SYNTAX ANALYSIS

UNIT-II

Context Free Grammar

- Grammar
  - It is a set of rules which checks whether a string belongs to a particular language or not.
  - To identify valid strings in a language, some rules should be specified to check whether string is valid or not.
  - These rules make a grammar

- Context Free Grammar
  - Notation used to specify the syntax of language
  - CFG's are used to design parsers.

Context Free Grammar

- It can be defined as 4-Tuple (V,∑, P, S)
  - Where,
  - V is a set of Non-terminals or Variables
  - ∑ is a set of terminals.
  - P is a set of Productions or set of rules.
  - S is a starting symbol.

- G is context free if every production (P) is of form

  A-> α, where **A ∈ V** and **α ∈ (V ∪ ∑)***

Derivations

- Replacing a given string's non-terminal by right hand side of production rule.
- Terminal strings are derived, beginning with start symbol, by repeatedly replacing a variable by some production.

- Derivations are denoted by:
  - Leftmost derivation
    - We apply production only to the leftmost variable at every step.
  - Rightmost derivation
    - We apply production only to the rightmost variable at every step.

Parse Trees (Derivation Trees)

- A Parse tree for a CFG G= (V,Σ, P, S) is a tree satisfying following conditions:
  - Root has label S
  - Every vertex has label which can be a variable (V), terminal (Σ) or ε.
  - If A-> C_1, C_2, ……… C_n is a production, then C_1, C_2, ……, C_n are children of node labeled A.
  - Leaf nodes are Terminals & Interior nodes are Variables or Non-terminals.

- Yield
  - Yield of derivation tree is the concatenation of the labels of the leaves in left to right ordering.

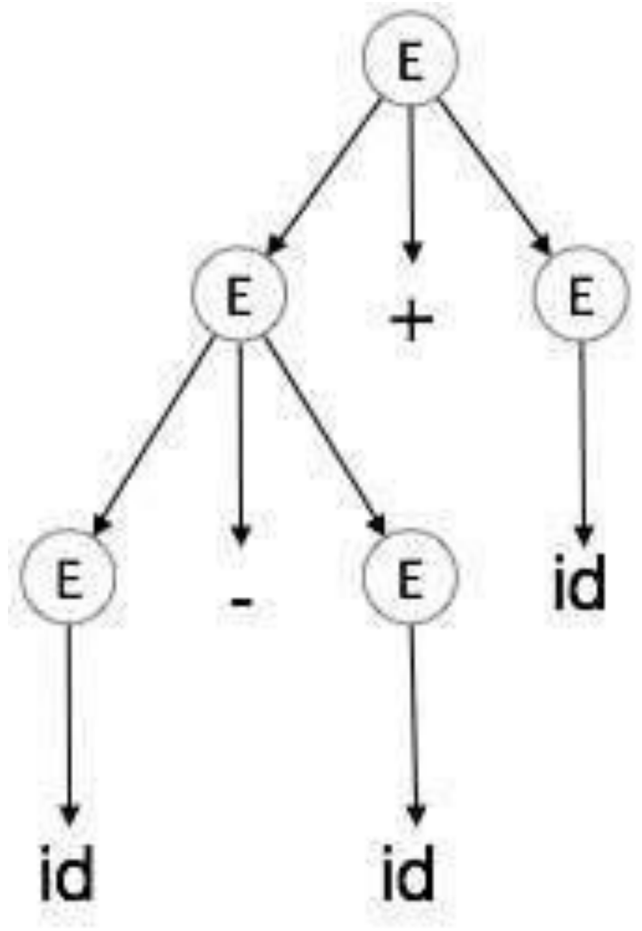
Ambiguity

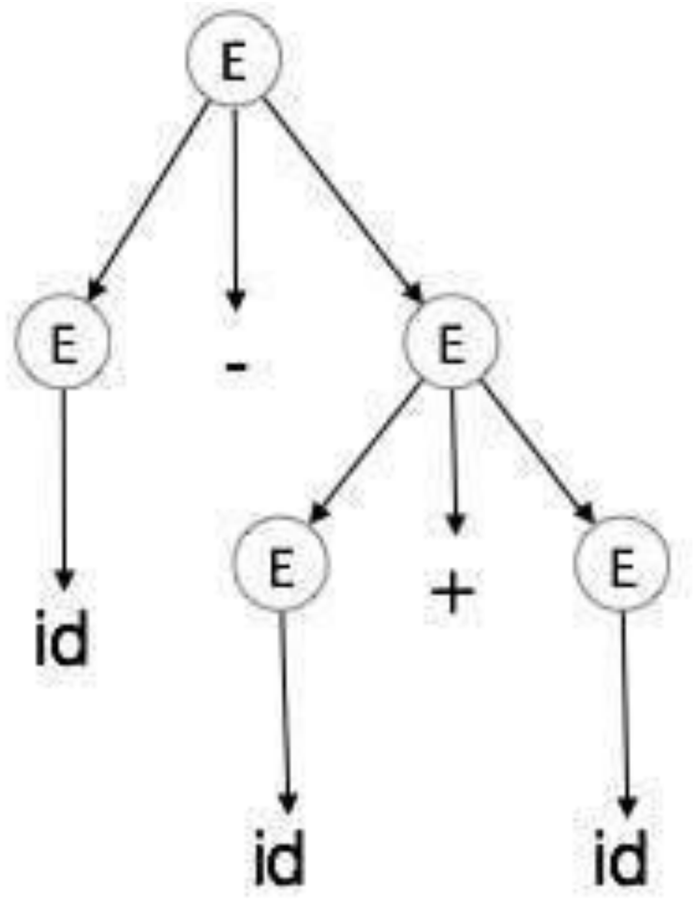
- A CGF G is said to be ambiguous if there exists more than one derivation tree for a given string.
- In other words, A grammar that produces more than one Leftmost or Rightmost derivation for the same sentence is called Ambiguous Grammar.

- E.g.
- Given a Grammar $E \to E+E \mid E-E \mid id$

  verify whether the grammar is Ambiguous or not for string id-id+id

Leftmost and Rightmost Derivation





Conversion to Unambiguous Grammar

- The productions containing more than one occurrence of a given non-terminal on its right hand side are ambiguous productions.
- E.g. E->E*E
- This can be converted to unambiguous grammar by introducing a new Non-terminal to move farther down to parse tree.

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

  E -> E+T | T

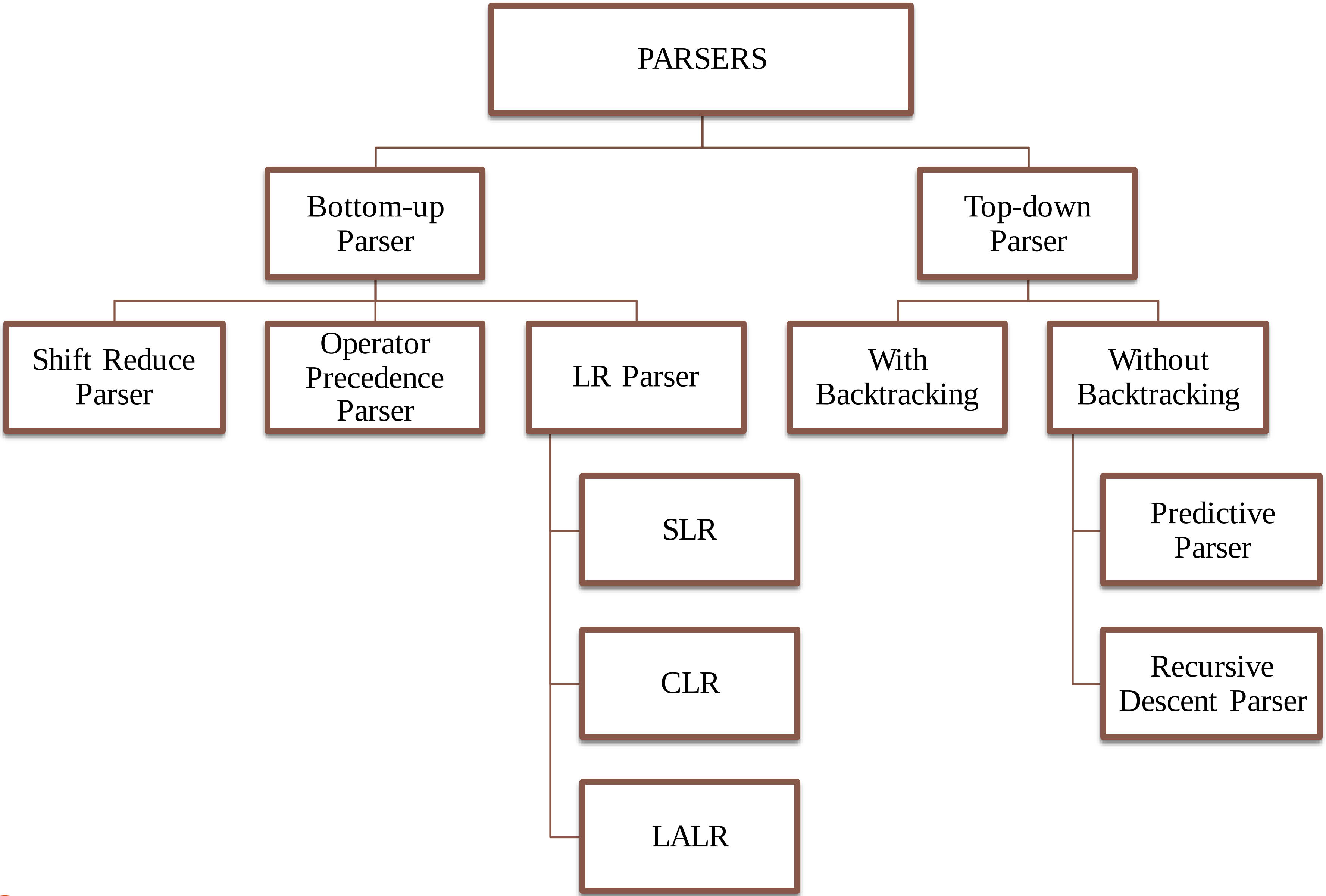
  T->T*F | F

  F-> (E) | id

Parsers

- Syntax analysis or Parsing is the second phase of Compilation.
- It takes inputs as tokens & convert them into Parse Tree.

- Types of Parser:
  - Bottom-up Parser
  - Top-down Parser


PARSERS
Bottom-up Parser
Top-down Parser
Shift Reduce Parser
Operator Precedence Parser
LR Parser
With Backtracking
Without Backtracking
SLR
CLR
LALR
Predictive Parser
Recursive Descent Parser

Shift Reduce Parsing

- It is a type of bottom up parser.
- Generates the Parse Tree from leaves to the root.
- The input string is reduced to the starting symbol.
- It can be achieved by applying rightmost derivation in reverse (i.e. from starting symbol to input string).

- Shift Reduce parser requires two data structures:
  - Input Buffer
  - Stack

- **Reduction:** Each replacement of right side of production by left side is called **reduction**
- **Handle:** Each replacement is called **Handle**.

Stack Implementation of Shift-Reduce Parsing

1. Use a stack and an input buffer.
2. Insert \$ at the bottom of stack and right end of input string in Input Buffer.
3. Shift: Parser shift zero or more input symbols onto stack until handle is on top of the stack.
4. Reduce: Parser reduces or replace handle on the top of stack to left side of production (R.H.S. is popped & L.H.S. is pushed).
5. Accept: Steps 3 & 4 will be repeated until it has detected an error or until stack contains start symbol (S) and input buffer is empty i.e. it contains \$.

Example

- Consider the grammer:

  E-> E + E

  E-> E * E

  E-> (E)

  E-> id

Perform shift reduce parsing for the string

$id_1+id_2*id_3$

STACK	INPUT STRING	ACTION
\$	$id_1 + id_2 * id_3$ \$	Shift
\$ id_1	$+ id_2 * id_3$ \$	Reduce by E -> id
\$ E	$+ id_2 * id_3$ \$	Shift
\$ E +	$id_2 * id_3$ \$	Shift
\$ E + id_2	$* id_3$ \$	Reduce by E -> id
\$ E + E	$* id_3$ \$	Shift
\$ E + E *	id_3 \$	Shift
\$ E + E * id_3	\$	Reduce by E -> id
\$ E + E * E	\$	Reduce by E -> E * E
\$ E + E	\$	Reduce by E -> E + E
\$ E	\$	**ACCEPT**

Example

- Consider the grammer:

  S -> CC

  C -> cC

  C -> d

Check whether the string "ccdd" is accepted or not using Shift Reduce Parsing.

STACK	INPUT STRING	ACTION
$	ccdd $	Shift
$c	cdd $	Shift
$cc	dd $	Shift
$ccd	d $	Reduce by C -> d
$ccC	d $	Reduce by C -> cC
$cC	d $	Reduce by C -> cC
$C	d $	Shift
$Cd	$	Reduce by C -> d
$CC	$	Reduce by S -> cC
$S	$	**ACCEPT**

Drawbacks of Shift Reduce Parsing

- **Shift | Reduce Conflict**
  - Sometimes it is difficult for the parser to decide whether to shift or to reduce.
- **Reduce | Reduce Conflict**
  - Sometimes, parser cannot decide which of the productions should be used for reduction.

Operator Precedence Parsing

- It is a type of bottom-up parsing which can be applied to operator grammar.
- Operator Grammar
  - A grammar G is said to be Operator Grammar if it has following properties:
    - Production should not contain ε on its right side.
    - There should not be two adjacent non-terminals at the right side of production.
- Operator Precedence Relations
  - There are three precedence relations which exist between pair of terminals.

