COMPILER DESIGN

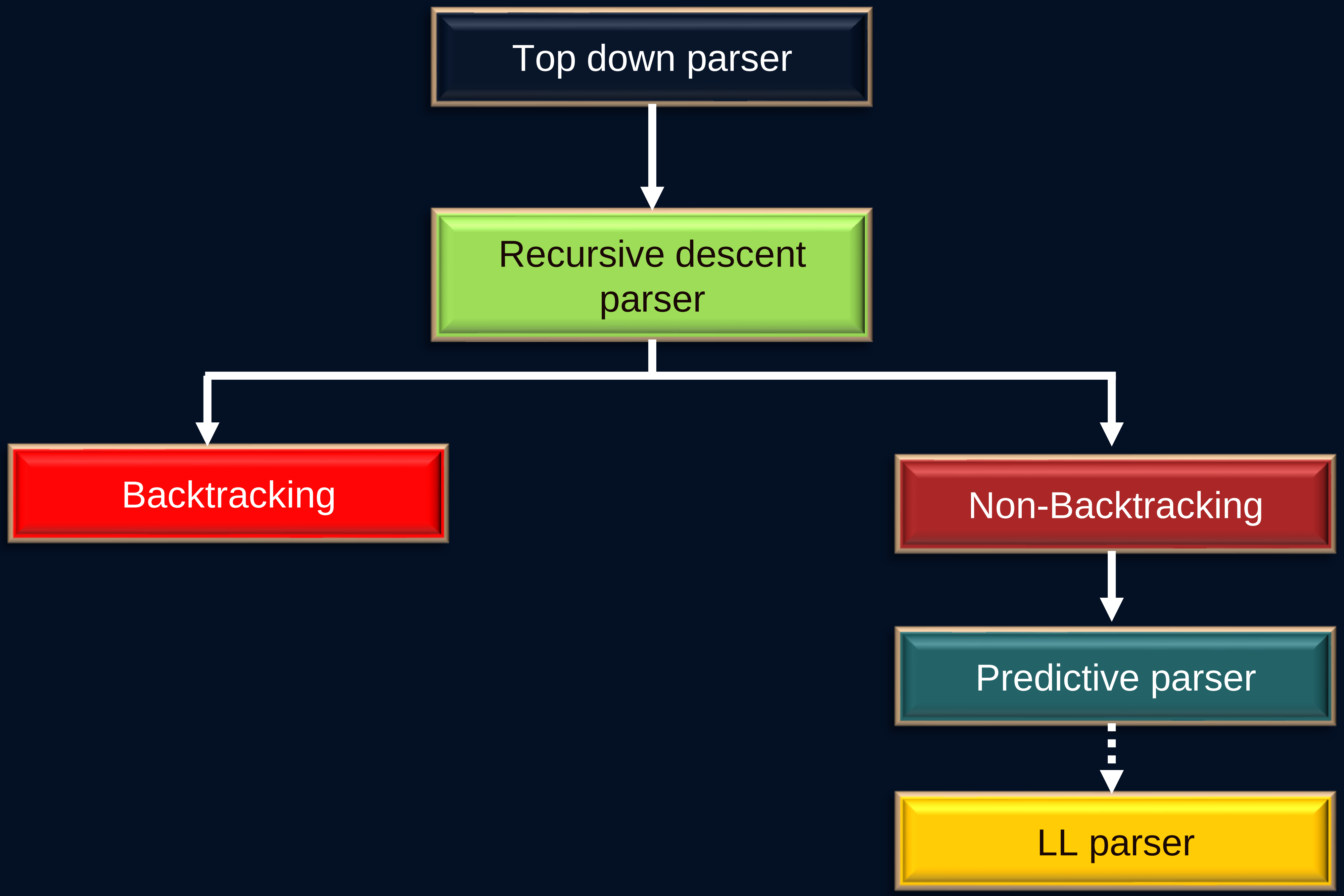
HANDLED BY
DIVYA B
DEPT. OF CSE
VJEC, CHEMPERI

TOP DOWN PARSING

- Parsing is the process of determining if a string of tokens can be generated by a grammar.
- There are mainly two parsing approaches
  - Top down parsing
  - Bottom up parsing
- In **top down parsing**, parse tree is constructed from top (root) to the bottom (leaves).
- In **bottom up parsing**, parse tree is constructed from bottom (leaves) to the top (root).

