST(Y_2), … FIRST(Y_n), add ε to FIRST(X)

FOLLOW

- FOLLOW(A), for any nonterminal A, is the set of terminals a that can appear immediately to the right if A in some sentential form
- More formally, a is in FOLLOW(A) if and only if there exists a derivation of the form $S \overset{*}{\Rightarrow} \alpha A a \beta$
- \$ is in FOLLOW(A) if and only if there exists a derivation of the form $S \overset{*}{\Rightarrow} \alpha A$

Computing FOLLOW

- Place `$` in `FOLLOW(S)`
- If there is a production $A \rightarrow \alpha B \beta$, then everything in FIRST(β) (except for ε) is 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 also in FOLLOW(B)

FIRST and FOLLOW Example

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

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

Creating a Predictive Parsing Table

- For each production A → α :
  - For each terminal a in FIRST(α) add A → α to M[A, a]
  - If ε is in FIRST(α) add A → α to M[A, b] for every terminal b in FOLLOW(A)
  - If ε is in FIRST(α) and $ is in FOLLOW(A) add A → α to M[A, $]
- Mark each undefined entry of M as an error entry (use some recovery strategy)

Example

$E \rightarrow TE'$

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

$T \rightarrow FT'$

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

$F \rightarrow (E) \ | \ id$

$\text{FIRST}(E) = \text{FIRST}(T) = \text{FIRST}(F) = \{(, \textbf{id}\}.$

$\text{FIRST}(E') = \{+, \epsilon\}$

$\text{FIRST}(T') = \{*, \epsilon\}$

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

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

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

NONTER-MINAL	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	$F \rightarrow \textbf{id}$			$F \rightarrow (E)$		

Constructing LL(1) Parsing Table -- Example

$E \to TE'$	$FIRST(TE')=\{(,id\}$	➔ $E \to TE'$ into $M[E,(]$ and $M[E,id]$
$E' \to +TE'$	$FIRST(+TE')=\{+\}$	➔ $E' \to +TE'$ into $M[E',+]$
$E' \to \varepsilon$	$FIRST(\varepsilon)=\{\varepsilon\}$	➔ none
	but since ε in $FIRST(\varepsilon)$	
	and $FOLLOW(E')=\{\$,)\}$	➔ $E' \to \varepsilon$ into $M[E',\$]$ and $M[E',)]$
$T \to FT'$	$FIRST(FT')=\{(,id\}$	➔ $T \to FT'$ into $M[T,(]$ and $M[T,id]$
$T' \to *FT'$	$FIRST(*FT')=\{*\}$	➔ $T' \to *FT'$ into $M[T',*]$
$T' \to \varepsilon$	$FIRST(\varepsilon)=\{\varepsilon\}$	➔ none
	but since ε in $FIRST(\varepsilon)$	
	and $FOLLOW(T')=\{\$,),+\}$	➔ $T' \to \varepsilon$ into $M[T',\$]$, $M[T',)]$ and $M[T',+]$
$F \to (E)$	$FIRST((E))=\{(\}$	➔ $F \to (E)$ into $M[F,(]$
$F \to id$	$FIRST(id)=\{id\}$	➔ $F \to id$ into $M[F,id]$

LL(1) Grammars

- A grammar whose parsing table has no multiply-defined entries is said to be LL(1) grammar.

one input symbol used as a look-head symbol do determine parser action

LL(1) left most derivation

input scanned from left to right

- The parsing table of a grammar may contain more than one production rule. In this case, we say that it is not a LL(1) grammar.

A Grammar which is not LL(1)

S → i C t S E | a

E → e S | ε

C → b

FOLLOW(S) = { $,e }

FOLLOW(E) = { $,e }

FOLLOW(C) = { t }

FIRST(iCtSE) = {i}

FIRST(a) = {a}

FIRST(eS) = {e}

FIRST(ε) = {ε}

FIRST(b) = {b}

	a	b	e	i	t	$
S	S → a			S → iCtSE		
E			E → e S E → ε			E → ε
C		C → b				

two production rules for M[E,e]

A Grammar which is not LL(1) (cont.)

- What do we have to do it if the resulting parsing table contains multiply defined entries?
  - If we didn't eliminate left recursion, eliminate the left recursion in the grammar.
  - If the grammar is not left factored, we have to left factor the grammar.
  - If its (new grammar's) parsing table still contains multiply defined entries, that grammar is ambiguous or it is inherently not a LL(1) grammar.
- A left recursive grammar cannot be a LL(1) grammar.
  - $A \rightarrow A\alpha \mid \beta$

    ➔ any terminal that appears in FIRST(β) also appears FIRST($A\alpha$) because $A\alpha \Rightarrow \beta\alpha$.

    ➔ If β is ε, any terminal that appears in FIRST(α) also appears in FIRST($A\alpha$) and FOLLOW(A).
- A grammar is not left factored, it cannot be a LL(1) grammar
  - $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$

    ➔ any terminal that appears in FIRST($\alpha\beta_1$) also appears in FIRST($\alpha\beta_2$).
- An ambiguous grammar cannot be a LL(1) grammar.

Properties of LL(1) Grammars

- A grammar G is LL(1) if and only if the following conditions hold for two distinctive production rules $A \rightarrow \alpha$ and $A \rightarrow \beta$
  1. Both α and β cannot derive strings starting with same terminals.
  2. At most one of α and β can derive to ε.
  3. If β can derive to ε, then α cannot derive to any string starting with a terminal in FOLLOW(A).

- A Grammar to be LL(1), following conditions must satisfied:

  For every pair of productions $A \rightarrow \alpha \mid \beta$

  {

  $FIRST(\alpha) \cap FIRST(\beta) = \Phi$

  and if FIRST(β) contains ε then

  $FIRST(\alpha) \cap FOLLOW(A) = \Phi$

  }

Example

- Test the Following Grammar is LL(1) or not ?

S → 1AB | ε

A → 1AC | 0C

B → 0S

C → 1

For Production S → 1AB | ε

FIRST(1AB) ∩ FIRST(ε) = {1} ∩ {ε} = Φ and

FIRST(1AB) ∩ FOLLOW(S) = {1} ∩ {$} = Φ

Similarly A → 1AC | 0C

FIRST(1AC) ∩ FIRST(0C) = {1} ∩ {0} = Φ

Hence The Grammar is LL(1)