Relation	Meaning
$p \lessdot q$	P has less precedence than q
$p \gtrdot q$	P has more precedence than q
$p \doteq q$	P has equal precedence as q

Association or precedence rules

- Operators having different precedence
- If operators have equal precedence then use association rules.
- Identifier has more precedence than all other operators and symbols
- $ has less precedence than all other operators and symbols.
- Example:

Operators	Precedence	Association
^	Highest	Right Associative
* and /	Next Highest	Left Associative
+ and -	Lowest	Left Associative

Example

Given a grammar E-> E+E | E*E | id

	id	+	*	$
id		$\gtrdot$	$\gtrdot$	$\gtrdot$
+	$\lessdot$	$\gtrdot$	$\lessdot$	$\gtrdot$
*	$\lessdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$
$	$\lessdot$	$\lessdot$	$\lessdot$	

Example

Construct a precedence relation table for the grammar given below:

E-> E + E

E-> E - E

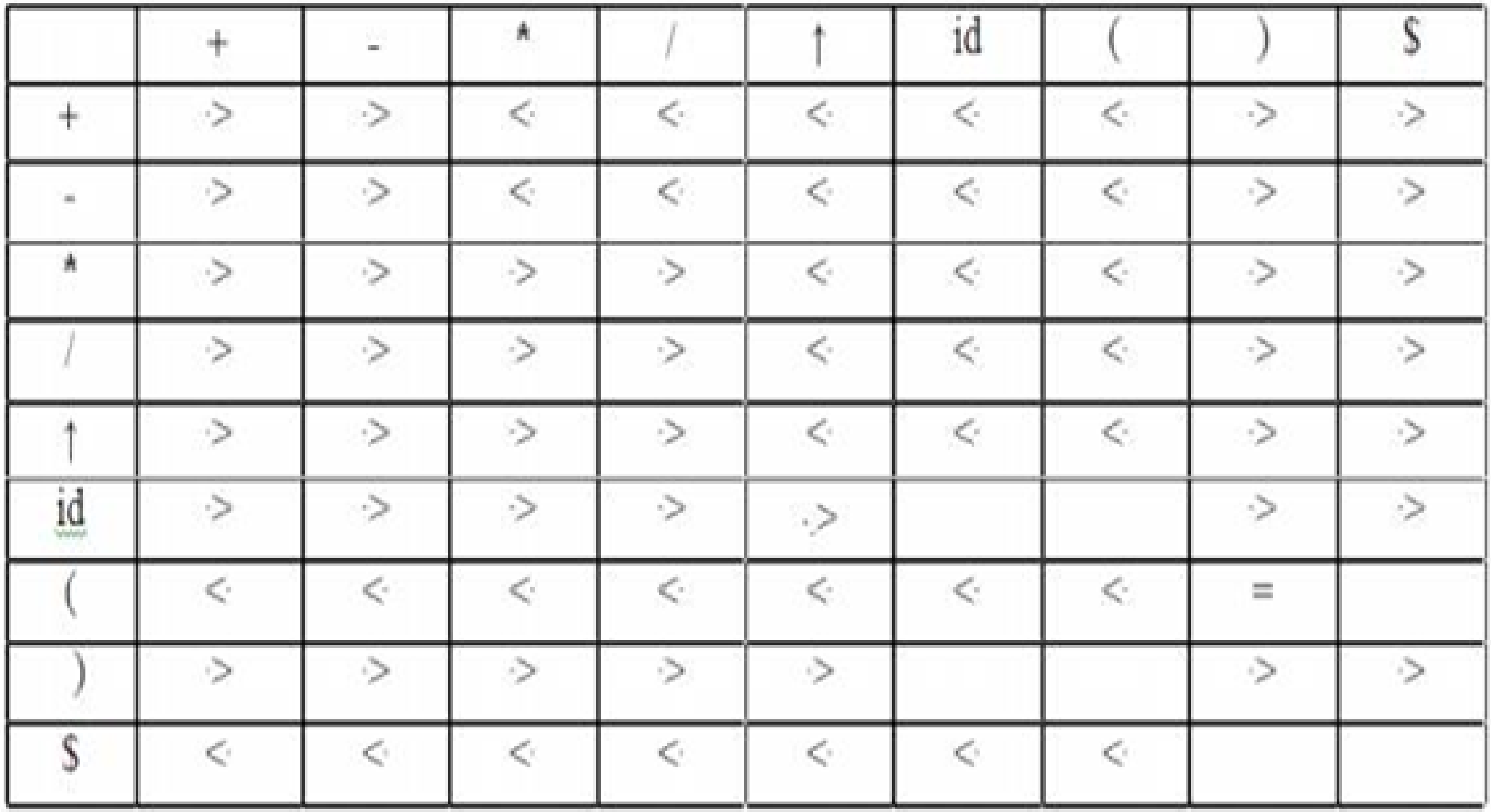
E-> E * E

E-> E / E

E-> E ^ E

E-> (E)

E-> id



	+	-	*	/	↑	id	(	)	$
+	·>	·>	<·	<·	<·	<·	<·	·>	·>
-	·>	·>	<·	<·	<·	<·	<·	·>	·>
*	·>	·>	·>	·>	<·	<·	<·	·>	·>
/	·>	·>	·>	·>	<·	<·	<·	·>	·>
↑	·>	·>	·>	·>	<·	<·	<·	·>	·>
id	·>	·>	·>	·>	·>			·>	·>
(	<·	<·	<·	<·	<·	<·	<·	=	
)	·>	·>	·>	·>	·>			·>	·>
$	<·	<·	<·	<·	<·	<·	<·		

Example

- Verify whether the given grammar is Operator Grammar or not.

  E -> E A E | (E) | id

  A -> + | - | *

- It is not an operator grammar as it has two adjacent non-terminals on RHS of production **E -> E A E**.
- It can be converted to operator grammar by substituting the value of A.
- The new Grammar will be:

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

Finding handles

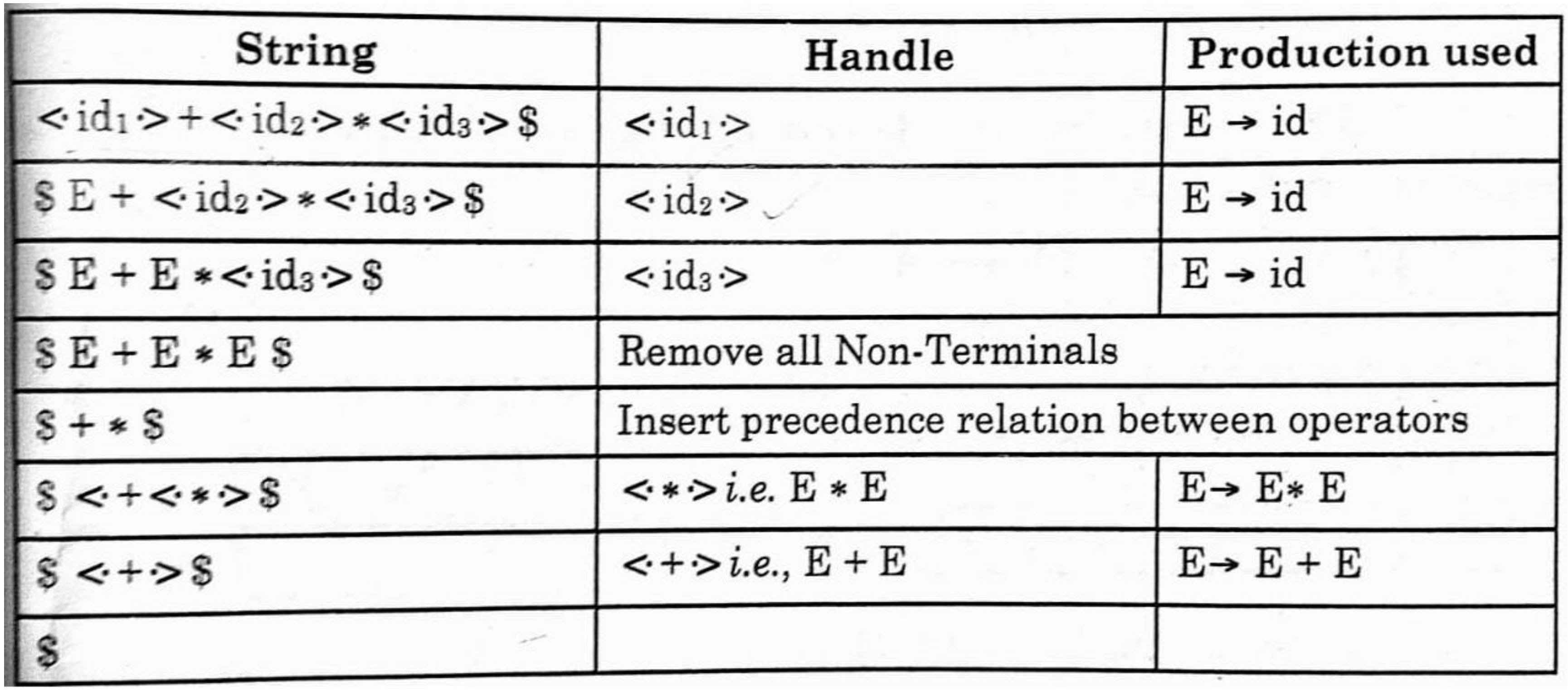
- Handles can be found by the following process:
  - Scan the input string from left to right until first $\gtrdot$ is encountered.
  - Scan backward until $\lessdot$ is encountered.
  - Handle is a string between $\lessdot$ and $\gtrdot$

- Example:

  Consider a Grammar E-> E+E | E*E | (E) | id

  Find handles at each step for reducing the string **id+id*id** using operator precedence parsing.

- **E-> E+E | E*E | (E) | id**
- Input String: **id+id*id**
- Attach $ at starting and ending of string i.e. **$ id+id*id $**
- Put precedence relation between operators & symbols using Precedence Relation Table.



String	Handle	Production used
$\lessdot id_1 \gtrdot + \lessdot id_2 \gtrdot * \lessdot id_3 \gtrdot \$$	$\lessdot id_1 \gtrdot$	$E \rightarrow id$
$\$ E + \lessdot id_2 \gtrdot * \lessdot id_3 \gtrdot \$$	$\lessdot id_2 \gtrdot$	$E \rightarrow id$
$\$ E + E * \lessdot id_3 \gtrdot \$$	$\lessdot id_3 \gtrdot$	$E \rightarrow id$
$\$ E + E * E \$$	Remove all Non-Terminals	
$\$ + * \$$	Insert precedence relation between operators	
$\$ \lessdot + \lessdot * \gtrdot \$$	$\lessdot * \gtrdot$ *i.e.* $E * E$	$E \rightarrow E * E$
$\$ \lessdot + \gtrdot \$$	$\lessdot + \gtrdot$ *i.e.*, $E + E$	$E \rightarrow E + E$
$\$$		

Precedence relations in operator grammar

- Equal precedence
  - If RHS of production is of the form $\boldsymbol{\alpha}$ **a** $\boldsymbol{\beta}$ **b** $\boldsymbol{\gamma}$,
  - where ,
    - α and γ can be any strings
    - β can be ε or single non-terminal
  - then, **a** $\doteq$ **b**

- Less precedence
  - If RHS of the production is of the form $\boldsymbol{\alpha}$ **a A** $\boldsymbol{\beta}$ and **A=>** $\boldsymbol{\gamma}$ **b $**
  - where,
    - γ is ε or single non-terminal
  - then, **a** $\lessdot$ **b**

- Greater precedence
  - If RHS of the production is of the form $\boldsymbol{\alpha}$ **A b** $\boldsymbol{\beta}$ and **A=>** $\boldsymbol{\gamma}$ **a $**
  - where,
    - $ is ε or single non-terminal
  - then, **a** $\gtrdot$ **b**

1. **$a \doteq b$ (Equal Precedence) :** If R.H.S of production is of form α **a** β **b** γ, where β can be ε or single non-terminal then **$a \doteq b$.**

Here, α and γ can be any strings.

E.g In grammar, $S \rightarrow m\,A\,c\,B\,e\,d$

On comparing mAcBed with $\alpha a \beta b \gamma$

$\alpha = mA,\ a = c\ ,\ \beta = B\ ,\ b = e,\ \gamma = d$

α	a	β	b	γ
mA	c	B	e	d

So, comparing **a with c** and **b with e** we get **$c \doteq e$.**

We can also make a different combination for a and b.

In Grammar **$S \rightarrow m\,A\,c\,Bed$**

$\alpha = \varepsilon,\ a = m\ ,\ \beta = A,\ b = c,\ \gamma = Bed$

α	a	β	b	γ
ε	m	A	c	Bed

$\therefore$ So comparing **a with m** and **b with c**

$\therefore m \doteq c$

2. $a \lessdot b$ **(Less then)**

If R.H.S of production is of form α a A β and A $\overset{+}{\Rightarrow} \gamma b\$$ where γ is ε or singl

non-terminal then a $\lessdot$ b.