TOP DOWN PARSING

- Top down parsing can be viewed as an attempt to find a leftmost derivation for an input string (that is expanding the leftmost terminal at every step).


Top down parser
Recursive descent parser
Backtracking
Non-Backtracking
Predictive parser
LL parser

TOP DOWN PARSING

```
void A() {
    Choose an A-production, A → X1 X2 ··· 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 */;
    }
}
```

A typical procedure for a nonterminal in a top down parser

RECURSIVE DESCENT PARSING

- It is the most general form of top down parsing.
- It may involve **backtracking**, that is making repeated scans of input, to obtain the correct expansion of the leftmost non-terminal.
- Unless the grammar is ambiguous or left-recursive, it finds a suitable parse tree.

RECURSIVE DESCENT PARSING

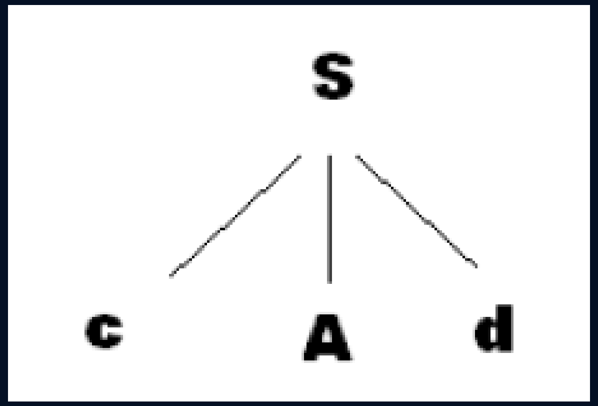
- Consider the grammar

$$S \rightarrow cAd$$
$$A \rightarrow ab|\mathrm{a}$$

and the input string w = cad.

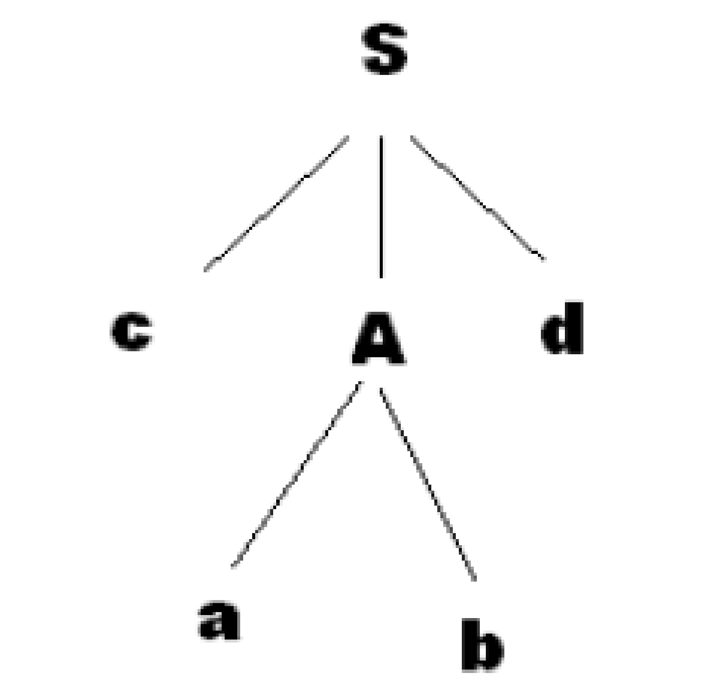
RECURSIVE DESCENT PARSING

- To construct a parse tree for this string top down, we initially create a tree consisting of a single node labelled **S**.
- An input pointer points to **c**, the first symbol of w.
- **S** has only one production, so we use it to expand **S** and obtain the tree as



RECURSIVE DESCENT PARSING

- The leftmost leaf, labeled **c**, matches the first symbol of input w, so we advance the input pointer to **a**, the second symbol of **w**, and consider the next leaf, labeled A.
- **A is expanded** using the first alternative A → **ab** to obtain the tree as

