

# COMPILER DESIGN

## CS6109 MODULE 5 & 6

BE COMPUTER SCIENCE AND ENGINEERING  
REGULATION 2018 RUSA

Thanasekhar B

# COURSE OBJECTIVES : CS6109 COMPILER DESIGN

- To know about the various transformations in the different phases of the compiler, error handling and means of implementing the phases
- To learn about the techniques for tokenization and parsing
- To understand the ways of converting a source language to intermediate representation
- To have an idea about the different ways of generating assembly code
- To have a brief understanding about the various code optimization techniques

# CS6109 COMPILER DESIGN

## ■ MODULE 5

- Recursive Descent Parsers
- LL(1) Parsers
- Shift Reduce Parser
- LR(0) items
- Simple LR parser

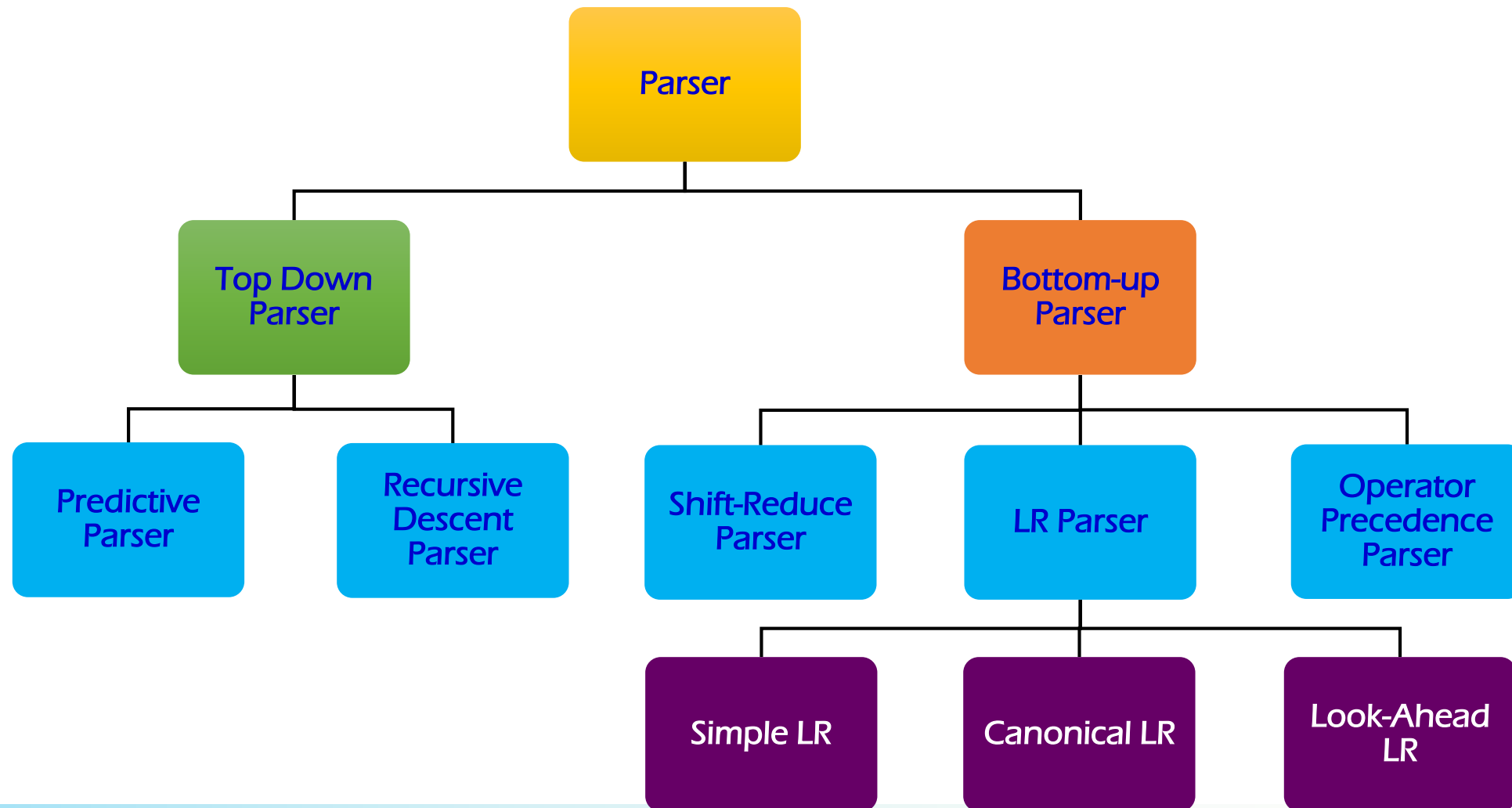
## ■ MODULE 6

- CLR Parser
- LALR Parser
- Parser Generators
- Design of a Parser Generator

## ■ Extra Learning Component

- Parser Design for Various Programming constructs using LEX and YACC
- Parsing Table Construction for Different Grammars using the Table Construction Algorithms

# Types of Parsers



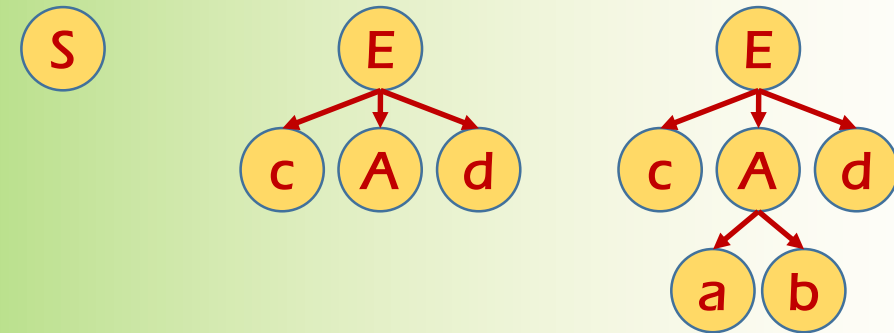
# Top Down Parsing

**Constructs parse tree for the string from the root to leaves**

# Top Down Parsing

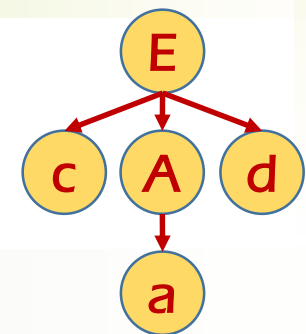
- Top-down parsers constructs a parse tree for the input string, starting from the root and creating the nodes of the parse tree in preorder (depth first)
- Also, the top-down parsing can be viewed as finding a leftmost derivation for an input string
- The class of grammars for which we can construct predictive parsers looking k symbols ahead in the input is sometimes called the LL(k) class
- The top down parsers require context free grammars without left recursion and left factoring
- Top down parsing may involve backtracking

- Consider the grammar  $S \rightarrow cAd, A \rightarrow ab|a$  and the input string and the input string **cad**.
- The following trees are constructed before being forced to back track



- Now the parser finds that the string "cad" cannot be generated and hence does backtracking by selecting the next alternate production for the non terminal A

- With  $A \rightarrow a$ , the parse tree gives the yield "cad", which is same as the given string and the parsing is completed