E.g In Grammar $S \rightarrow m\,A\,c\,D$

$A \rightarrow i$

On comparing m A c D with α a A β and A → i with $A \rightarrow \gamma b\$$

α	a	A	β
ε	m	A	cD

A →	γ	b	\$
A →	ε	i	ε

$\therefore$ $\alpha = \varepsilon$, a = m, A = A, β = cD

$\therefore$ $\gamma = \varepsilon$,

and b = i

$\therefore$ Applying the rule,

a $\lessdot$ b means m $\lessdot$ i

3. a ⋗ b (Greater Than)

If R.H.S of production is of form $\alpha A b \beta$ and $A \overset{+}{\Rightarrow} \gamma a \$$ where $ is ε or single no

terminal then **a ⋗ b.**

E.g In Grammar $S \rightarrow m\,A\,c\,D$

$A \rightarrow i$

On comparing mA cD with $\alpha A\, b\beta$ and $A \rightarrow i$ with $A \rightarrow \gamma a\$$

α	A	b	β
m	A	c	D

A→	γ	a	$
A→	ε	i	ε

$\therefore \quad \alpha = m, \quad A = A, \quad b = c, \quad \beta = D$

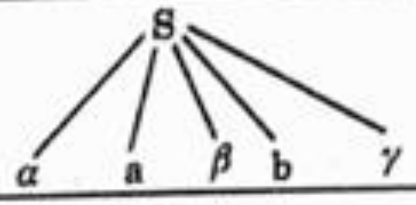
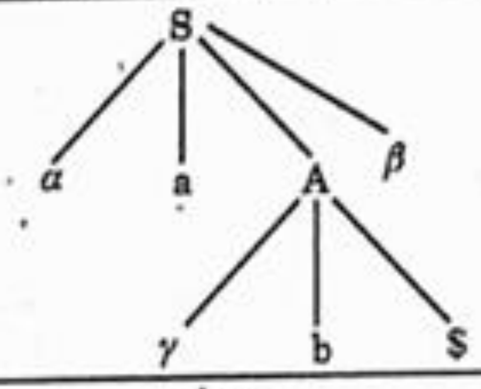
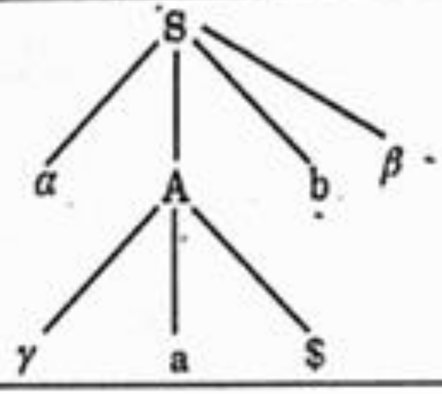
on comparing i with $\gamma a\$$

$\therefore \quad \gamma = \varepsilon, \quad a = i, \quad \$ = \varepsilon$

$\therefore$ On applying rule,

a ⋗ b means i ⋗ c.

The Precedence relations between terminal symbols can also be shown by a parse Tree :

1. a = b	
2. a ⋖ b	
3. a ⋗ b	

FIRST and LAST

- Consider the Grammar:

  E -> E + T | T

  T -> T * F | F

  F -> (E) | id

  1. E -> E + T

     FIRST(E) = {+}
  2. T -> T * F

     FIRST(T) = {*}
  3. F -> (E) | id

     FIRST(F) = {(, id}

- Also, **E -> T -> F**
- Therefore,
  - FIRST(F) is contained in FIRST(T)
  - FIRST(T) is contained in FIRST(E)
- **FIRST(F) = {(, id}**
- FIRST(T) = FIRST(T) ∪ FIRST(F)

  = **{*, (, id}**
- FIRST(E) = FIRST(E) ∪ FIRST(T)

  = **{+, *, (, id}**

Similarly, LAST can be found.

NON- TERMINAL	FIRST	LAST
F	(, id	), id
T	*, (, id	*,), id
E	+, *, (, id	+, *,), id

Operator Precedence Grammar

- A Grammar is called Operator Precedence Grammar if
  - it does not contain ε
  - Precedence relations $\lessdot$, $\doteq$, $\gtrdot$ are disjoint
    - i.e. if **a** $\gtrdot$ **b** exists, then **b** $\gtrdot$ **a** will not exist.

- Example

  Check whether the following Grammar is OPG or not.

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

LEADING

- If production is of the form A -> aα or A -> Baα
  - where,
    - B is a non-terminal
    - α can be any string
  - First terminal symbol on RHS is

    LEADING (A)= {a}

- If production is of the form A -> Bα
  - If a is in LEADING (B),
  - Then a will also be in LEADING (A).

TRAILING

- If production is of the form A -> αa or A -> αaB
  - where,
    - B is a non-terminal
    - α can be any string
  - TRAILING (A)= {a}

- If production is of the form A -> αB
  - If a is in TRAILING (B),
  - Then a will also be in TRAILING (A).

Example

- Compute LEADING and TRAILING for non-terminals E, T, F in the following Grammar

  E-> E + T | T

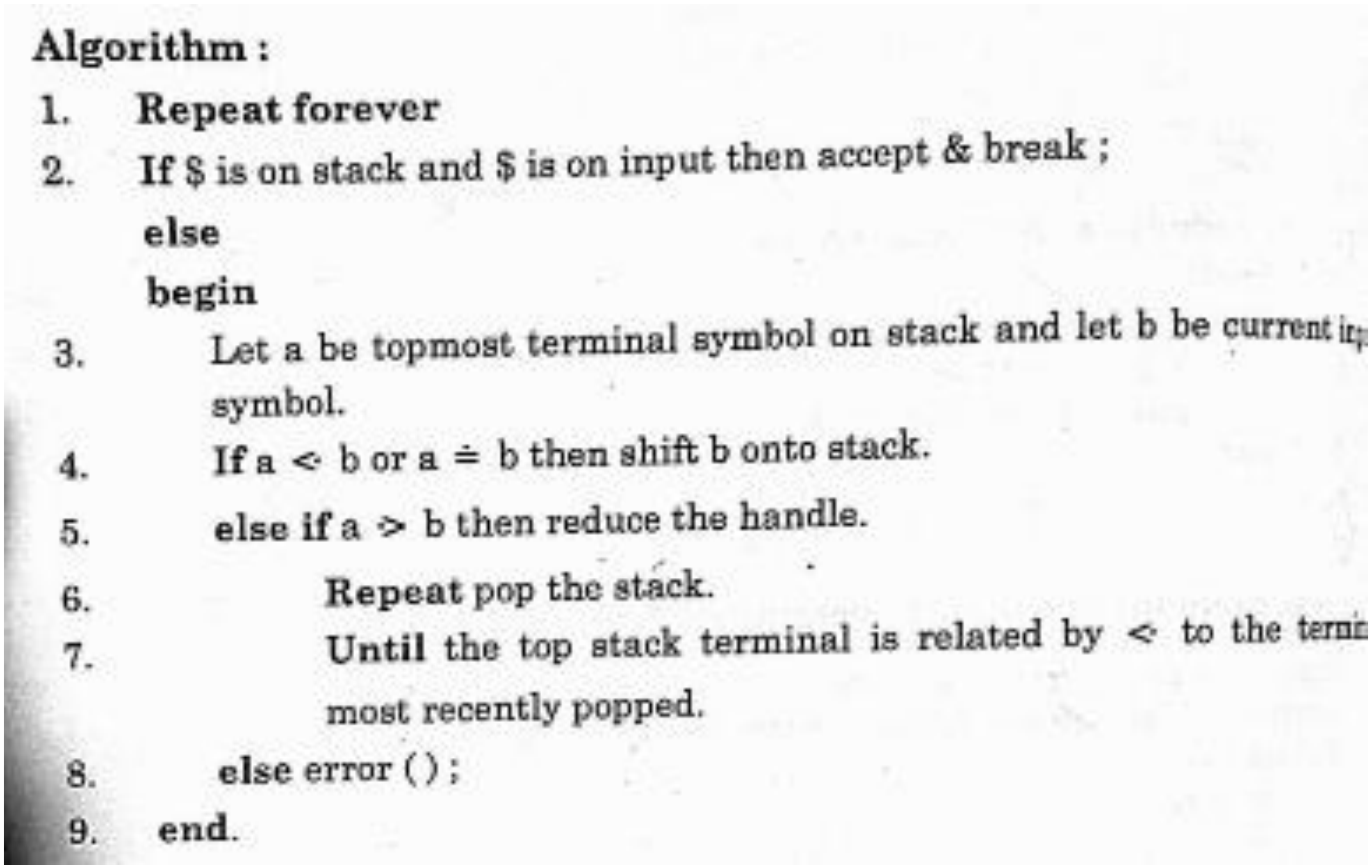
  T-> T * F | F

  F -> (E) | id

- LEADING (F) = { (, id}
- LEADING (T) = {*, (, id}
- LEADING (E) = {+, *, (, id}

- TRAILING (F) = {), id}
- TRAILING (T) = { *,), id}
- TRAILING (E) = {+, *,), id}

Operator Parsing Algorithm



Algorithm :

```
1.  Repeat forever
2.  If $ is on stack and $ is on input then accept & break ;
     else
     begin
3.      Let a be topmost terminal symbol on stack and let b be current i
        symbol.
4.      If a $\lessdot$ b or a $\doteq$ b then shift b onto stack.
5.      else if a $\gtrdot$ b then reduce the handle.
6.              Repeat pop the stack.
7.              Until the top stack terminal is related by $\lessdot$ to the term
                most recently popped.
8.      else error ( ) ;
9.   end.
```

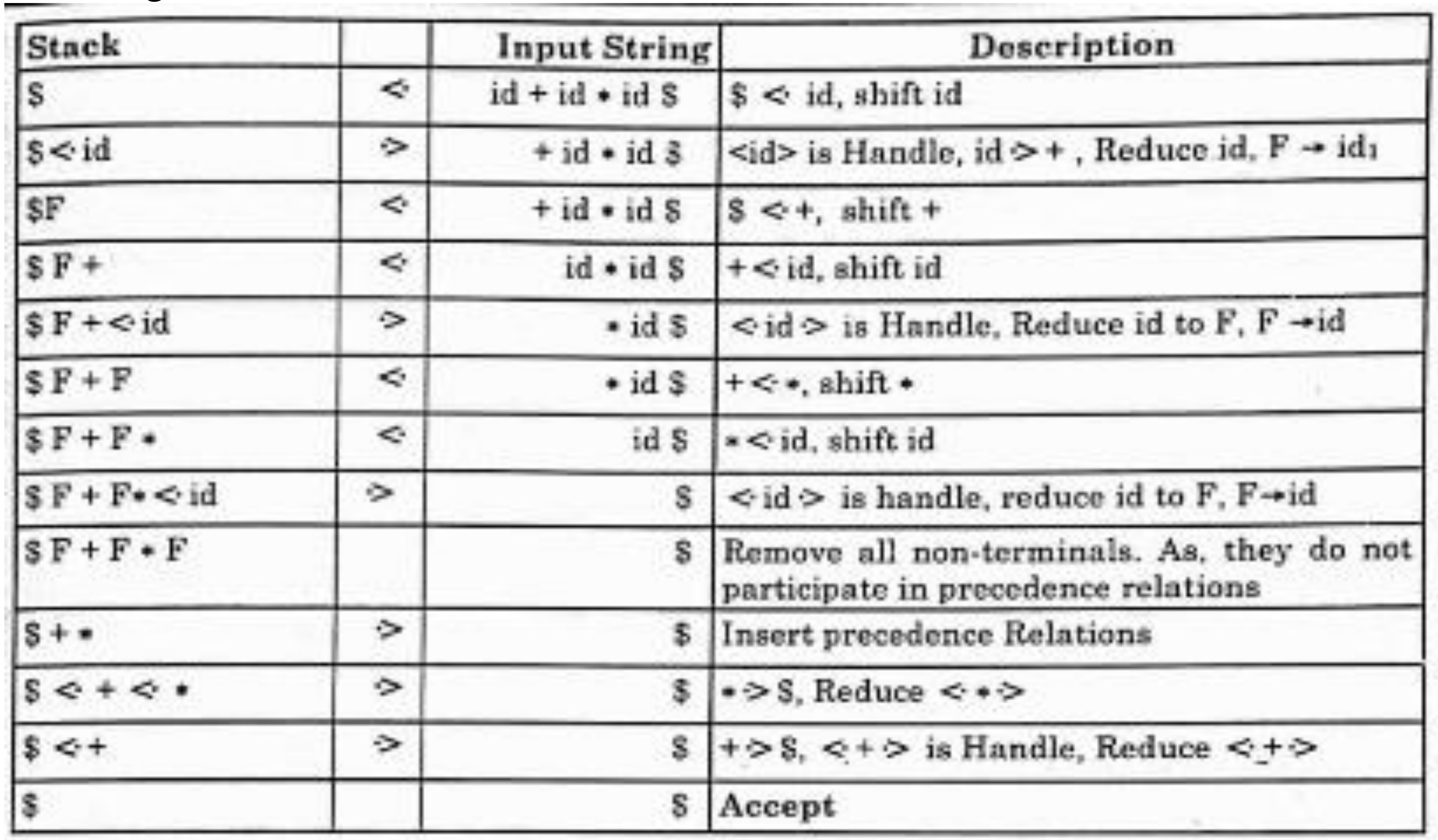
- Consider the Grammar

  E-> E +T | T

  T-> T * F | F

  F -> (E) | id

  Perform Stack implementation for string **id+id*id** using Operator Precedence Parsing.