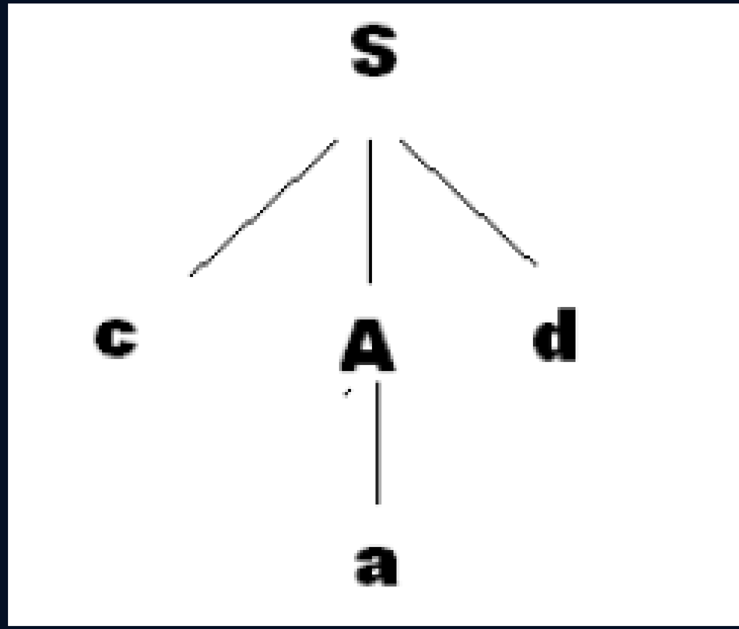


RECURSIVE DESCENT PARSING

- We have a match for the second input symbol, **a**, so we advance the input pointer to **d**, the third input symbol, and compare d against the next leaf, labeled **b**.
- Since **b** does not match **d**, failure is reported and we must go back to **A** to see whether there is another alternative for **A** that has not been tried which might produce a match.

RECURSIVE DESCENT PARSING

- In going back to **A**, we must reset the input pointer to position 2 , the position it had when we first came to **A**, which means that the procedure for **A** must store the input pointer in a local variable.
- The second alternative for **A** produces the tree as



RECURSIVE DESCENT PARSING

- The leaf **a** matches the second symbol of **w** and the leaf **d** matches the third symbol.
- Since we have produced a parse tree for **w**, we halt and announce successful completion of parsing.
- The string is parsed completely and the parser stops.

PREDICTIVE PARSING

- A predictive parsing is a special form of recursive-descent parsing, in which the current input token unambiguously determines the production to be applied at each step.
- The goal of predictive parsing is to construct a top-down parser that never backtracks.