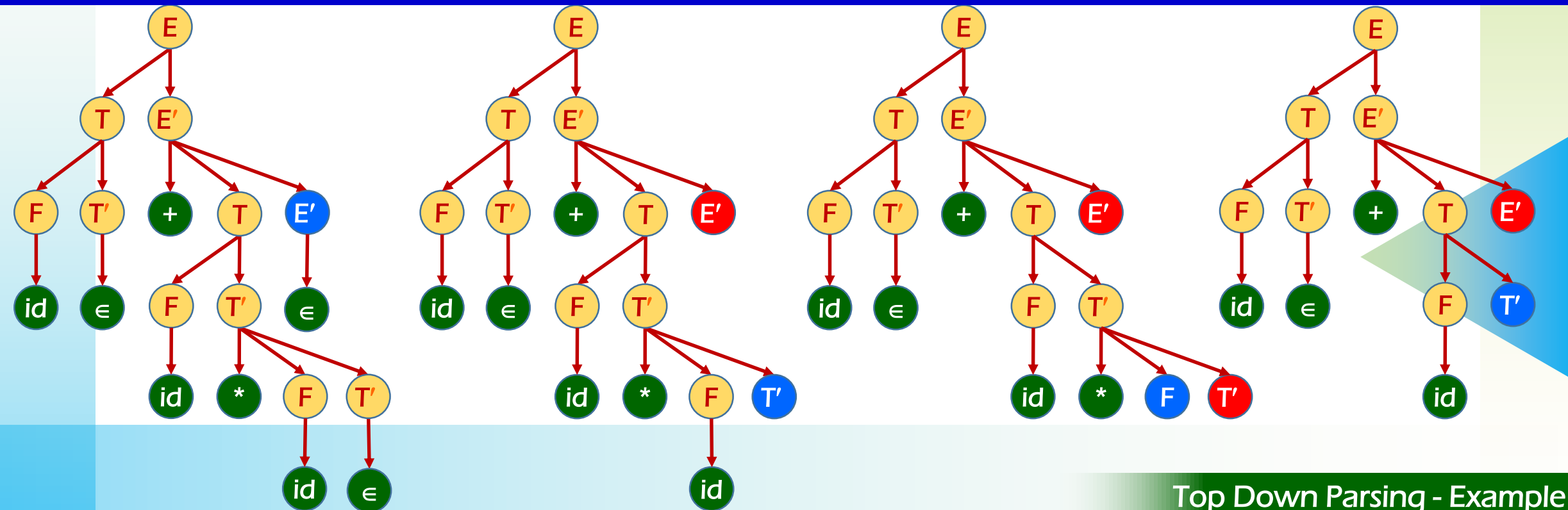
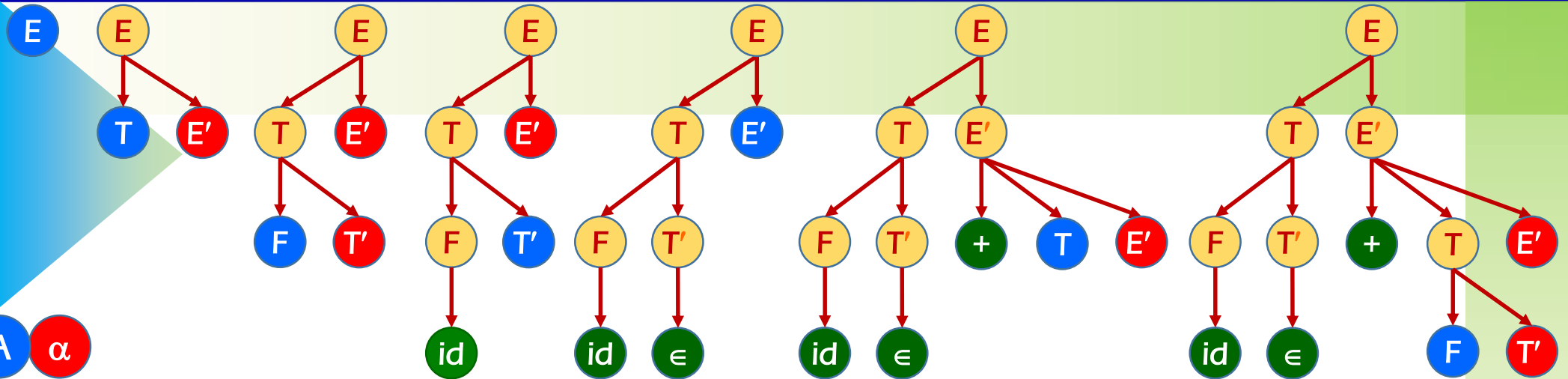


## Grammar

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

## String

id+id\*id



Top Down Parsing - Example

# Recursive Descent Parser

A recursive-descent parser consists of a set of procedures, one for each nonterminal, and the execution begins with the procedure for the start symbol, which halts and announces success if its procedure body scans the entire input string

$E \rightarrow TE'$        $E' \rightarrow +TE' / \epsilon$

$T \rightarrow FT'$        $T' \rightarrow *FT' / \epsilon$

$F \rightarrow (E) / \text{id}$

The mutually recursive procedures used by the parser are listed here.

**procedure E()**

```
begin
  T();
  EPRIME();
end;
```

**procedure EPRIME()**

```
begin
  If input == '+' then begin
    ADVANCE();
    T();
    EPRIME();
  end;
end;
```

**procedure T()**

```
begin
  F();
  TPRIME();
end;
```

**procedure TPRIME()**

```
begin
  If input == '*' then begin
    ADVANCE();
    F();
    TPRIME();
  end;
end;
```

**procedure F()**

```
begin
  if input == 'id'
  then ADVANCE();
  else if input == '('
  then begin
    ADVANCE();
    E();
    if input == ')'
    then ADVANCE();
    else ERROR();
  end
  else ERROR();
end;
```

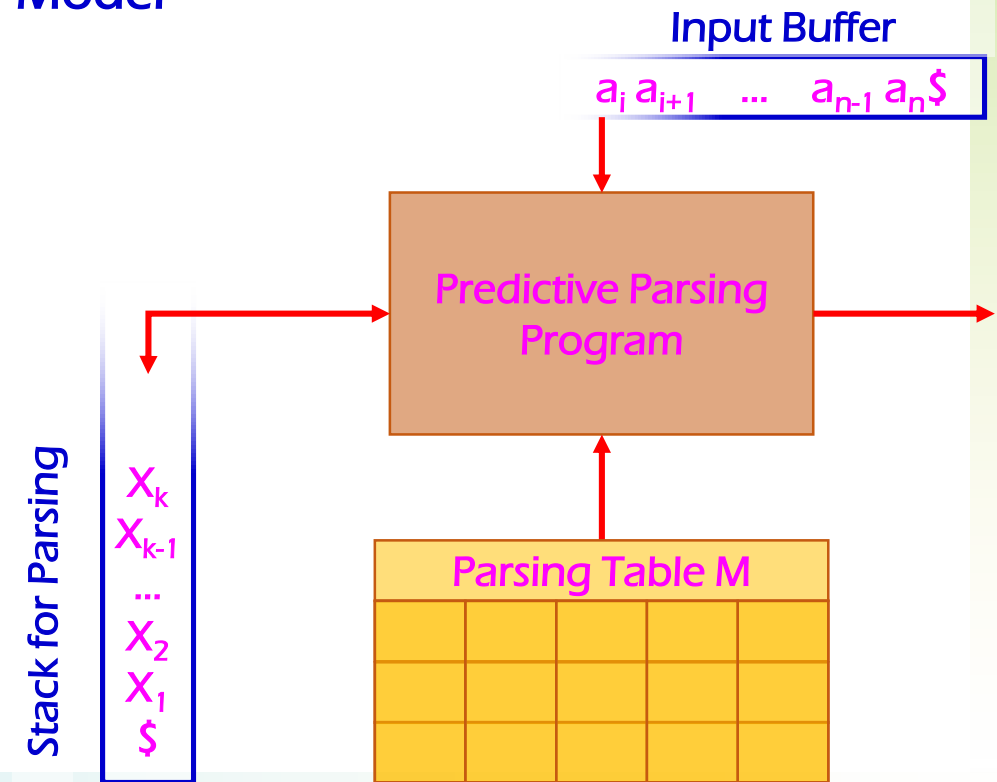


# Predictive Parser

The tabular implementation of the Top Down Parser is called Predictive Parser. The stack of activation records is explicitly maintained by the parser rather than by the language in which the parser is written.