Stack		Input String	Description
$	⋖	id + id * id $	$ ⋖ id, shift id
$⋖id	⋗	+ id * id $	<id> is Handle, id ⋗ + , Reduce id, F → id;
$F	⋖	+ id * id $	$ ⋖ +, shift +
$ F +	⋖	id * id $	+ ⋖ id, shift id
$ F + ⋖ id	⋗	* id $	⋖ id ⋗ is Handle, Reduce id to F, F → id
$ F + F	⋖	* id $	+ ⋖ *, shift *
$ F + F *	⋖	id $	* ⋖ id, shift id
$ F + F* ⋖ id	⋗	$	⋖ id ⋗ is handle, reduce id to F, F→id
$ F + F * F		$	Remove all non-terminals. As, they do not participate in precedence relations
$ + *	⋗	$	Insert precedence Relations
$ ⋖ + ⋖ *	⋗	$	* ⋗ $, Reduce ⋖ * ⋗
$ ⋖+	⋗	$	+ ⋗ $, ⋖ + ⋗ is Handle, Reduce ⋖ + ⋗
$		$	Accept

Precedence Functions

- Precedence relations between any two operators or symbols in precedence table can be converted to two precedence functions f and g that map terminal symbols to integers.

- For symbols a and b.

$f(a) \lessdot g(b)$ whenever $a \lessdot b$

$f(a) \doteq g(b)$ whenever $a \doteq b$

$f(a) \gtrdot g(b)$ whenever $a \gtrdot b$

Computation of Precedence Functions

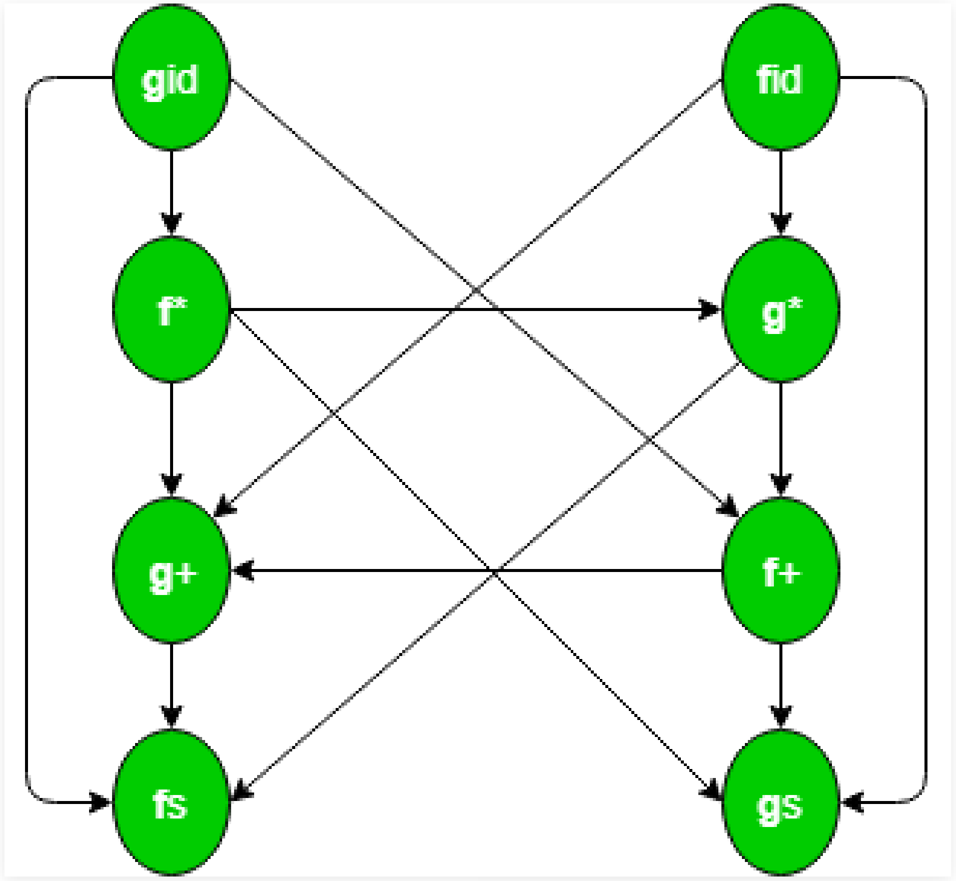
- For each terminal a, create symbol f_a and g_a.
- Make node for each symbol
  - If $a \doteq b$, then f_a and g_b are in same group or node.
  - If $a \doteq b$ & $c \doteq b$, then f_a & f_c must be in same group or node.
- If $a \lessdot b$,
  - mark an edge from g_b to f_a
- If $a \gtrdot b$,
  - mark an edge from f_a to g_b.
- If graph constructed has cycle, then no precedence functions exist.
- If there are no cycles,
  - $f(a)$ = length of longest path beginning at group of f_a.
  - $g(a)$ = length of longest path beginning from group of g_a.

Example

- Construct precedence graph and precedence function for following table:

	id	+	*	$
id		·>	·>	·>
+	<·	·>	<·	·>
*	<·	·>	·>	·>
$	<·	<·	<·	·>

	id	+	*	$
f	4	2	4	0
g	5	1	3	0



Shift Reduce Parsing

- It is a type of Bottom Up parser.
  - Generates parse tree from the leaves to the root.
- The input string will be reduced to the starting symbol.
- This reduction can be achieved by applying Rightmost derivation in reverse i.e. from starting symbol to the input string.

- SR Parser requires 2 data structures:
  - **INPUT BUFFER**
  - **STACK**

Example

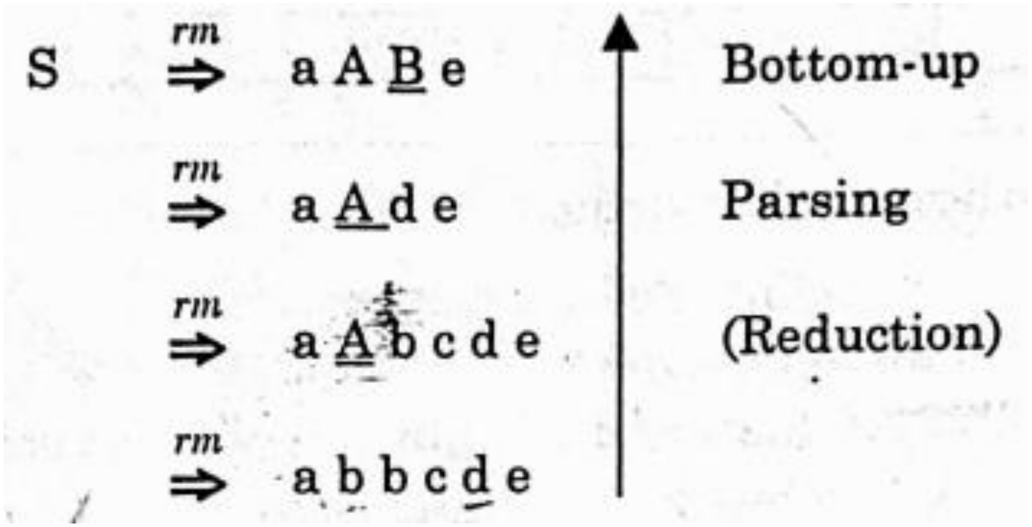
- Perform bottom up parsing for the given string **abbcde** on the following Grammar:

  S -> aABe

  A -> Abc | b

  B -> d

Solution:



$$
\begin{aligned}
S &\overset{rm}{\Rightarrow} a\,A\,\underline{B}\,e \\
&\overset{rm}{\Rightarrow} a\,\underline{A}\,d\,e \\
&\overset{rm}{\Rightarrow} a\,\underline{A}\,b\,c\,d\,e \\
&\overset{rm}{\Rightarrow} a\,b\,b\,c\,\underline{d}\,e
\end{aligned}
$$

Stack Implementation of Shift Reduce Parsing

- Following steps are to be followed for Shift Reduce Parsing:
  - Use a Stack and an Input buffer.
  - Insert $ at the bottom of the Stack and the right end of the input string in Input buffer.
  - Shift
    - Shift zero or more input symbols onto the stack until handle is on the top of stack.
  - Reduce
    - Handle at the Stack top is replaced by the LHS of the production.
  - Accept
    - Above two steps will be repeated until detected an error or the stack and the input buffer contains the start symbol only.

Example

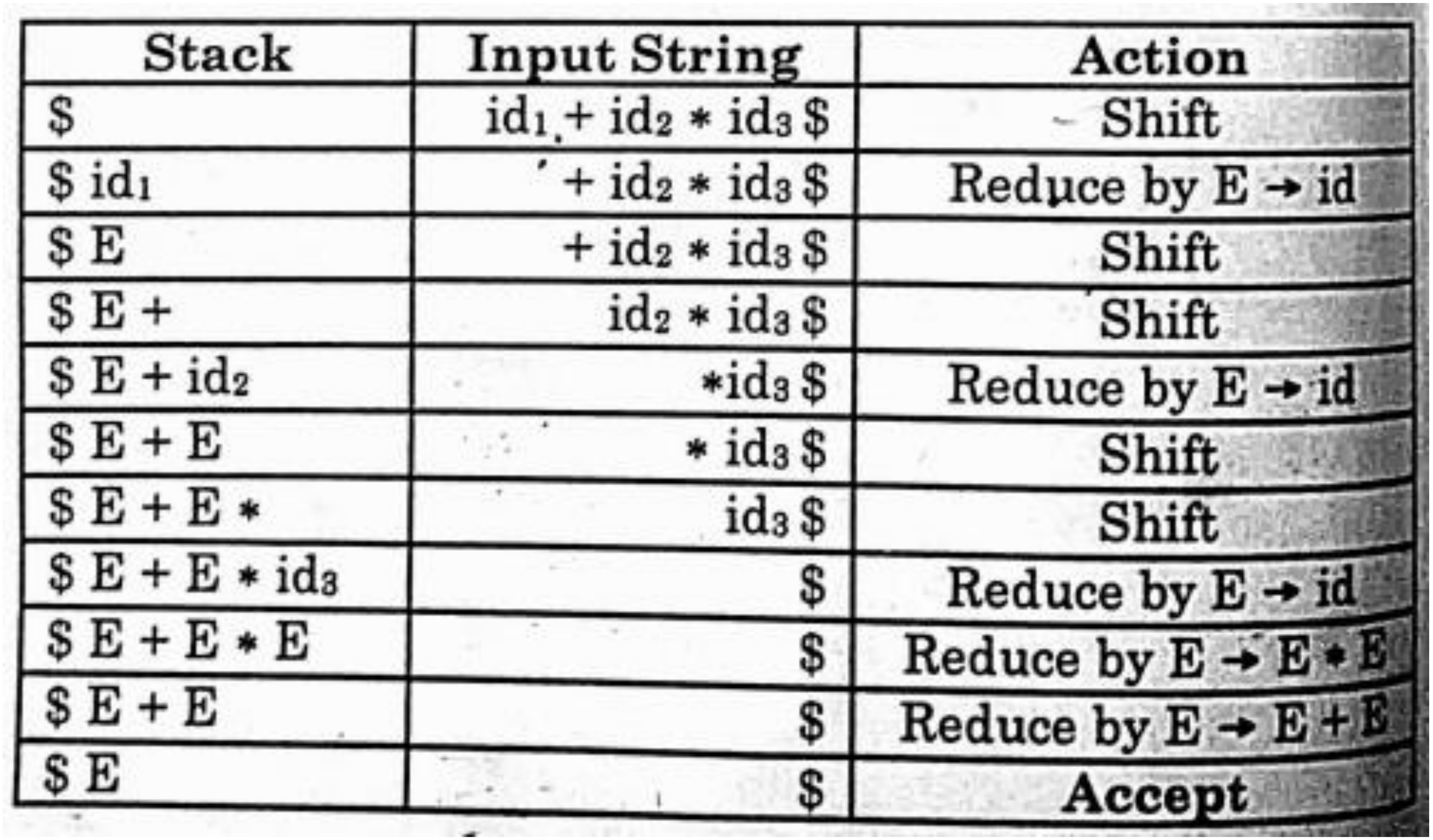
- Perform Shift Reduce Parsing for the string $\mathbf{id_1+id_2*id_3}$ for the following Grammar:

  E -> E + E

  E -> E * E

  E -> (E)

  E -> id



Stack	Input String	Action
\$	$id_1 + id_2 * id_3$ \$	Shift
\$ id_1	$+ id_2 * id_3$ \$	Reduce by $E \rightarrow id$
\$ E	$+ id_2 * id_3$ \$	Shift
\$ E +	$id_2 * id_3$ \$	Shift
\$ E + id_2	$*id_3$ \$	Reduce by $E \rightarrow id$
\$ E + E	$* id_3$ \$	Shift
\$ E + E *	id_3 \$	Shift
\$ E + E * id_3	\$	Reduce by $E \rightarrow id$
\$ E + E * E	\$	Reduce by $E \rightarrow E * E$
\$ E + E	\$	Reduce by $E \rightarrow E + E$
\$ E	\$	Accept

Drawbacks of Shift Reduce Parsing

- Shift Reduce Conflict
  - Sometimes, SR parser cannot decide whether to shift or to reduce.

- Reduce Reduce Conflict
  - Sometimes, parser cannot decide which of the productions should be used for reduction.

Top Down Parsing

- Parse tree is generated from top to bottom i.e. from Root to leaves and expand it till all leaves are generated.
- Starts derivation from start symbol of grammar.
- Performs leftmost derivation at each step.