PREDICTIVE PARSING

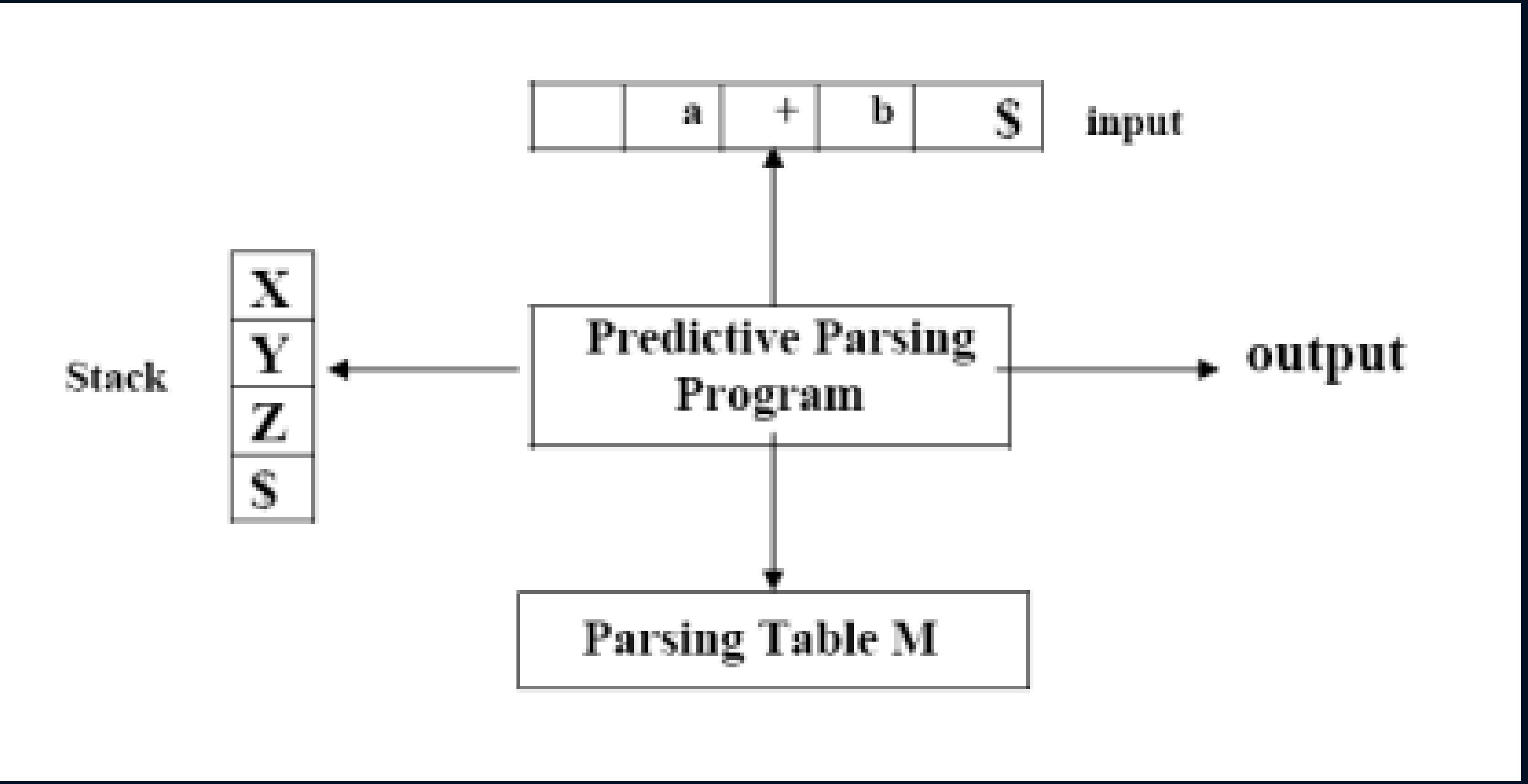
- To do so we must transform the grammar in two ways
  - Eliminate left recursion
  - Perform left factoring
- These rules eliminate most common causes for backtracking although they do not guarantee a completely backtrack-free parsing .

PREDICTIVE PARSING

- It is possible to build a nonrecursive predictive parser by maintaining a stack explicitly, rather than implicitly via recursive calls.
- The key problem during predictive parsing is that of determining the production to be applied for a nonterminal.
- The nonrecursive parser in looks up the production to be applied in a parsing table.

PREDICTIVE PARSING

- Requirements
  - Stack
  - Parsing table
  - Input buffer
  - Output stream



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 is in a configuration in which it 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 shown below.

set ip to the first symbol of $w\$$
repeat
  let X be the top of the stack;
  let a be the symbol pointed by ip;
  if $X \in V_T$ **or** $X = \$$ **then**
    if $X = a$ **then**
      pop the stack;
      advance ip;
    else error
  else
    let $X \longmapsto Y_1 Y_2 \cdots Y_k$ be $M[X, a]$;
    if no such production **then error**;
    pop X from the stack;
    push $Y_k, Y_{k-1}, \cdots, Y_1$ onto the stack;
    output $X \longmapsto Y_1 Y_2 \cdots Y_k$;
until $X = \$$

PREDICTIVE PARSING

- Consider the input string as id+id*id and the grammar as

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

$$E \underset{lm}{\Rightarrow} TE' \underset{lm}{\Rightarrow} FT'E' \underset{lm}{\Rightarrow} \mathbf{id}\,T'E' \underset{lm}{\Rightarrow} \mathbf{id}\,E' \underset{lm}{\Rightarrow} \mathbf{id} + TE' \underset{lm}{\Rightarrow} \cdots$$

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 $E' \to \epsilon$

Figure 4.21: Moves made by a predictive parser on input $\mathbf{id} + \mathbf{id} * \mathbf{id}$

PREDICTIVE PARSING

- Construction of the parsing table is aided by two functions
  - FIRST
  - FOLLOW

PREDICTIVE PARSING

FIRST

- If 'α' is any string of grammar symbols, then FIRST(α) be the set of terminals that begin the string derived from α.
- If $\alpha \overset{*}{\Rightarrow} \varepsilon$ then ε is added to FIRST(α).
- First is defined for both terminals and non terminals.