## Predictive Parser Model

- The parser has an input buffer, used to store the input string to be parsed and terminator \$
- The stack contains the sequence of grammar symbols, preceded by \$ which denotes the bottom of the stack
- The stack is initialized by \$ and the start symbol of the grammar
- The parsing table is a 2-dimensional array M, in which the rows are indexed by the non terminals and the columns are indexed by the terminals along with \$.



# FIRST and FOLLOW

- The Predictive Parsing Table construction uses two sets of terminals namely FIRST and FOLLOW
- While parsing FIRST and FOLLOW allow us to choose the production to apply, based on the next input symbol
- During panic-mode error recovery, FOLLOW can be used as synchronizing tokens

## FIRST

Consists of set of terminals, that may occur as the first terminal in the sentences derived from the respective variables

1. If  $X$  is a terminal then  $\text{FIRST}(X)$  is  $\{X\}$
2. If  $X$  is a non-terminal and  $X \rightarrow Y_1 Y_2 \dots Y_k$  for  $k \geq 1$  is a production, then add 'a' to  $\text{FIRST}(X)$ , if for some  $i$ ,  $a$  is in  $\text{FIRST}(Y_i)$  and  $Y_1 Y_2 \dots Y_{i-1} \xRightarrow{*} \epsilon$
3. If  $\epsilon$  is in  $\text{FIRST}(Y_j)$  for all  $j = 1, 2, \dots, k$  then add  $\epsilon$  to  $\text{FIRST}(X)$
4. If  $X \rightarrow \epsilon$  is a production then add  $\epsilon$  to  $\text{FIRST}(X)$

## FOLLOW

Consists of the set of terminals that can appear immediately to the right of  $A$  in some sentential form  $\alpha A a \beta$  for some  $\alpha$  and  $\beta$

1.  $\$$  is in  $\text{FOLLOW}(S)$ , where  $S$  is the start symbol
2. If there is a production of the form  $A \rightarrow \alpha B \beta$ ,  $\beta \neq \epsilon$ , then everything in  $\text{FIRST}(\beta)$  except  $\epsilon$  is in  $\text{FOLLOW}(B)$
3. If there is a production  $A \rightarrow \alpha B$  or  $A \rightarrow \alpha B \beta$ ,  $\beta \xRightarrow{*} \epsilon$ , then everything in  $\text{FOLLOW}(A)$  is in  $\text{FOLLOW}(B)$
4. Apply the rules 2, 3 repeatedly until nothing can be added to any of the FOLLOW set


# FIRST and FOLLOW Computation

## Grammar Productions

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

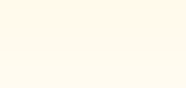
## FIRST Set


$E = \{ (, id \}$   
 $E' = \{ +, \epsilon \}$   
 $T = \{ (, id \}$   
 $T' = \{ *, \epsilon \}$   
 $F = \{ (, id \}$



 $A \rightarrow \alpha$

## FOLLOW Set

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


 $A \rightarrow \alpha B \beta, \beta \neq \epsilon$


 $A \rightarrow \alpha B$


 $A \rightarrow \alpha B \beta, \beta \xRightarrow{*} \epsilon$

# Predictive Parsing Table Construction

## Grammar Productions

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

## FIRST Set

$$\begin{aligned} E &= \{ (, id \} \\ E' &= \{ +, \epsilon \} \\ T &= \{ (, id \} \\ T' &= \{ *, \epsilon \} \\ F &= \{ (, id \} \end{aligned}$$

## FOLLOW Set

$$\begin{aligned} E &= \{ \$, ) \} \\ E' &= \{ \$, ) \} \\ T &= \{ \$, ), + \} \\ T' &= \{ \$, ), + \} \\ F &= \{ \$, ), +, * \} \end{aligned}$$

## Table Construction Algorithm

1. For each production of the form  $A \rightarrow \alpha$ , do the steps 2 and 3.
2. For each terminal 'a' in FIRST ( $\alpha$ ), add  $A \rightarrow \alpha$  to  $M[A, a]$ .
3. If  $\epsilon$  is in FIRST ( $\alpha$ ) then for each terminal 'b' in FOLLOW (A) add  $A \rightarrow \alpha$  to  $M[A, b]$ .
4. If  $\epsilon$  is in FIRST ( $\epsilon$ ) and  $\$$  is in FOLLOW (A) then add  $A \rightarrow \alpha$  to  $M[A, \$]$ .
5. Make each undefined entry of M as error.

$T \rightarrow$ N↓	+	*	(	)	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			$F \rightarrow (E)$		$F \rightarrow id$	

# Parsing with Predictive Parser

## Predictive Parsing Algorithm

Initialize the buffer with  $w\$$ ;

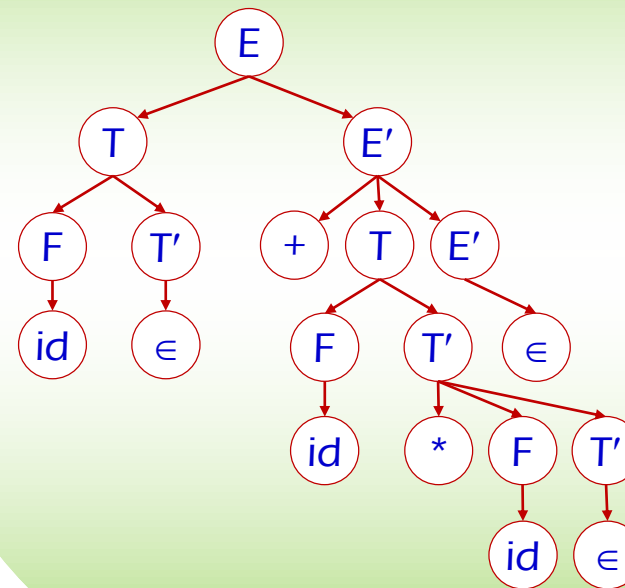
Initialize the stack with  $\$$ ;

Let  $X$  be the top stack and ' $a$ ' the next input;

while ( $X \neq \$$ )

```
{
  if ( $X == a$ ) then  pop  $X$ ;
                    remove ' $a$ ' from input;
  else  ERROR();
  if ( $M[X, a] == X \rightarrow Y_1 Y_2 \dots Y_k$ )
    then
      {  pop  $X$ ;
        push  $Y_k Y_{k-1} \dots Y_1, Y_1$  on top;
      }
  else ERROR();
}
```

Parse Tree for  $id+id*id$



Stack	Input	Output
$\$E$	$id+id*id\$$	
$\$E'T$	$id+id*id\$$	$E \rightarrow TE'$
$\$E'T'F$	$id+id*id\$$	$T \rightarrow FT'$
$\$E'T'id$	$id+id*id\$$	$F \rightarrow id$
$\$E'T'$	$+id*id\$$	
$\$E'$	$+id*id\$$	$T' \rightarrow \epsilon$
$\$E'T+$	$+id*id\$$	$E' \rightarrow +TE'$
$\$E'T$	$id*id\$$	
$\$E'T'F$	$id*id\$$	$T \rightarrow FT'$
$\$E'T'id$	$id*id\$$	$F \rightarrow id$
$\$E'T'$	$*id\$$	
$\$E'T'F*$	$*id\$$	$T' \rightarrow *FT'$
$\$E'T'F$	$id\$$	
$\$E'T'id$	$id\$$	$F \rightarrow id$
$\$E'T'$	$\$$	
$\$E'$	$\$$	$T' \rightarrow \epsilon$
$\$$	$\$$	$E' \rightarrow \epsilon$