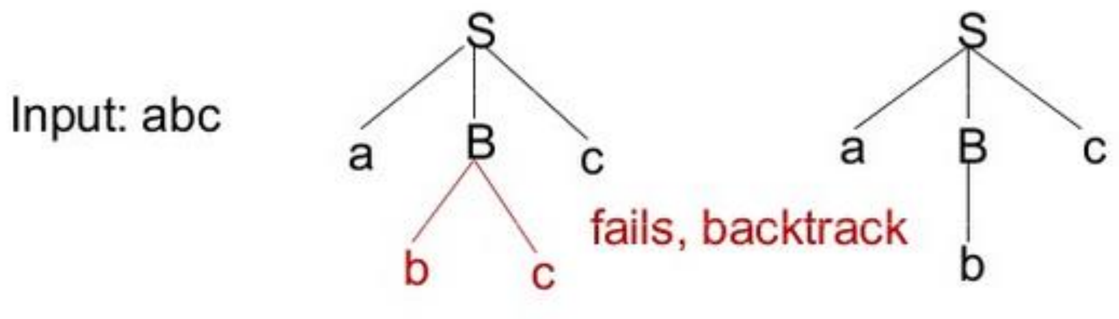
- Types of Top Down parsing
  - With backtracking
    - Parser can make repeated scans of input.
    - If required input string is not achieved by applying one production rule, another production rue can be applied at each step to get the required string.
  - Without backtracking
    - Once the production rule is applied, it cannot be undone.
    - Below given are the parsers which follow top down parsing without backtracking:
      - Recursive Descent Parser
      - Predictive Parser
      - LL(1) Parser

Top Down Parsing with Backtracking

- Consider the grammar

  S -> aBc

  B -> bc|b



Limitations of Top-Down parsing with Backtracking

- Following are the limitation of backtracking:
  - Backtracking
    - Seems very simple & easy but choosing a different alternative causes a lot of problems.
  - Left Recursion
    - A grammar is said to be left recursive if it has production of the form A-> Aα|β.
    - Causes the parser to enter into infinite loop.
  - Choosing right production
    - The order in which alternatives are tried can effect the language accepted.
  - Difficult to locate error
    - When failure occurs, it is difficult to know where error actually occurred.

Elimination of Left Recursion

- Can be done by introducing a new non terminal A such that

  $A \to A\alpha \mid \beta$

  After removal of left recursion

  $A \to \beta A'$

  $A' \to \alpha A' \mid \varepsilon$

- This type of Recursion is called Immediate Left Recursion.
- In general,

$$A \to A\alpha_1 \mid A\alpha_2 \mid \dots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \dots \mid \beta_n$$

with

$$A \to \beta_1 B \mid \beta_2 B \mid \dots \mid \beta_n B$$
$$B \to \alpha_1 B \mid \alpha_2 B \mid \dots \mid \alpha_m B \mid \varepsilon$$

Example

- E -> E+T | T

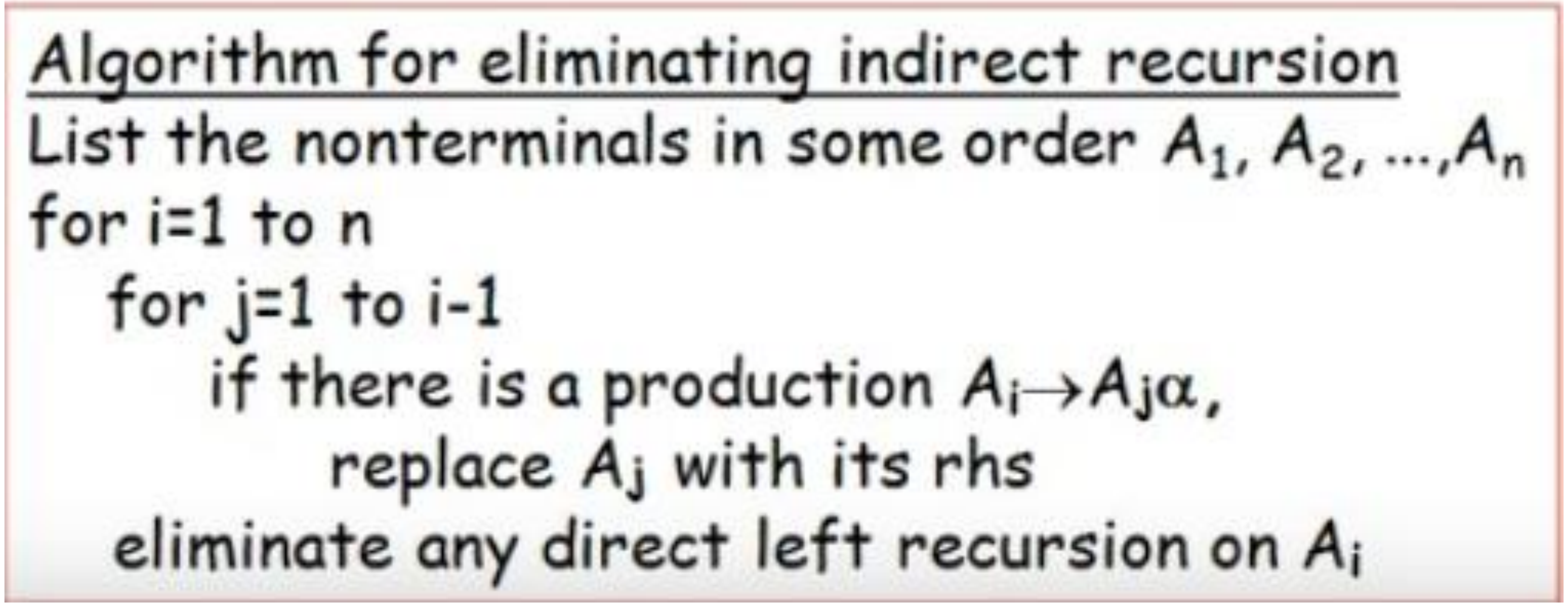
  T -> T*F | F

  F -> (E) | id

- S -> a | ^ | (T)

  T -> T, S | S

Non-immediate left recursion



<u>Algorithm for eliminating indirect recursion</u>

```
List the nonterminals in some order $A_1, A_2, \ldots, A_n$
for i=1 to n
    for j=1 to i-1
        if there is a production $A_i \to A_j\alpha$,
            replace $A_j$ with its rhs
    eliminate any direct left recursion on $A_i$
```

Example

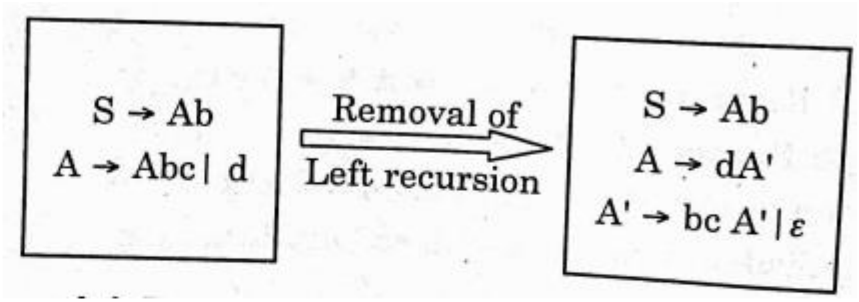
- S -> Ab ...(1)

  A -> Sc|d ...(2)

  Substituting the value of (1) in (2)

  S -> Ab

  A -> Abc|d



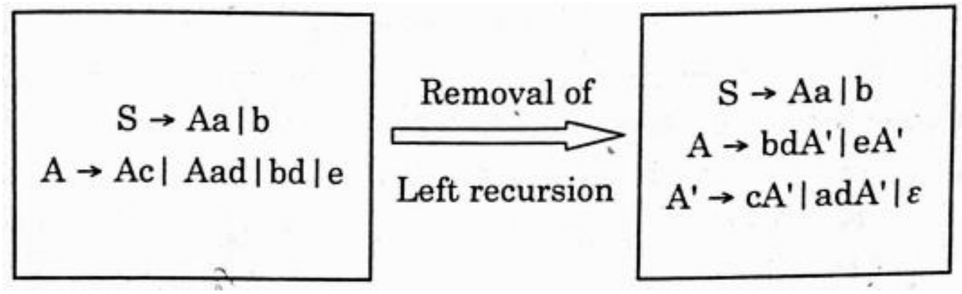
- S -> Aa|b ...(1)

  A -> Ac|Sd|e ...(2)

  Substituting the value of (1) in (2)

  S -> Aa|b

  A -> Ac|Aad|bd|e

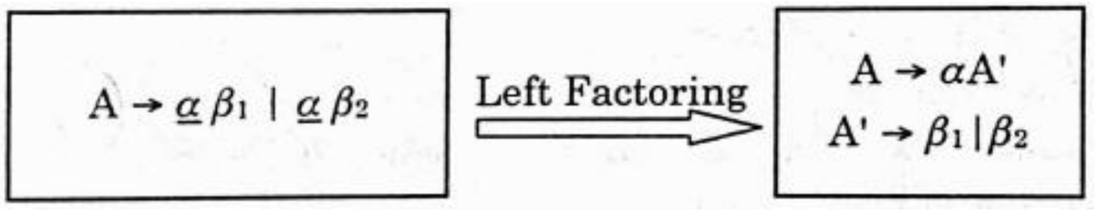


Left Factoring

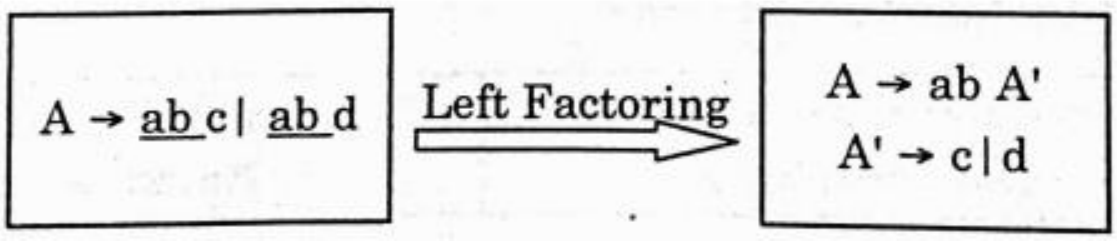
- If more than one production is available to expand non-terminal, it is not clear that which production should be used to expand non-terminal
- E.g.

  A -> **ab**c|**ab**d

  - Such type of grammar can be left factored as:


$A \rightarrow \underline{\alpha}\beta_1 \mid \underline{\alpha}\beta_2$
Left Factoring
$A \rightarrow \alpha A'$
$A' \rightarrow \beta_1 \mid \beta_2$

  - Similarly, the above grammar can be left factored as:


$A \rightarrow \underline{ab}\,c \mid \underline{ab}\,d$
Left Factoring
$A \rightarrow ab\,A'$
$A' \rightarrow c \mid d$

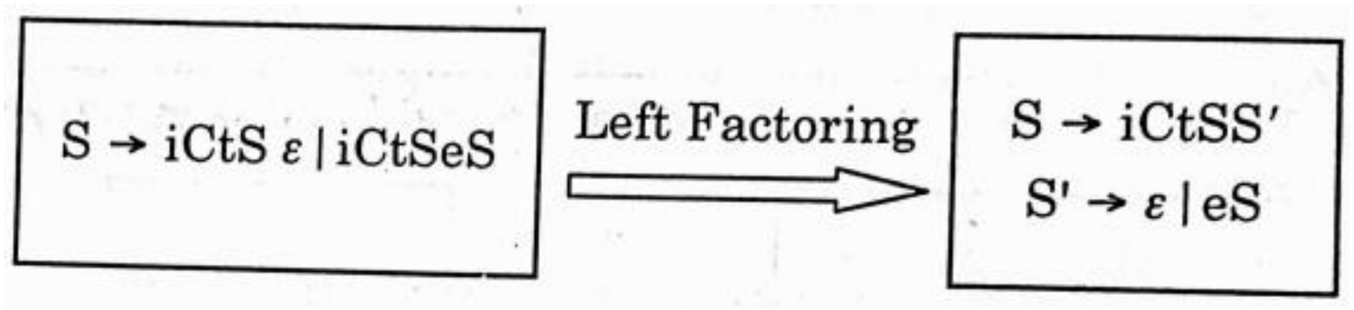
Example

- Perform Left Factoring on the given grammar

  S -> iCtS | iCtSeS | b

  C -> d

- **Solution:**



- The complete grammar will now be:

  S -> iCtSS' | b

  S'-> ε | eS

  C -> d

Backtracking Vs Non-Backtracking

Parsing with backtracking	Parsing without backtracking
Parser can try all alternatives in any order till it successfully parses the string.	Parser has to select correct alternative at each step.
Backtracking takes a lot of time to perform parsing.	Takes less time.
Grammar can have left recursion.	Left recursion will be removed.
Lot of overhead while undoing	Less overhead.
Information once inserted will be removed while backtracking	Information will not be removed.
Difficult to locate error.	Easy to locate error.

Top Down parsing without backtracking

- While doing parsing without backtracking, following steps should be followed:
  - Left recursion should be removed.
  - Left factoring should be performed on given grammar.

- Mainly two types of parsers can be used:
  - Recursive Descent Parser
  - Predictive Parser.

Recursive Descent Parser

- It is a parser that uses various recursive procedures to process the input string with no backtracking.