PREDICTIVE PARSING

FIRST

- If **X** is **a** terminal , then FIRST(**X**) is {**X**}
- If **X**→ε then add ε to FIRST(**X**)
- If **X** is a non terminal and $\mathbf{X \rightarrow Y_1 Y_2 Y_3 \ldots Y_n}$, then put '**a**' in FIRST(**X**) if for some **i**, **a** is in FIRST($\mathbf{Y_i}$) and ε is in all of FIRST($\mathbf{Y_1}$),...FIRST($\mathbf{Y_{i-1}}$).

PREDICTIVE PARSING

FIRST

- Find the FIRST( ) for the production
  - $S \rightarrow abc \mid def \mid ghi$
  - $FIRST(S) = \{a, d, g\}$

- Find the FIRST( ) for the productions given below
  - $S \rightarrow ABC \mid ghi \mid jkl$
  - $A \rightarrow a \mid b \mid c$
  - $B \rightarrow b$
  - $D \rightarrow d$

FIRST(D) = {d}

FIRST(B) = {b}

FIRST(A) = {a, b, c}

FIRST(S) = {FIRST(A), g, j} = {a, b, c, g, j}

- Find the FIRST( ) for the production
  - $S \rightarrow ABC$
  - $A \rightarrow a \mid b \mid \varepsilon$
  - $B \rightarrow c \mid d \mid \varepsilon$
  - $C \rightarrow e \mid f \mid \varepsilon$

FIRST(C) = {e, f, ε}

FIRST(B) = {c, d, ε}

FIRST(A) = {a, b, ε}

FIRST(S) = {FIRST(A)} = {a, b, c, d, e, f, ε}

FIRST (A) contains ε. Replacing A with ε in S makes $S \rightarrow BC$. So FIRST (B) has to be taken. Similarly FIRST(C).

- Find the FIRST( ) for the productions
  - $E \rightarrow TE'$
  - $E' \rightarrow +TE' \mid \varepsilon$
  - $T \rightarrow FT'$
  - $T' \rightarrow *FT' \mid \varepsilon$
  - $F \rightarrow (E) \mid id$

FIRST(F) = {(, id}

FIRST(T′) = {*, ε}

FIRST(T) = FIRST(F) = {(, id}

FIRST(E′) = {+, ε}

FIRST(E) = {FIRST(T)} = {(, id}

PREDICTIVE PARSING

FOLLOW

- FOLLOW is defined only for non-terminals of the grammar G.
- It can be defined as the set of terminals of grammar **G** , which can immediately follow the non-terminal in a production rule from start symbol.
- In other words, if **A** is a nonterminal, then FOLLOW(**A**) is the set of terminals '**a**' that can appear immediately to the right of **A** in some sentential form.

PREDICTIVE PARSING

FOLLOW

- If S is the start symbol, then add $ to FOLLOW(S).
- If there is a production rule **A→*α*Bβ** then everything in FIRST(β) except for ε is placed in FOLLOW(**B**).
- If there is a production **A→*α*B** , or a production **A→*α*Bβ** where FIRST(β) contains ε then everything in FOLLOW(**A**) is in FOLLOW(**B**).
- ε is never part of FOLLOW.

PREDICTIVE PARSING

FOLLOW

- For finding follow of a non-terminal check for that non-terminal only on the RHS of the production. i.e., to find FOLLOW(A) search for non-terminal A on the right hand side of the production.

- Find the FOLLOW( ) for the production
  - $S \rightarrow ACD$

$$\boxed{FOLLOW(S) = \{\$\}}$$

- Find the FOLLOW( ) for the production
  - $S \rightarrow aSbS \mid bSaS \mid \varepsilon$

$$\boxed{FOLLOW(S) = \{b, a, \$\}}$$

- Find the FOLLOW( ) for the productions
  - $S \rightarrow AaAb \mid BbBa$
  - $A \rightarrow \varepsilon$
  - $B \rightarrow \varepsilon$

FOLLOW(S) = {\$}

FOLLOW(A) = {a, b}

FOLLOW(B) = {a, b}

- Find the FOLLOW( ) for the productions
  - $S \rightarrow ABC$
  - $A \rightarrow DEF$
  - $B \rightarrow \varepsilon$
  - $C \rightarrow \varepsilon$
  - $D \rightarrow \varepsilon$
  - $E \rightarrow \varepsilon$
  - $F \rightarrow \varepsilon$