# Error Recovery in Predictive Parsing

## Detecting Error

- An error is detected during predictive parsing when the terminal on top of the stack does not match the next input symbol
- When non terminal  $A$  is on top of the stack,  $a$  is the next input symbol, and  $M[A, a]$  is error (i.e., the parsing-table entry is empty)

## Panic Mode

- Skips symbols on the input until a token from the synchronizing set appears
- Heuristics for the selection of Synchronizing set
  - Place all symbols in  $FOLLOW(A)$  into the synchronizing set for nonterminal  $A$
  - the symbols that begin higher-level constructs can be added to the synchronizing set of a lower level constructs
  - It may be possible to resume parsing by adding  $FIRST(A)$  to the synchronizing set of  $A$
  - Error may be postponed (and not missed) by adding empty productions
- If a terminal on top of the stack cannot be matched, a simple idea is to pop the terminal, issue a message saying that the terminal was inserted, and continue parsing

# Predictive Parsing Table Construction – Example 2

Grammar Productions							FIRST Set			FOLLOW Set			Parsing (a, a)		
$S \rightarrow a / ^ / ( T )$ $T \rightarrow T , S / S$ Removing Left Recursion $S \rightarrow a / ^ / ( T )$ $T \rightarrow S T'$ $T' \rightarrow , S T' / \epsilon$							$S = \{ a, ^, ( \}$ $T = \{ a, ^, ( \}$ $T' = \{ , , \epsilon \}$			$S = \{ \$, ,, ) \}$ $T = \{ ) \}$ $T' = \{ ) \}$			Stack	Input	Output
													\$S	(a,a)\$	
													\$)T(	(a,a)\$	$S \rightarrow (T)$
													\$)T	a,a)\$	
													\$)T'S	a,a)\$	$T \rightarrow ST'$
													\$)T'a	a,a)\$	$S \rightarrow a$
													\$)T'	,a)\$	
													\$)T'S,	,a)\$	$T' \rightarrow ,ST'$
													\$)T'S	a)\$	
													\$)T'a	a)\$	$S \rightarrow a$
													\$)T'	)\$	
													\$)	)\$	
													\$	\$	

$T \rightarrow$ N↓	a	^	(	)	,	\$
S	$S \rightarrow a$	$S \rightarrow ^$	$S \rightarrow (T)$			
T	$T \rightarrow ST'$	$T \rightarrow ST'$	$T \rightarrow ST'$			
T'				$T' \rightarrow \epsilon$	$T' \rightarrow ,ST'$	

# Explaining LL(1) Grammar

## Grammar Productions

$$S \rightarrow iCtS \mid iCtSeS \mid a$$

$$C \rightarrow b$$

Removing Left Factoring

$$S \rightarrow iCtSS' \mid a$$

$$S' \rightarrow \epsilon \mid eS$$

$$C \rightarrow b$$

## FIRST Set

$$S = \{ i, a \}$$

$$S' = \{ \epsilon, e \}$$

$$C = \{ b \}$$

## FOLLOW Set

$$S = \{ \$, e \}$$

$$S' = \{ \$, e \}$$

$$C = \{ t \}$$

## Definition of LL(1) Grammar

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

(OR)

A grammar  $G$  is LL(1) if and only if whenever  $A \rightarrow \alpha \mid \beta$  are two disjoint productions of  $G$ , the following conditions holds.

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

$\begin{matrix} T \rightarrow \\ N \downarrow \end{matrix}$	i	t	e	a	b	\$
S	$S \rightarrow iCtSS'$			$S \rightarrow a$		
S'			$S' \rightarrow eS$ $S' \rightarrow \epsilon$			$S' \rightarrow \epsilon$
C				$T' \rightarrow \epsilon$	$C \rightarrow b$	

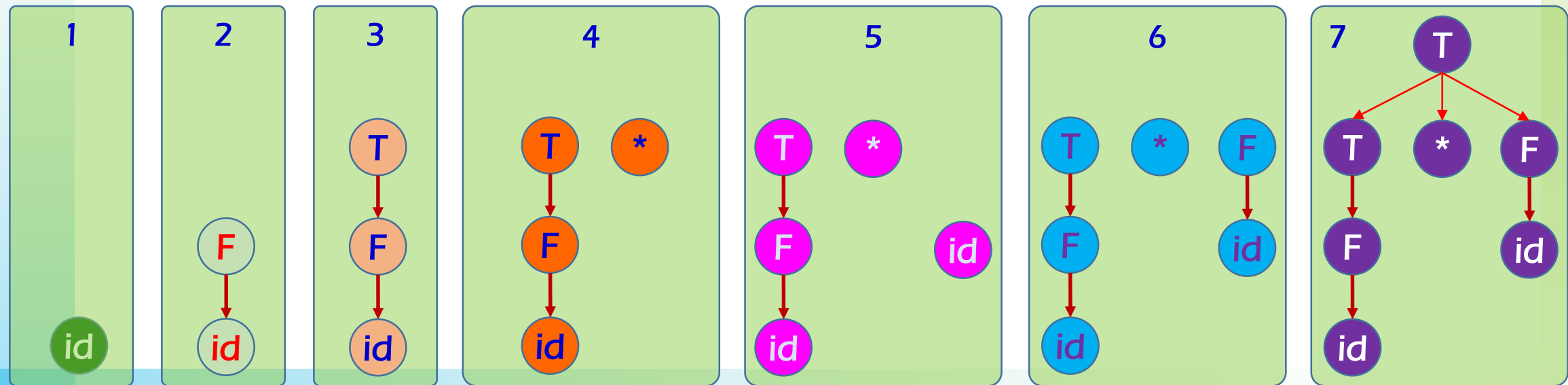


# Bottom up Parsing

**Constructing Parse Tree from Leaves to Root**

# Parse Tree construction in bottom-up

- A bottom-up parse corresponds to the construction of a parse tree for an input string beginning at the leaves (the bottom) and working up towards the root (the top)
- The parse tree construction for the string `id*id` using the grammar  $T \rightarrow T * F \mid F ; F \rightarrow id$  is shown here



# Handle Pruning

- A handle of a right sentential form  $\gamma$  is a production

$$A \rightarrow \beta \text{ and a position of } \gamma$$

where the string  $\beta$  may be found and replaced by  $A$  to produce the previous right sentential form in a rightmost derivation of  $\gamma$

- If  $S \xRightarrow{*} \alpha A w \Rightarrow \alpha \beta w$ , then  $A \rightarrow \beta$  in the position following  $\alpha$  is a handle of  $\alpha \beta w$ ,  $w$  contains only terminals
- The process of reducing the given string  $w$  to the starting symbol of the grammar by locating the handle and replacing it by the left side of the production successively is termed as handle pruning
- The handle pruning gives the reverse of the right most derivation, called Canonical reduction sequence
- For the right most derivation  $S = \gamma_0 \Rightarrow \gamma_1 \Rightarrow \gamma_2 \dots \Rightarrow \gamma_{n-1} \Rightarrow \gamma_n = w$ , the canonical reduction sequence is obtained by locating the handle  $\beta_n$  in  $\gamma_n$  and replace  $\beta_n$  by left side of some production  $A_n \rightarrow \beta_n$  to obtain the  $(n-1)^{\text{th}}$  right sentential form  $\gamma_{n-1}$