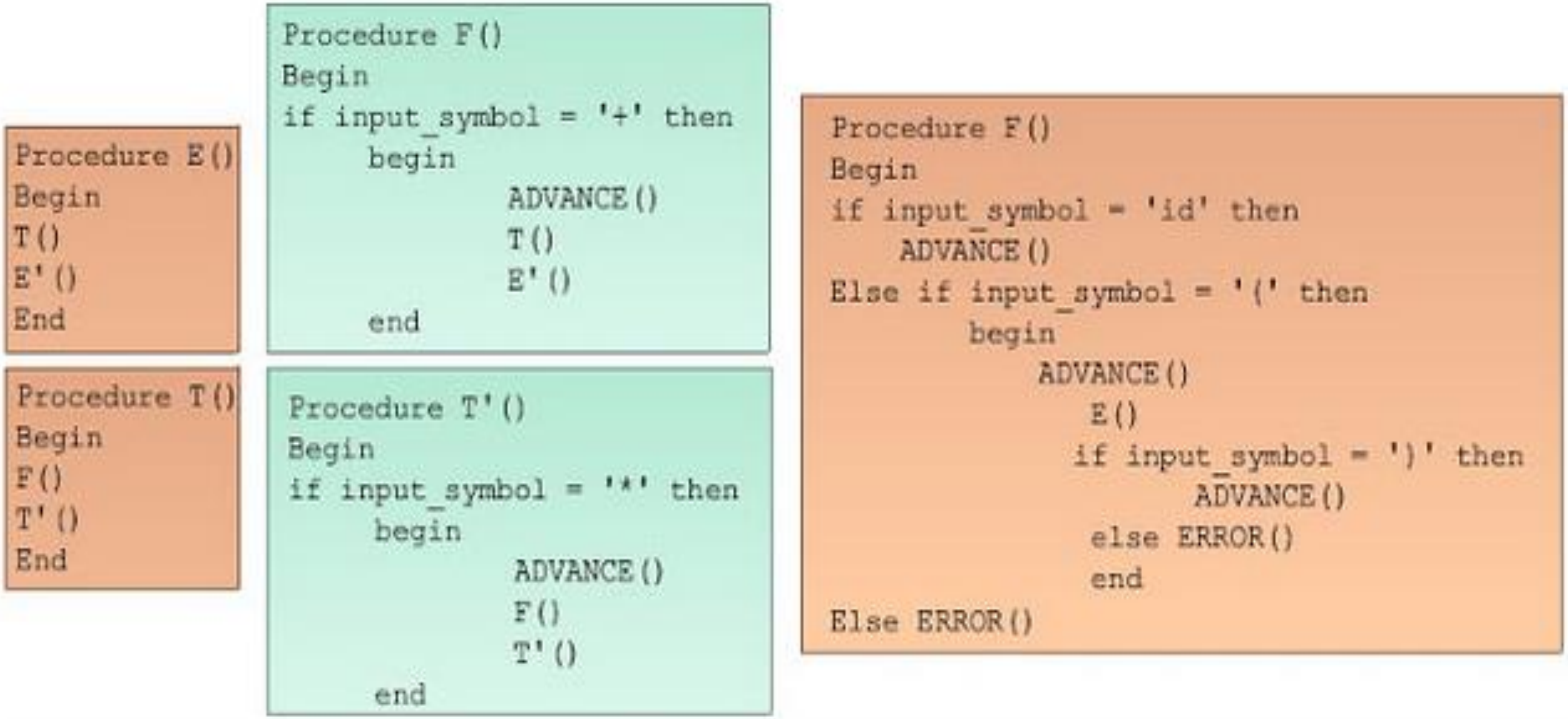
- Example:
- Given a grammar E->TE'

  E' -> +TE' | ε

  T-> FT'

  T' -> *FT' | ε

  F -> (E) | id



```
Procedure E()
Begin
T()
E'()
End
```

```
Procedure T()
Begin
F()
T'()
End
```

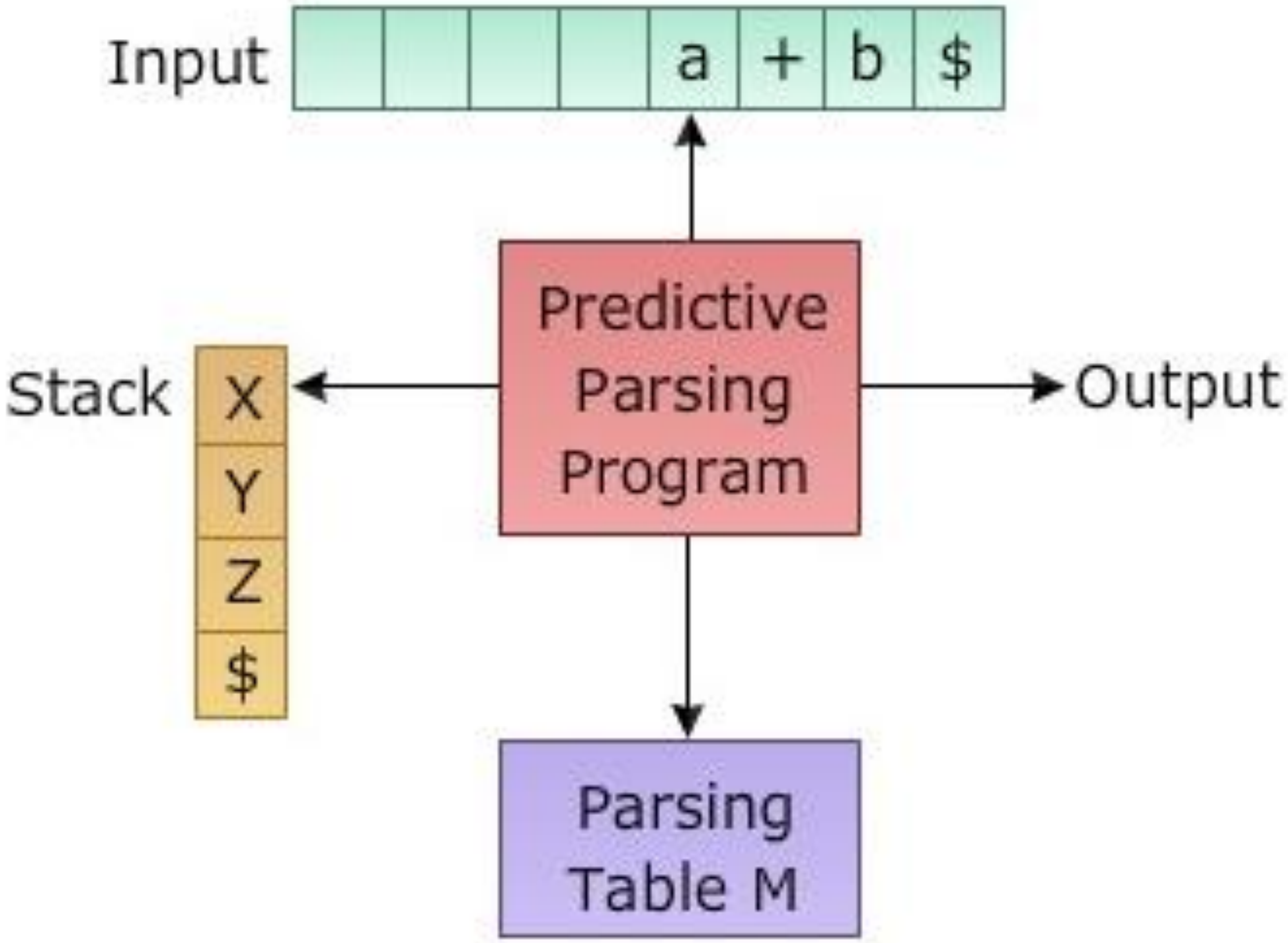
```
Procedure F()
Begin
if input_symbol = '+' then
      begin
              ADVANCE()
              T()
              E'()
      end
```

```
Procedure T'()
Begin
if input_symbol = '*' then
      begin
              ADVANCE()
              F()
              T'()
      end
```

```
Procedure F()
Begin
if input_symbol = 'id' then
    ADVANCE()
Else if input_symbol = '(' then
        begin
            ADVANCE()
               E()
              if input_symbol = ')' then
                    ADVANCE()
               else ERROR()
               end
Else ERROR()
```

Predictive Parser

- Implements Top-Down parsing without backtracking.
- Performs parsing using stack implementation.
- It is non recursive.
- It has following components:
  - Input Buffer
  - Stack
  - Parsing Table
    - Two-dimensional array or Matrix M[A,a] where A is non terminal and ‘a’ is a terminal symbol.
  - Parsing Program
    - Performs some action by comparing the symbol on top of stack and current input symbol to be read on input buffer
  - Action
    - Takes various actions depending upon the symbol on TOS and current input symbol.


Input
a
+
b
$
Stack
X
Y
Z
$
Predictive Parsing Program
Output
Parsing Table M

Action that can be taken

Description	Top of Stack	Current Input Symbol	Action
If stack is empty	$	$	Parsing is successful and halted
If symbol on TOS and input buffer are terminals and same	a	a	POP a from stack & advance to next input symbol
If symbol on TOS and input buffer are terminals and different	a	b	Error
If TOS is non terminal & input symbol is terminal	X	a	Refer to entry M[X,a] in parsing table. If M[X,a]-> ABC, POP X from stack and PUSH C B A onto stack

Construction of Predictive Parser

- Compute FIRST and FOLLOW for different symbols of grammar

- Construct predictive parser table using FIRST and FOLLOW

- Perform parsing of input string with the help of table.

Computation of FIRST

- FIRST(α) is defined as collection of terminal symbols which are first letters of strings derived from α.
- FIRST(α) = {α | α -> αβ for some string β}

- If X is a grammar symbol, then FIRST(X) will be:
  1. If X is terminal symbol, then FIRST(X) = {X}
  2. If X -> ε, FIRST(X) = {ε}
  3. If X is non-terminal & X ->aα, then FIRST(X) = {a}
  4. If X->$Y_1Y_2Y_3$, then FIRST(X) will be
     a. If Y_1 is terminal, then
        FIRST(X) = FIRST($Y_1Y_2Y_3$) = {Y_1}
     b. If Y_1 is non-terminal and FIRST(Y_1) does not contain ε.
        FIRST(X) = FIRST($Y_1Y_2Y_3$) = FIRST(Y_1)
     c. If FIRST(Y_1) contains ε, then
        FIRST(X) = FIRST($Y_1Y_2Y_3$)
        = FIRST(Y_1) – {ε} ∪ FIRST(Y_2Y_3)

Computation of FOLLOW

- FOLLOW(A) is defined as collection of terminal symbols that appear immediately to the right of A.
- FOLLOW(A) = {a | S => $\alpha A a \beta$ where α, β can be any strings}

1. If S is start symbol, Follow(S) = {\$}
2. If production is of the form A -> $\alpha B \beta$, $\beta \neq \varepsilon$.
   a. If FIRST (β) does not contain ε then,
      - FOLLOW(B) = {FIRST(β)}

   b. If FIRST (β) contain ε then,
      - FOLLOW(B) = FIRST(β) – {ε} $\cup$ FOLLOW(A)
3. If production is of the form A -> αB then FOLLOW (B) = {FOLLOW(A)}.

Construction of Predictive Parsing Table

ALGORITHM

- For each terminal a in FIRST(α)
  - add A-> α to M[A,a].

- If ε is in FIRST(α), and b is in FOLLOW(A), then
  - add A-> α to M[A,b].

- If ε is in FIRST(α), and $ is in FOLLOW(A), then
  - add A-> α to M[A,$].

- All remaining entries in table M are error.

Steps to perform Predictive Parsing

- Elimination of left recursion
- Left factoring
- Computation of FIRST and FOLLOW
- Construction of Predictive parsing table
- Parse the input string.

Algorithm for Predictive Parsing

```
while Stack is not empty do
{
    let X be top symbol on stack
    let a be next input symbol
    if X is terminal or $ then
        if X = a then
            POP X from stack & remove a from input
        else error();
    else
        If M[X,a] = A-> $Y_1Y_2.......Y_n$
            POP X and PUSH $Y_nY_{n-1}........Y_2Y_1$ onto stack
        else error();
}
```

Example

Consider the Grammar

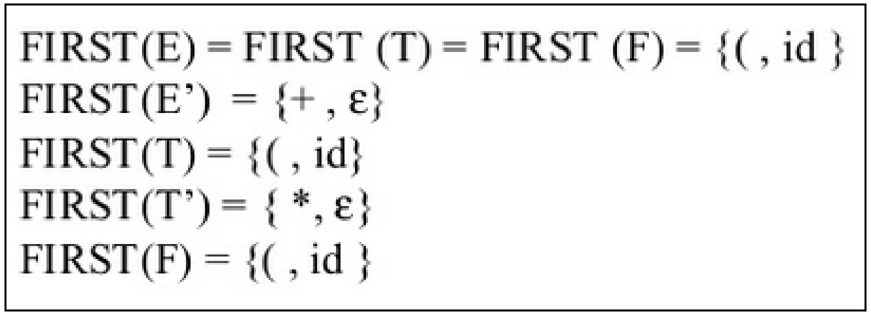
E->TE'

E' -> +TE'/ ε

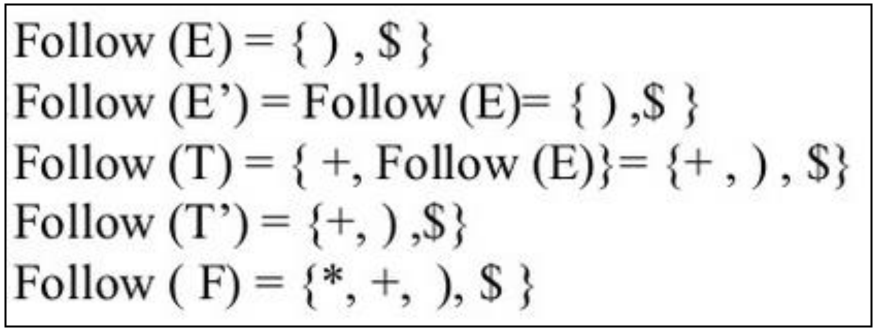
T-> FT'