FIRST(B) contains ε. ε cannot be there in FOLLOW so ε needs to be replaced in the production for B making it $S \rightarrow AC$

FOLLOW(S) = {$}

FOLLOW(A) = FIRST(B) = FIRST (C) = FOLLOW(S)= {$}

- Find the FOLLOW( ) for the production
  - $E \rightarrow TE'$
  - $E' \rightarrow +TE' \mid \varepsilon$
  - $T \rightarrow FT'$
  - $T' \rightarrow *FT' \mid \varepsilon$
  - $F \rightarrow (E) \mid id$

FOLLOW(E) = {\$,)}

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

FOLLOW(T) = FIRST(E')
= {+, FOLLOW(E)} ('coz of ε in FIRST(E')
= {+,), \$)

- Find the FOLLOW( ) for the production
  - $E \rightarrow TE'$
  - $E' \rightarrow +TE' \mid \varepsilon$
  - $T \rightarrow FT'$
  - $T' \rightarrow *FT' \mid \varepsilon$
  - $F \rightarrow (E) \mid id$

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

FOLLOW(F) = FIRST(T')
= {*, FOLLOW(E)} ('coz of ε in FIRST(T')
= {*, +,), $}

- FIRST and for the following productions are
  - $E \rightarrow TE'$
  - $E' \rightarrow +TE' \mid \varepsilon$
  - $T \rightarrow FT'$
  - $T' \rightarrow *FT' \mid \varepsilon$
  - $F \rightarrow (E) \mid id$

Non-terminal	FIRST	FOLLOW
E	(, id	), $
E'	+, ∈	), $
T	(, id	+,), $
T'	*, ∈	+,), $
F	(, id	+, *,), $

- Find the FIRST( ) and FOLLOW( ) for the productions
  - $S \rightarrow A$
  - $A \rightarrow BC \mid DBC$
  - $B \rightarrow bB' \mid \varepsilon$
  - $B' \rightarrow bB' \mid \varepsilon$
  - $C \rightarrow c \mid \varepsilon$
  - $D \rightarrow a \mid d$

Non-terminal	FIRST	FOLLOW
S	{a, b, d, ε}	{\$}
A	{a, b, d, ε}	{\$}
B	{b, ε}	{c, \$}
C	{c, ε}	{\$}
D	{a, d}	{b, c, \$}

Algorithm for 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(A), add *A→α* to **M[A, a].**
  3. If ε is in FIRST(α), then for each terminal *b* in FOLLOW (**A**), add **A**→α to **M[A, b]** . If ε is in FIRST (α) and **$** is in FOLLOW(**A**), add A→α to M[A, $].
  4. Mark each undefined entry for M as error.

NON-TERMINAL	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)		

- Start from the start symbol, E. In the FIRST(E) column add the production E→TE'.
- Since FIRST(E') has ε, add E→ε in columns corresponding to FOLLOW(E).
- Similarly, fill the rest of the entries.

LL(1) GRAMMARS

- Predictive parsers, that is, recursive descent parsers needing no backtracking can be constructed for a class of grammars called LL(1).
- If the parsing table of a CFG does not have multiple entries then it is called LL(1).

LL(1) GRAMMARS

- In LL(1),
  - First L stands for scanning the input from left to right.
  - Second L stands for producing a leftmost derivation.
  - 1 stands for using one input symbol of lookahead at each step to make parsing action decisions.
- No left recursive or ambiguous grammar can be LL(1).
- A language is said to be LL(1) if it can be generated by a LL(1) grammar.

LL(1) GRAMMARS

- Check whether the dangling-else problem is LL(1) or not.

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

Non-terminal	FIRST	FOLLOW
S	i, a	e, $
S'	e, ∈	e, $
E	b	t

Non-terminal	i	t	a	e	b	$
S	$S \longrightarrow i\ E\ t\ S\ S'$		$S \longrightarrow a$			
S'				$S' \longrightarrow e\ S$ $S' \longrightarrow \epsilon$		$S' \longrightarrow \epsilon$
E					$E \longrightarrow b$	

More than one productions are there in a single cell and so dangling-else is not LL(1).