# Shift-Reduce Parsing

- The shift-reduce parsing method attempts to construct the parse tree for the string beginning at the leaves (bottom) and working towards the root (top)
- It is equivalent to reducing the given string to the start symbol
- Consider the Grammar  $S \rightarrow aAcBe$        $A \rightarrow Ab \mid b$        $B \rightarrow d$
- Let the string to be parsed be **abbcde**

The parsing of the string is shown below

<b>a</b> bbcde	$A \rightarrow b$
a <b>A</b> bcde	$A \rightarrow Ab$
aAc <b>d</b> e	$B \rightarrow d$
aAc <b>B</b> e	$S \rightarrow aAcBe$
<b>S</b>	

# Stack Implementation of Shift-Reduce Parser

- The data structures needed for implementing shift-reduce parsing is a stack and an input buffer
- \$ denotes bottom of the stack
- The buffer is initialized with the string to be parsed
- The parser operates by shifting zero or more input symbols until a handle  $\beta$  is on top of the stack
- Then  $\beta$  is replaced by left side of the appropriate production
- This process is repeated until it has detected an error or the stack contains only the start symbol and the input buffer is empty
- Actions of the Parser
  - shift : The next input symbol is shifted to the stack from the input buffer
  - reduce : When the parser finds the right end of the handle on the top of the stack, it locates the left end of the handle within the stack and decides the nonterminal to replace the handle
  - accept : The parser announces the successful completion of the parsing
  - error : The parser discovers that a syntax error has occurred and calls an error recovery routine

# Shift-Reduce Parsing for $\text{id} + \text{id} * \text{id}$ using $E \rightarrow E + E \mid E * E \mid (E) \mid \text{id}$

## Parsing Process

Step 1: Constructing the right most derivation for  $\text{id} + \text{id} * \text{id}$

$E$	$\Rightarrow$	$E + E$
$E$	$\Rightarrow$	$E + E * E$
$E$	$\Rightarrow$	$E + E * \text{id}_3$
$E$	$\Rightarrow$	$E + \text{id}_2 * \text{id}_3$
$E$	$\Rightarrow$	$\text{id}_1 + \text{id}_2 * \text{id}_3$

Step 2: Handle Pruning (Canonical Reduction sequence Construction)

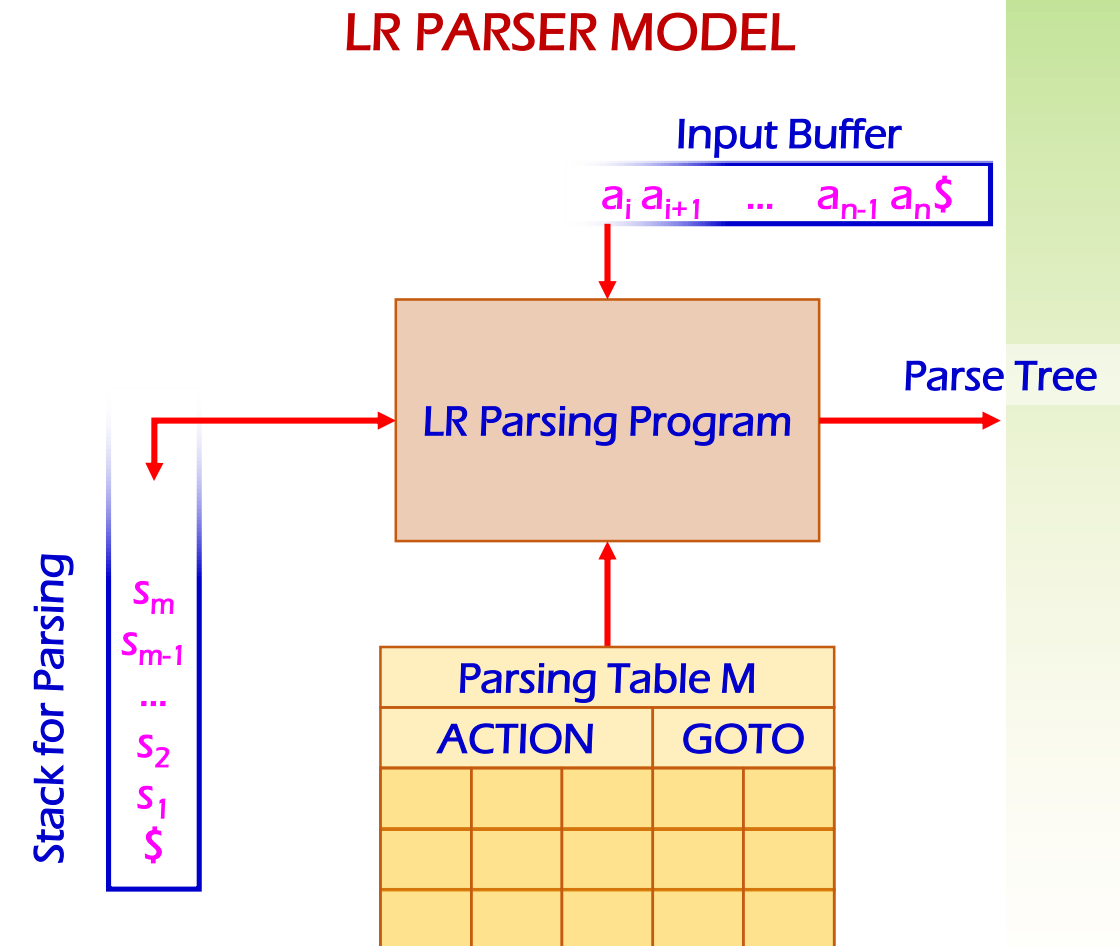
<u>Right sentential form</u>	<u>Handle</u>	<u>Reducing Production</u>
$\text{id}_1 + \text{id}_2 * \text{id}_3$	$\text{id}_1$	$E \rightarrow \text{id}$
$E + \text{id}_2 * \text{id}_3$	$\text{id}_2$	$E \rightarrow \text{id}$
$E + E * \text{id}_3$	$\text{id}_3$	$E \rightarrow \text{id}$
$E + E * E$	$E * E$	$E \rightarrow E * E$
$E + E$	$E + E$	$E \rightarrow E + E$
$E$		

## Step 3: Stack Implementation for $\text{id} + \text{id} * \text{id}$

<u>Stack</u>	<u>Input</u>	<u>Action</u>
\$	$\text{id}_1 + \text{id}_2 * \text{id}_3 \$$	shift
$\$ \text{id}_1$	$+ \text{id}_2 * \text{id}_3 \$$	reduce by $E \rightarrow \text{id}$
$\$ E$	$+ \text{id}_2 * \text{id}_3 \$$	shift
$\$ E +$	$\text{id}_2 * \text{id}_3 \$$	shift
$\$ E + \text{id}_2$	$* \text{id}_3 \$$	reduce by $E \rightarrow \text{id}$
$\$ E + E$	$* \text{id}_3 \$$	shift
$\$ E + E *$	$\text{id}_3 \$$	shift
$\$ E + E * \text{id}_3$	$\$$	reduce by $E \rightarrow \text{id}$
$\$ E + E * E$	$\$$	reduce by $E \rightarrow E * E$
$\$ E + E$	$\$$	reduce by $E \rightarrow E + E$
$\$ E$	$\$$	accept

# LR Parsers

- LR parsers are the non-backtracking bottom up parsers scanning the input from **L**eft to right and construct the **R**ight most derivation in reverse
- LR parsers can be constructed to recognize virtually all programming language constructs for which context free grammars can be written
- The parser consists of a driver routine and a parsing table
- The principal drawback of the LR method is that it is too much work to construct an LR parser
- The techniques used for producing LR parsing table are
  - Simple LR**, which is the simple method but it may fails to produce parsing table for certain grammar.
  - Canonical LR**, which is the most powerful and work for large class of grammars but very expensive to implement.
  - Look-ahead LR**, having intermediate power in between SLR and Canonical LR methods.