T' -> *FT'/ ε

F -> (E)/id

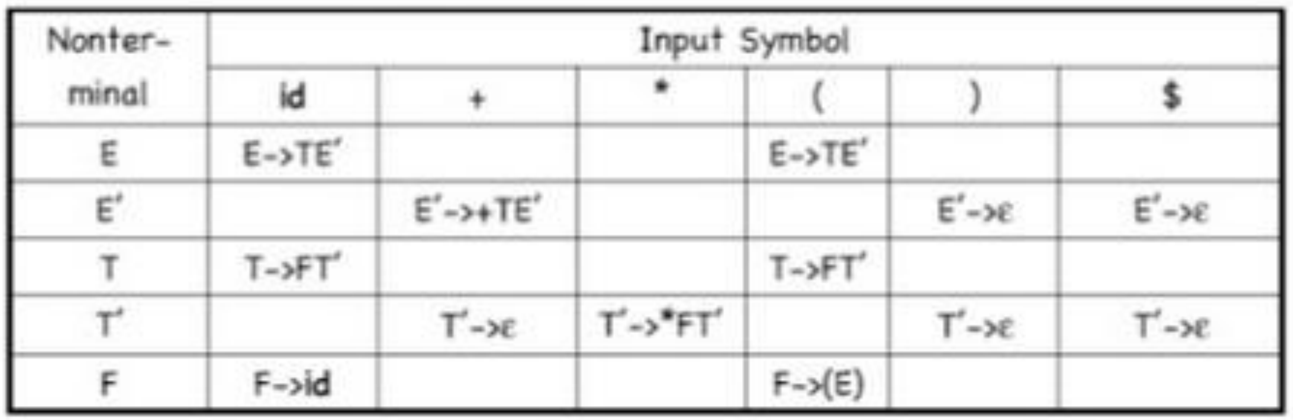


FIRST(E) = FIRST (T) = FIRST (F) = {(, id }
FIRST(E') = {+ , ε}
FIRST(T) = {(, id}
FIRST(T') = { *, ε}
FIRST(F) = {(, id }



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

Parsing Table



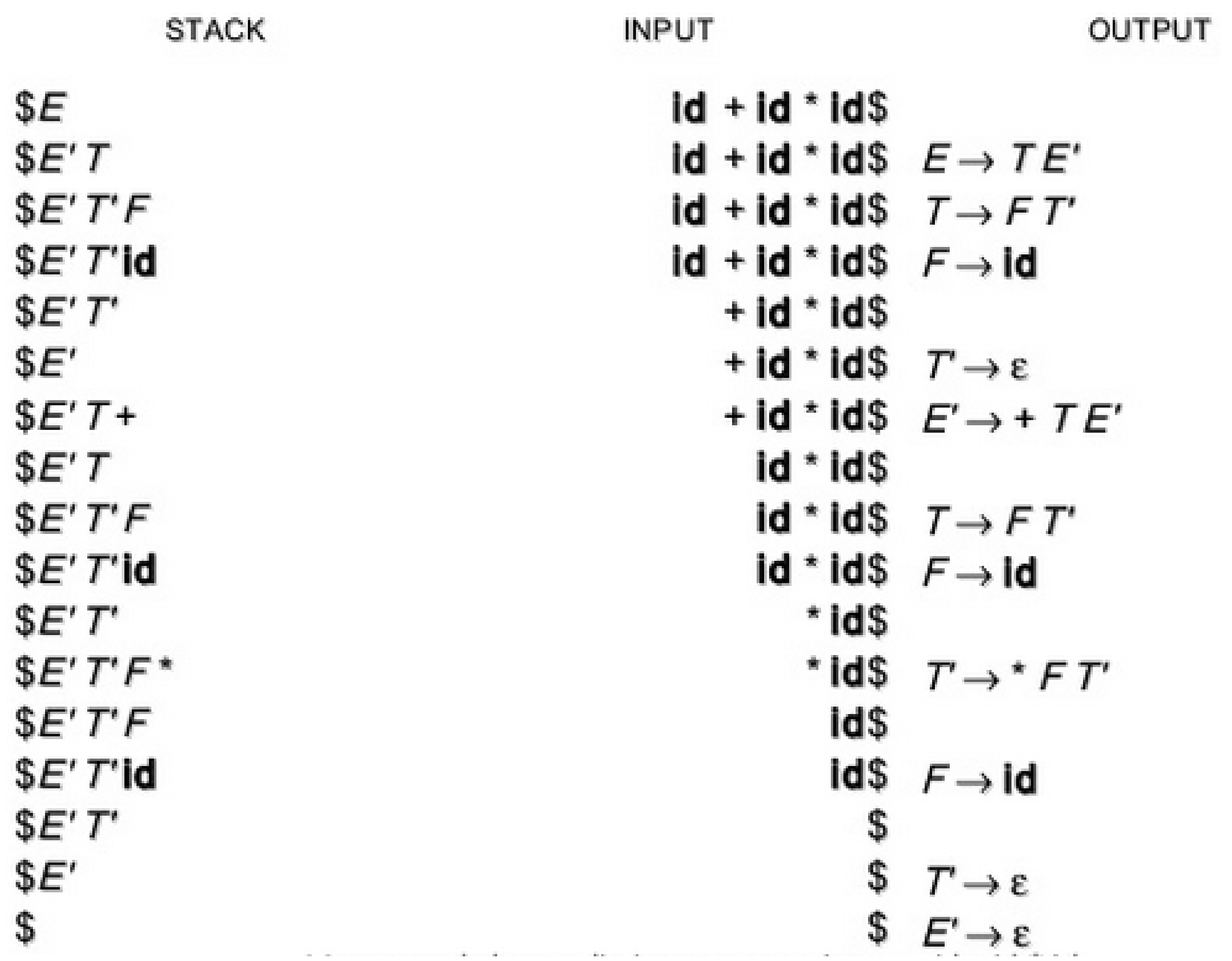
Nonter-minal	Input Symbol					
	id	+	*	(	)	$
E	E->TE'			E->TE'		
E'		E'->+TE'			E'->ε	E'->ε
T	T->FT'			T->FT'		
T'		T'->ε	T'->*FT'		T'->ε	T'->ε
F	F->id			F->(E)		

E -> T E'
E' -> + T E' | ε
T -> F T'
T' -> * F T' | ε
F -> (E) | id

Initial stack is $E

Initial input is id + id * id $

Sequence of Moves (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**\$	$T' \rightarrow \varepsilon$
$\$E'T+$	+ **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**
$\$E'T'$	\$	
$\$E'$	\$	$T' \rightarrow \varepsilon$
\$	\$	$E' \rightarrow \varepsilon$

LL(1) Grammar

- In LL(1)
  - First **L** means **left to right scanning of input**
  - Second **L** means **Leftmost derivation**
  - **1** is the **number of look ahead symbols** used by parser at each step to make parsing action.

- If only one symbol will decide which action to perform, then these parsers are called as LL(1) parser.
- Grammars on which these parsers are applied are called LL(1) grammar.

LL(1) Grammar

- Parsing table has no multiple defined entries.
- If production in grammar is of the form A-> α|β then:
  - FIRST(α) ∩ FIRST(β) ≠ {A} where a can be any terminal symbol.
  - FIRST(α) ∩ FIRST(β) ≠ {ε} i.e. atmost one of α & β can derive empty string.
  - If β => ε , then FIRST(α) ∩ FOLLOW(A) = ϕ i.e. α does not derive any string beginning with terminal in FOLLOW(A).

Example

- S -> iCtSS’|a

  S’-> eS| ε

  C -> b

- S-> (L)|a

  L-> L,S|S

Check whether string (a,a) is accepted or not

LR Parser

- It is a type of bottom up parser used to parse CFG.
- LR parser is actually called LR(K) parsing where,
  - L stands for Left to Right scanning of input.
  - R stands for rightmost derivation.
  - K is the number of input symbols of Look ahead that are used in making parsing decisions.

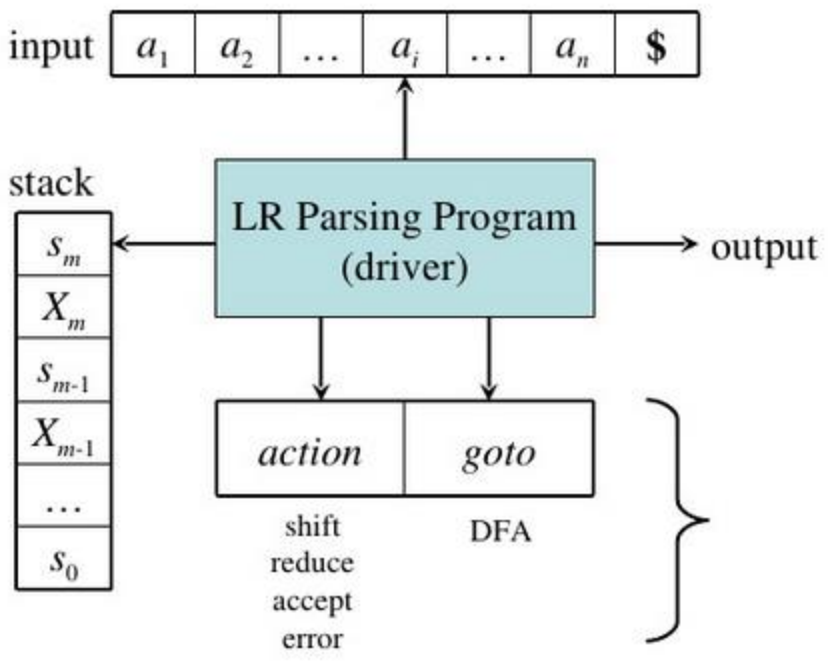
- Advantages of LR parsing:
  - LR parsers can recognize virtually all programming language written with CFG.
  - Uses non backtracking bottom up parser.
  - Can detect syntax error quickly.
  - Can recognize more languages than predictive parses.

- DISADVANTAGES of LR parsing:
  - Lot of work is needed to manually create LR parsing tables.

Structure of LR Parsers

- Following are the components of LR parser:
  - Input Buffer
  - Stack
  - Parsing Table
  - Parsing program
  - Actions

Model of an LR Parser



- Input Buffer:
  - Contains the string to be parsed, followed by \$ which indicates the end of the string.

- Stack:
  - Used to store grammar symbol and states
  - $s_0X_1s_1X_2\ldots\ldots\ldots X_ms_m\$$

  - Where
    - $X_1X_2\ldots\ldots\ldots X_m$ are grammar symbols.
    - $s_{0,}\ s_1\ \ldots\ldots\ldots s_m$ are states.

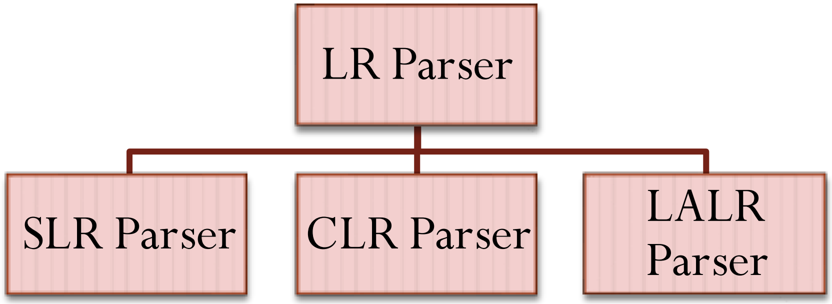
- Parsing Table:
  - It is divided into two sections
    - **Actions:**
      - Actions can be shift, reduce, accept and error.
    - **goto:**
      - It takes state and grammar symbols as arguments and produces a state e.g. goto (s,X)

- Parsing Program:
  - It is a program which performs following functions:
    - It maintains initial configuration (s_0,w\$) where s_0 is starting state, w is string
    - It maintains the following configuration
    - $s_0X_1s_1X_2\ldots\ldots X_ms_m,\ a_i\, a_{i+1} \ldots\ldots\ a_n\$$
    - It determines s_m the state currently on top of stack and a_i the current input symbol.
    - It takes action depending on the entry Action[s_m, a_j] in the parsing table.

- Actions:
  - Actions taken by parsers are of following types:
    - Shift
      - If action $[S_m, a_i]$ = shift s then shift a_i with state s onto the stack
      - $s_0X_1s_1X_2\ldots\ldots X_ms_ma_is$, $a_{i+1} \ldots\ldots a_n\$$
    - Reduce
      - If action $[S_m, a_i]$ = reduce A-> β, then parser perform reduction
      - Configuration becomes $s_0X_1s_1X_2\ldots\ldots X_{m-r}s_{m-r}As$, $a_i\ a_{i+1} \ldots\ldots a_n\$$
      - Here,
        - o S= goto$[s_{m-r}, A]$
        - o R=length of β
        - o Number of symbols popped = 2r (r state symbols + r grammar symbols)
    - Accept
      - If action $[S_m, a_i]$ = accept, Parsing is completed.
    - Error
      - If action $[S_m, a_i]$ = error, parser calls an error correcting function.