# LR(0) Items

- An LR(0) item of a grammar  $G$  is a production of  $G$  with a dot at some position of the body and  $A \rightarrow XYZ$  yields the four items

$A \rightarrow \cdot XYZ$      $A \rightarrow X \cdot YZ$      $A \rightarrow XY \cdot Z$      $A \rightarrow XYZ \cdot$

- $A \rightarrow \epsilon$  has only one item of the form  $A \rightarrow \cdot$
- An item indicates how much of a production we have seen at a given point in the parsing process
- One collection of sets of LR(0) items is termed as **canonical LR(0) collection** and it is represented by an LR(0) automaton
- Canonical LR(0) collection for a grammar  $G$  is generated with an augmented grammar  $G'$  and two functions namely **CLOSURE** and **GOTO**
- If  $G$  is a grammar with start symbol  $S$ , then, the **augmented grammar  $G'$**  has a new start symbol  $S'$  and production  $S' \rightarrow S$
- The purpose of this new starting production is to indicate to the parser when it should stop parsing and announce acceptance of the input
- The parser announce acceptance when and only when it is about to reduce by  $S' \rightarrow S$

procedure ITEMS (  $G'$  )

begin

$C := \{ \text{CLOSURE} ( \{ S' \rightarrow \cdot S \} ) \}$

  Repeat

    For each set of items  $I$  in  $C$  and each grammar symbol  $X$  such that  $\text{GOTO} ( I, X )$  is not empty and not in  $C$  do

      Add  $\text{GOTO} ( I, X )$  to  $C$ .

  Until no more items can be added to  $C$ ;

end

procedure CLOSURE (  $I$  )

begin

  Repeat

    For each item  $A \rightarrow \alpha \cdot B \beta$  in  $I$  and each production  $B \rightarrow \gamma$  in  $G$  such that  $B \rightarrow \cdot \gamma$  is not in  $I$  do

      Add  $B \rightarrow \cdot \gamma$  to  $I$ ;

  Until no more items can be added to  $I$ ;

  Return (  $I$  );

end

procedure GOTO (  $I, X$  )

begin

  If  $A \rightarrow \alpha \cdot X \beta$  is in state  $I$  then

    return (  $\text{CLOSURE} ( \{ A \rightarrow \alpha X \cdot \beta \} )$  );

end



# LR(0) ITEMS

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

The augmented grammar for the given grammar is

0)  $E' \rightarrow E$

1)  $E \rightarrow E + T$

2)  $E \rightarrow T$

3)  $T \rightarrow T * F$

4)  $T \rightarrow F$

5)  $F \rightarrow ( E )$

6)  $F \rightarrow id$

The initial state  $I_0$  is obtained by

$CLOSURE \{ ( E' \rightarrow .E ) \}$ .

$CLOSURE \{ ( E' \rightarrow .E ) \} = I_0$

$E' \rightarrow .E$

$E \rightarrow .E + T$

$E \rightarrow .T$

$T \rightarrow .T * F$

$T \rightarrow .F$

$F \rightarrow .( E )$

$F \rightarrow .id$

$GOTO(I_0, E) = I_1$

$E' \rightarrow E.$

$E \rightarrow E.+T$

$GOTO(I_0, T) = I_2$

$E \rightarrow T.$

$T \rightarrow T.*F$

$GOTO(I_0, F) = I_3$

$T \rightarrow F.$

$GOTO(I_0, ( ) = I_4$

$F \rightarrow (.E )$

$E \rightarrow .E + T$

$E \rightarrow .T$

$T \rightarrow .T * F$

$T \rightarrow .F$

$F \rightarrow .( E )$

$F \rightarrow .id$

$GOTO(I_0, id) = I_5$

$F \rightarrow id.$

$GOTO(I_1, +) = I_6$

$E \rightarrow E.+T$

$T \rightarrow .T * F$

$T \rightarrow .F$

$F \rightarrow .( E )$

$F \rightarrow .id$

$GOTO(I_2, *) = I_7$

$T \rightarrow T*.F$

$F \rightarrow .( E )$

$F \rightarrow .id$

$GOTO(I_4, E) = I_8$

$F \rightarrow (E.)$

$E \rightarrow E.+T$

$GOTO(I_4, T) = I_2$

$GOTO(I_4, F) = I_3$

$GOTO(I_4, ( ) = I_4$

$GOTO(I_4, id) = I_5$

$GOTO(I_6, T) = I_9$

$E \rightarrow E.+T.$

$T \rightarrow T.*F$

$GOTO(I_6, F) = I_3$

$GOTO(I_6, ( ) = I_4$

$GOTO(I_6, id) = I_5$

$GOTO(I_7, F) = I_{10}$

$T \rightarrow T * F.$

$GOTO(I_7, ( ) = I_4$

$GOTO(I_7, id) = I_5$

$GOTO(I_8, ) = I_{11}$

$F \rightarrow ( E ).$

$GOTO(I_8, +) = I_6$

$GOTO(I_9, *) = I_7$



# SLR Parsing table Construction algorithm

Let  $C = \{ I_0, I_1, I_2, \dots, I_n \}$ , the parsing actions for the state  $i$  are defined as follows.

1. If  $A \rightarrow \alpha.a\beta$  is in  $I_i$ ,  $\text{GOTO}(I_i, a) = I_j$  then  $\text{ACTION}[i, a] = \text{"shift } j\text{"}$
2. If  $A \rightarrow \alpha.$  is in  $I_i$  then  $\text{ACTION}[i, a] = \text{"reduce } A \rightarrow \alpha\text{"}$   $\forall a$  in  $\text{FOLLOW}(A)$
3. If  $S' \rightarrow S.$  is in  $I_i$  then  $\text{ACTION}[i, \$] = \text{accept}$
4. If  $\text{GOTO}(I_i, A) = I_j$  then  $\text{GOTO}(i, A) = j$
5. Make all the undefined entries as Error
6. The initial state of the parser is the state constructed from the item  $[S' \rightarrow .S]$

	ACTION						GOTO		
	+	*	(	)	id	\$	E	T	F
0			s4		s5		1	2	3
1	s6					accept			
2	r2	s7		r2		r2			
3	r4	r4		r4		r4			
4			s4		s5		8	2	3
5	r6	r6		r6		r6			
6			s4		s5			9	3
7			s4		s5				10
8	s6			s11					
9	r1	s7		r1		r1			
10	r3	r3		r3		r3			
11	r5	r5		r5		r5			

# Parsing with SLR Parsing Table

## LR Parser Configuration (State of the Parser)

- A configuration of an LR parser is a pair, where the first component is the stack content, and the second component is the remaining input

$$(s_0 s_1 s_2 \dots s_{m-1} s_m, a_i a_{i+1} a_{i+2} \dots a_n \$)$$

- This configuration represents the following right-sentential form, where  $X_i$  is the grammar symbol represented by state  $s_i$

$$X_1 X_2 \dots X_{m-1} X_m a_i a_{i+1} \dots a_n$$

- Let  $a_i$  be the current input symbol, and  $s_m$  be the state on top of the stack
- Based on the entry  $\text{ACTION}[s_m, a_i]$ , the configuration is continuously modified until accept or error is reached

## LR Parsing Method

- If  $\text{ACTION}[s_m, a_i] = \text{shift } s$ , then the parser executes a shift action and enters into the configuration

$$(s_0 s_1 s_2 \dots s_{m-1} s_m s, a_{i+1} a_{i+2} \dots a_n)$$

The top state is  $s$  and the current input symbol  $a_{i+1}$

- If  $\text{ACTION}[s_m, a_i] = \text{reduce } A \rightarrow \beta$ , then the parser executes a reduce action and enters into the configuration

$$(s_0 s_1 s_2 \dots s_{m-r} s, a_i a_{i+1} a_{i+2} \dots a_n \$)$$

where  $s = \text{GOTO}[s_{m-r}, A]$  and  $r$  is the length of  $\beta$

The parser pops  $r$  number of symbols from the stack, and push  $s$  onto the stack, where  $s = \text{GOTO}[s_{m-r}, A]$

- If  $\text{ACTION}[s_m, a_i] = \text{accept}$ , then parsing is completed.
- If  $\text{ACTION}[s_m, a_i] = \text{error}$ , then the parser has discovered an error and calls an error recovery routine

# SLR Parsing Table Construction for $S \rightarrow AS|b$   $A \rightarrow SA|a$

	ACTION			GOTO	
	a	b	\$	S	A
0					
1					
2					
3					
4					
5					
6					
7					
8					
9					
10					
11					
12					

## Drawback of SLR Parser

- If the  $i^{\text{th}}$  state contains  $A \rightarrow \alpha.$  and 'a' is in FOLLOW (A) then the  $i^{\text{th}}$  state calls for a reduction by  $A \rightarrow \alpha$  for the input symbol 'a'
- When i appears on the top of the stack, a viable prefix  $\beta\alpha$  may be on the stack, such that  $\beta A$  cannot be followed by a in the right sentential form
- For example, the Grammar  
 $S \rightarrow L=R; S \rightarrow R; L \rightarrow *R; L \rightarrow \text{id}; R \rightarrow L$   
 has the second state with the items  
 $S \rightarrow L.=R$   
 $R \rightarrow L.$  FOLLOW (R) has '='  
 Since FOLLOW (R) has '=', it calls for a reduction by  
 $R \rightarrow L$   
 which creates a right sentential form beginning with  
 $R=$
- Even though the state 2 is corresponding to the viable prefix L, it should not call for a reduction
- Now it is required to have more information in the state 2 that will rule out the possibility of reduction
- For that, each state of the LR parser has an input symbol, and only if that input symbol follows the handle a then perform the reduction
- The extra information is incorporated in the state by redefining the items to include a terminal as the second component
- The general form of those items is  $A \rightarrow \alpha . \beta, a$  where  $A \rightarrow \alpha\beta$  is the production and 'a' is a terminal or \$.
- Items of the form  $A \rightarrow \alpha . \beta, a$  is called LR (1) items; the 1 refers to the length of the look ahead
- This look ahead has no effect if the item is of the form  $A \rightarrow \alpha . \beta, a$  where  $\beta \neq \epsilon$ , but an item of the form  $A \rightarrow \alpha. , a$  calls for reduction by  $A \rightarrow \alpha$  only if the next input symbol is 'a'

LR(0) Items for

$S \rightarrow L=R; S \rightarrow R; L \rightarrow *R; L \rightarrow id; R \rightarrow L$

# LR(1) Items

```

procedure ITEMS ( G' )
begin
  C := { CLOSURE ( { S' → .S, $ } ) }
  Repeat
    For each set of items I in C and each grammar symbol X such that GOTO ( I, X ) is not empty and not in C do
      Add GOTO ( I, X ) to C.
  Until no more items can be added to C;
End

procedure CLOSURE ( I )
begin
  Repeat
    For each item A → α . B β, a in I, each production B → γ and for each terminal 'b' in FIRST(βa) such that B → . γ, b is
    not in I do
      Add B → . γ, b to I;
  Until no more items can be added to I;
  return ( I );
end

procedure GOTO ( I, X )
begin
  If A → α . X β, a is in state I then return ( CLOSURE ( { A → α X . β, a } ) );
end

```



# LR(1) Items for the CFG $G = (\{S, C\}, \{c, d\}, \{S \rightarrow CC, C \rightarrow cC, C \rightarrow d\}, S)$

The Augmented grammar  $G'$

- (0)  $S' \rightarrow S$
- (1)  $S \rightarrow CC$
- (2)  $C \rightarrow cC$
- (3)  $C \rightarrow d$

$CLOSURE(\{S' \rightarrow .S, \$\}) = I_0$

- $S' \rightarrow .S, \$$
- $S \rightarrow .CC, \$$
- $C \rightarrow .cC, c/d$
- $C \rightarrow .d, c/d$

$GOTO(I_0, S) = I_1$

$S' \rightarrow S., \$$

$GOTO(I_0, C) = I_2$

- $S \rightarrow C.C, \$$
- $C \rightarrow .cC, \$$
- $C \rightarrow .d, \$$

$GOTO(I_0, c) = I_3$

- $C \rightarrow c.C, c/d$
- $C \rightarrow .cC, c/d$
- $C \rightarrow .d, c/d$

$GOTO(I_0, d) = I_4$

$C \rightarrow d., c/d$

$GOTO(I_2, C) = I_5$

$S \rightarrow CC., \$$

$GOTO(I_2, c) = I_6$

- $C \rightarrow c.C, \$$
- $C \rightarrow .cC, \$$
- $C \rightarrow .d, \$$

$GOTO(I_2, d) = I_7$

$C \rightarrow d., \$$

$GOTO(I_3, C) = I_8$

$C \rightarrow cC., c/d$

$GOTO(I_3, c) = I_3$

$GOTO(I_3, d) = I_{40}$

$GOTO(I_6, C) = I_9$

$C \rightarrow cC., \$$

$GOTO(I_6, c) = I_6$

$GOTO(I_6, d) = I_7$

	ACTION			GOTO	
	c	d	\$	S	C
0					
1					
2					
3					
4					
5					
6					
7					
8					
9					

Given LR(1) Items  $C = \{I_0, I_1, I_2, \dots, I_n\}$  for the CFG  $G'$ , the parsing table is created using the following rules

1. If  $A \rightarrow \alpha.a\beta$ ,  $b$  is in  $I_i$ ,  $GOTO(I_i, a) = I_j$  then  $ACTION[i, a] = \text{"shift } j\text{"}$
2. If  $A \rightarrow \alpha.$ ,  $b$  is in  $I_i$  then  $ACTION[i, b] = \text{"reduce } A \rightarrow \alpha\text{"}$
3. If  $S' \rightarrow S., \$$  is in  $I_i$  then  $ACTION[i, \$] = \text{accept}$
4. If  $GOTO(I_i, A) = I_j$  then  $GOTO(i, A) = j$
5. Make all the undefined entries as Error
6. The initial state of the parser is the state constructed from the item  $[S' \rightarrow .S, \$]$

# Constructing LALR Parsing Table

- The LALR parsing table is very smaller than the canonical LR table
- For that it is required to identify the sets of LR(1) items having the same core and merging these sets with common cores into one set of items
- The ACTION functions are modified to reflect the non-error actions of all sets of items in the merged state
- For all LR(1) Grammars, (Grammar whose LR(1) sets of items do not produce parsing action conflict) the union of states will never produce any parsing action conflict
- The LALR parser behaves similar to the canonical LR parser
- When we have an erroneous input string to be parsed, the LALR parser declares error after doing some unnecessary reductions, while the canonical LR parser declares error immediately
- Both the parsers never shifts invalid symbols from the buffer to the stack

## Algorithm for constructing LALR Parsing Table

- Construct  $C := \{ I_0, I_1, \dots, I_n \}$ , the collection of sets of LR(1) items
- For each core present among the sets of LR(1) items, find all sets having the same core and replace these sets by their unions
- Let  $C' := \{ J_0, J_1, J_2, \dots, J_m \}$  be the resulting sets of LR(1) items
- The parsing actions for each state 'i' are constructed from the set of items  $J_i$  using the canonical LR parsing table construction algorithm
- If  $J$  is the union of  $I_1, I_2, \dots, I_m$ , then the cores of  $\text{GOTO}(I_1, X)$ ,  $\text{GOTO}(I_2, X)$ , ...  $\text{GOTO}(I_m, X)$  are all same. Let  $K$  be the set of all items having the same core as  $\text{GOTO}(I_1, X)$ . Then  $\text{GOTO}(J, X) = K$ .
- The table produced by this algorithm is called LALR parsing table for  $G$
- If there is no parsing action conflict, then the given grammar is said to be LALR(1) grammar

## LALR(1) Items for the CFG

 $G = (\{S, C\}, \{c, d\}, \{S \rightarrow CC, C \rightarrow cC, C \rightarrow d\}, S)$ 
The Augmented grammar  $G'$ (0)  $S' \rightarrow S$ (1)  $S \rightarrow CC$ (2)  $C \rightarrow cC$ (3)  $C \rightarrow d$  $CLOSURE(\{S' \rightarrow .S, \$\}) = I_0$  $S' \rightarrow .S, \$$  $S \rightarrow .CC, \$$  $C \rightarrow .cC, c/d$  $C \rightarrow .d, c/d$  $GOTO(I_0, S) = I_1$  $S' \rightarrow S., \$$  $GOTO(I_0, C) = I_2$  $S \rightarrow C.C, \$$  $C \rightarrow .cC, \$$  $C \rightarrow .d, \$$  $GOTO(I_0, c) = I_3$  $C \rightarrow c.C, c/d$  $C \rightarrow .cC, c/d$  $C \rightarrow .d, c/d$  $GOTO(I_0, d) = I_4$  $C \rightarrow d., c/d$  $GOTO(I_2, C) = I_5$  $S \rightarrow CC., \$$  $GOTO(I_2, c) = I_6$  $C \rightarrow c.C, \$$  $C \rightarrow .cC, \$$  $C \rightarrow .d, \$$  $GOTO(I_2, d) = I_7$  $C \rightarrow d., \$$  $GOTO(I_3, C) = I_8$  $C \rightarrow cC., c/d$  $GOTO(I_3, c) = I_3$  $GOTO(I_3, d) = I_4$  $GOTO(I_6, C) = I_9$  $C \rightarrow cC., \$$  $GOTO(I_6, c) = I_6$  $GOTO(I_6, d) = I_7$  $GOTO(I_0, c) = I_{36}$  $C \rightarrow c.C, c/d/\$$  $C \rightarrow .cC, c/d/\$$  $C \rightarrow .d, c/d/\$$  $GOTO(I_0, d) = I_{47}$  $C \rightarrow d., c/d/\$$  $GOTO(I_3, C) = I_{89}$  $C \rightarrow cC., c/d/\$$ 

	ACTION			GOTO	
	c	d	\$	S	C
0	s36	s47		1	2
1					
2	s36	s47			5
36	s36	s47			89
47	r3	r3	r3		
5			r1		
89	r2	r2	r2		

# Using ambiguous grammar for parsing table construction

- Every ambiguous grammar fails to be LR
- Ambiguous grammar for expressions provides a shorter, more natural specification than any equivalent unambiguous grammar
- The equivalent unambiguous grammars do more work, such as using unit productions and takes more parsing time
- These ambiguous grammars are provided with disambiguating rules that allow only one parse tree for each sentence
- The parsing action conflicts in the expression grammar can be resolved with the help of precedence and associativity information for the operators

# Handling ambiguous expression grammar

$E' \rightarrow E$   $I_4 = \text{GOTO}(I_1, *)$   
 $E \rightarrow E + E$   $E \rightarrow E^*.E$   
 $E \rightarrow E * E$   $E \rightarrow .E + E$   
 $E \rightarrow \text{id}$   $E \rightarrow .E * E$   
 $I_0 = \text{Closure}(\{E' \rightarrow .E\})$   $E \rightarrow .\text{id}$   
 $E' \rightarrow .E$   $I_5 = \text{GOTO}(I_3, E)$   
 $E \rightarrow .E + E$   $E \rightarrow E + E.$   
 $E \rightarrow .E * E$   $E \rightarrow E. + E$   
 $E \rightarrow .\text{id}$   $E \rightarrow E. * E$   
 $I_1 = \text{GOTO}(I_0, E)$   $I_2 = \text{GOTO}(I_3, \text{id})$   
 $E' \rightarrow E.$   $I_6 = \text{GOTO}(I_4, E)$   
 $E \rightarrow E. + E$   $E \rightarrow E^*.E.$   
 $E \rightarrow E. * E$   $E \rightarrow E. + E$   
 $I_2 = \text{GOTO}(I_0, \text{id})$   $E \rightarrow E. * E$   
 $E \rightarrow \text{id}.$   $I_2 = \text{GOTO}(I_4, \text{id})$   
 $I_3 = \text{GOTO}(I_1, +)$   $I_3 = \text{GOTO}(I_5, +)$   
 $E \rightarrow E + .E$   $I_4 = \text{GOTO}(I_5, *)$   
 $E \rightarrow .E + E$   $I_3 = \text{GOTO}(I_6, +)$   
 $E \rightarrow .E * E$   $I_4 = \text{GOTO}(I_6, *)$   
 $E \rightarrow .\text{id}$

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

	ACTION				GOTO
	+	*	id	\$	E
0			s2		1
1	s3	s4		acc	
2	r3	r3		r3	
3			s2		5
4			s2		6
5	S3/r1	S4/r1		r1	
	r1	s4			
6	S3/r2	S4/r2		r2	
	r2	r2			

# Handling dangling-else grammar

$S' \rightarrow S$   
 $S \rightarrow iSeS$   
 $S \rightarrow iS$   
 $S \rightarrow a$   
 $I_0 = \text{Closure}(\{S' \rightarrow .S\})$   
 $S' \rightarrow .S$   
 $S \rightarrow .iSeS$   
 $S \rightarrow .iS$   
 $S \rightarrow .a$   
 $I_1 = \text{GOTO}(I_0, S)$   
 $S' \rightarrow S.$   
 $I_2 = \text{GOTO}(I_0, i)$   
 $S \rightarrow i.SeS$   
 $S \rightarrow i.S$   
 $S \rightarrow .iSeS$   
 $S \rightarrow .iS$   
 $S \rightarrow .a$   
 $I_3 = \text{GOTO}(I_0, a)$   
 $S \rightarrow a.$   
 $I_4 = \text{GOTO}(I_2, S)$   
 $S \rightarrow iS.eS$   
 $S \rightarrow iS.$   
 $I_2 = \text{GOTO}(I_2, i)$   
 $I_3 = \text{GOTO}(I_2, a)$   
 $I_5 = \text{GOTO}(I_4, e)$   
 $S \rightarrow iSe.S$   
 $S \rightarrow .iSeS$   
 $S \rightarrow .iS$   
 $S \rightarrow .a$   
 $I_6 = \text{GOTO}(I_5, S)$   
 $S \rightarrow iSeS.$   
 $I_2 = \text{GOTO}(I_5, i)$   
 $I_3 = \text{GOTO}(I_5, a)$   
 $\text{FOLLOW}(S) = \{\$, e\}$

	ACTION				GOTO
	i	e	a	\$	S
0	s2		s3		1
1				Acc	
2	s2		s3		4
3		r3		r3	
4		S5/r2 s5		r2	
5	s2		s3		6
6		r1		r1	