Types of LR Parser

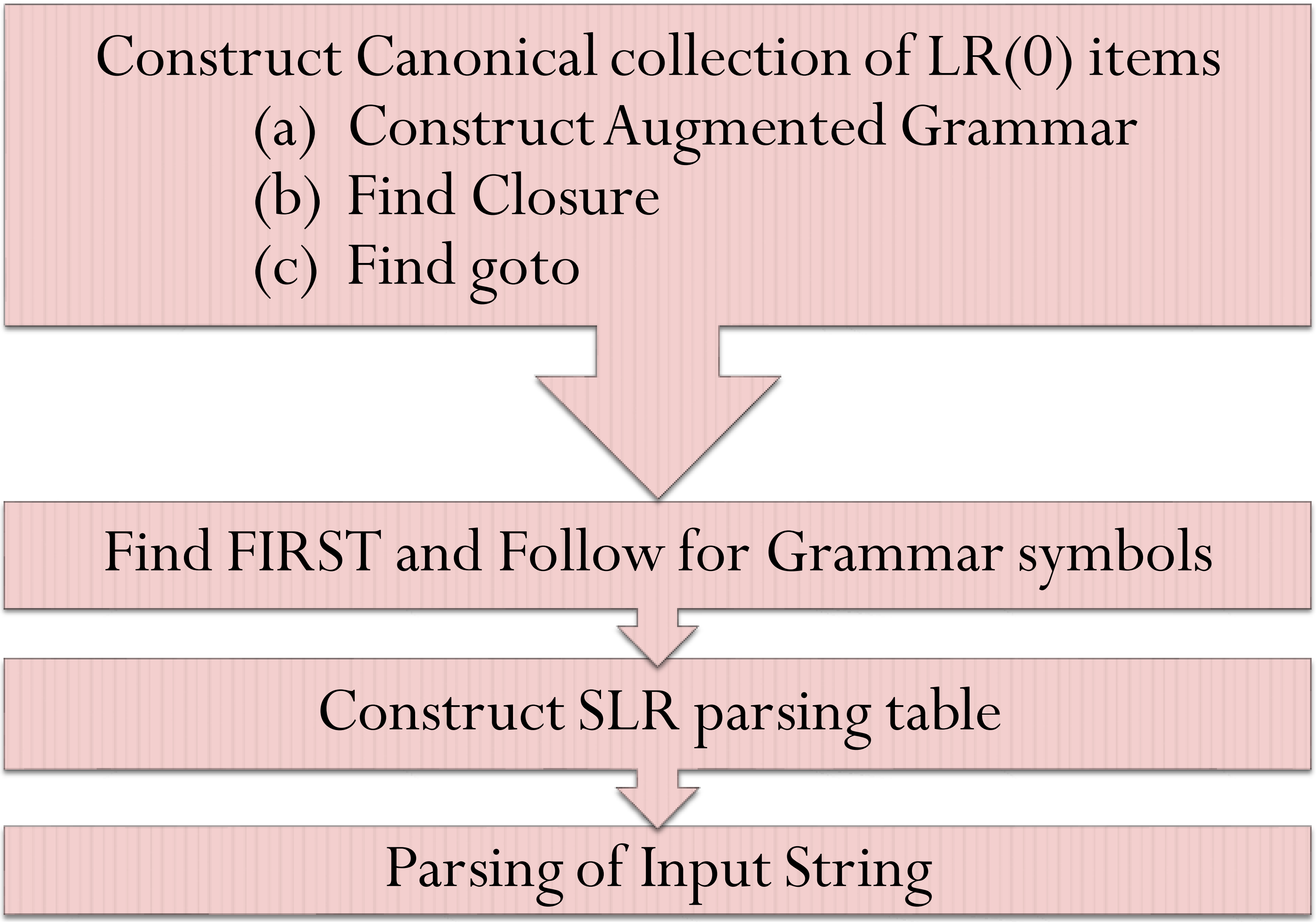
- There are basically three types of LR parser:
  - Simple LR parser (SLR)
    - Easy to implement.
    - Fails to produce table for some classes of grammars.
  - Canonical LR parser (CLR)
    - Most powerful.
    - Work on large classes of grammar.
  - Look Ahead LR parser (LALR)
    - Intermediate in power between SLR and CLR.



Comparison

	SLR Parser	**LALR Parser**	**CLR Parser**
1.	Easy and cheap to implement.	Easy to cheap and implement.	Expensive and difficult to implement.
2.	SLR parser is smallest in size.	LALR and SLR have same size. Less number of states.	CLR is largest in size. Very large number of states.
3.	Error detection is not immediate.	Error detection is not immediate.	Error detection can be done immediately.
4.	Fails to produce parsing table for certain class of grammars.	Intermediate in power between SLR and CLR.	Very powerful and work on large class of grammar.
5.	Requires less time and space complexity.	Requires more time and space complexity.	Requires more time and space complexity.

Working of SLR Parser


Construct Canonical collection of LR(0) items
(a) Construct Augmented Grammar
(b) Find Closure
(c) Find goto
Find FIRST and Follow for Grammar symbols
Construct SLR parsing table
Parsing of Input String

Canonical collection of LR(0) items

- The LR(0) item for grammar G consist of a production in which symbol dot (.) is inserted at some position on RHS of production.
- E.g. for production S-> ABC, generated LR(0) items will be:
  - S-> . ABC
  - S-> A . BC
  - S-> AB . C
  - S-> ABC .

- Canonical LR(0) collection helps to construct LR parser called Simple LR (SLR) parser.

Construction of Canonical LR(0) grammar

- Three things are required for constructing LR(0) grammar:
  - Augmented Grammar
    - if grammar G has start symbol S, augmented grammar is new grammar G' with new start symbol S'.
    - It will contain the production S'->S

  - Closure function
    - For a CFG G, if I is the set of items or states of grammar G then,
      - Every item in I is in closure (I).
      - If rule A-> α . B β is a rule in closure (I) and there is another rule for B such as B-> γ then
        - Closure (I) will consist of A-> α . B β and B-> . γ

  - goto function
    - If there is a production A-> α . X β in I then goto (I,X) is defined as a closure of set of items of A-> α . X β
      - Where I is the set of items
      - X is grammar symbol (non-terminal)

I_0: E’ -> .E
E -> .E+T
E -> .T
T -> .T*F
T -> .F
F -> .(E)
F -> .id

I_1 = goto(I_0, E)
E’ -> E.
E -> E.+T

I_2 = goto(I_0, T)
E -> T.
T -> T.*F

I_3 = goto(I_0, F)
T -> F.

I4= goto(I_0, ( )
F -> (.E)
E -> .E+T
E -> .T
T -> .T*F
T -> .F
F -> .(E)
F -> .id

I_5 = goto(I_0, id)
F-> id.

I_6 = goto(I_1, +)
E -> E+.T
T -> .T*F
T -> .F
F -> .(E)
F -> .id

I_7 = goto(I_2, *)
T -> T*.F
F -> .(E)
F -> .id

I_8 = goto(I_4, E)
F -> (E.)
E -> E.+T

I_2 = goto(I4, T)
E -> T.
T -> T.*F

I_3 = goto(I4, F)
T -> F.

I_9 = goto(I_6, T)
E -> E+T.
E -> .T
T -> T.*F

I3 = goto(I_6, F)
T -> F.

I_5 = goto(I_4, id)
F -> id.

I_4= goto(I_4, ( )
F -> (.E)
E -> .E+T
E -> .T
T -> .T*F
T -> .F
F -> .(E)
F -> .id

I4= goto(I_6, ( )
F -> (.E)
E -> .E+T
E -> .T
T -> .T*F
T -> .F
F -> .(E)
F -> .id

I3 = goto(I_6, F)
T -> F.

I_4= goto(I_7, ( )

I4= goto(I_6, ( )
F -> (.E)
E -> .E+T
E -> .T
T -> .T*F
T -> .F
F -> .(E)
F -> .id

I_5 = goto(I_6, id)
F -> id.

I_{10} = goto(I_7, F)
T -> T*F.

I_5 = goto(I_7, id)
F -> id.

I_{11}= goto(I_8,))
F -> (E).

I_6 = goto(I_8, +)
E -> E+.T
T -> .T*F
T -> .F
F -> .(E)
F -> .id

I_7 = goto(I_9, *)
T -> T*.F
F -> .(E)
F -> .id

Computing FOLLOW

- FOLLOW (E) = {$, +,)}
- FOLLOW (T) = {$, +,), *}
- FOLLOW (F) = {$, +,), *}

Construction of SLR parsing table

- There are basically two parts of SLR parsing table:
  - Action
  - Goto

- Method:
  - Initially construct set of items
  - C = $\{I_0, I_1, I_2, \ldots\ldots\ldots I_n\}$ where C is a collection of LR(0) items for grammar.
  - Parsing actions are based on each item of state I_i.

Action and goto

1. If A-> α.aβ is in I_i and goto(I_i, a) = I_j
   - Then set Action [i, a] = "shift j"

2. If A-> α. Is in I_i
   - Then set Action [i, a] to reduce "A-> α" for all symbols a,
   - where a ϵ Follow(A)

3. If S'->S. is in I_i
   - Entry in the action table Action [i,$] = "accept".

4. Goto transitions for state I is considered for non-terminals only.
   - If goto (I_i, A) = I_j then goto [i, A] = j.

5. All entries defined by rule 2 and rule 3 are considered to be "error".

EXAMPLE

- Consider the grammar

0	E’ -> E
1	E -> E + T
2	E -> T
3	T -> T * F
4	T -> F
5	F -> (E)
6	F -> id

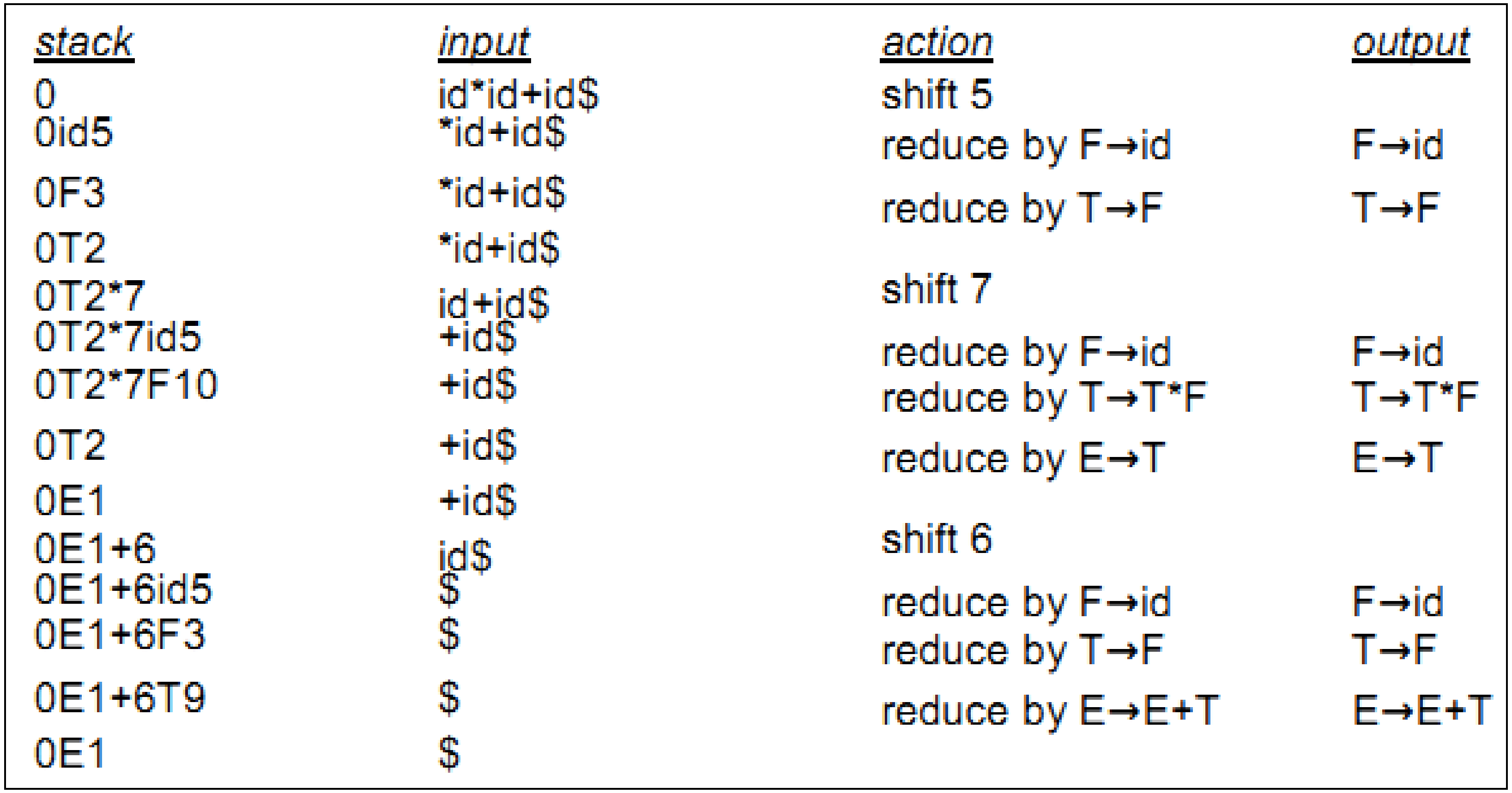
SLR Parsing table

State	Action						Goto		
	id	+	*	(	)	$	E	T	F
0									
1									
2									
3									
4									
5									
6									
7									
8									
9									
10									
11									

SLR Parsing Table

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

Parsing input string **id*id+id**

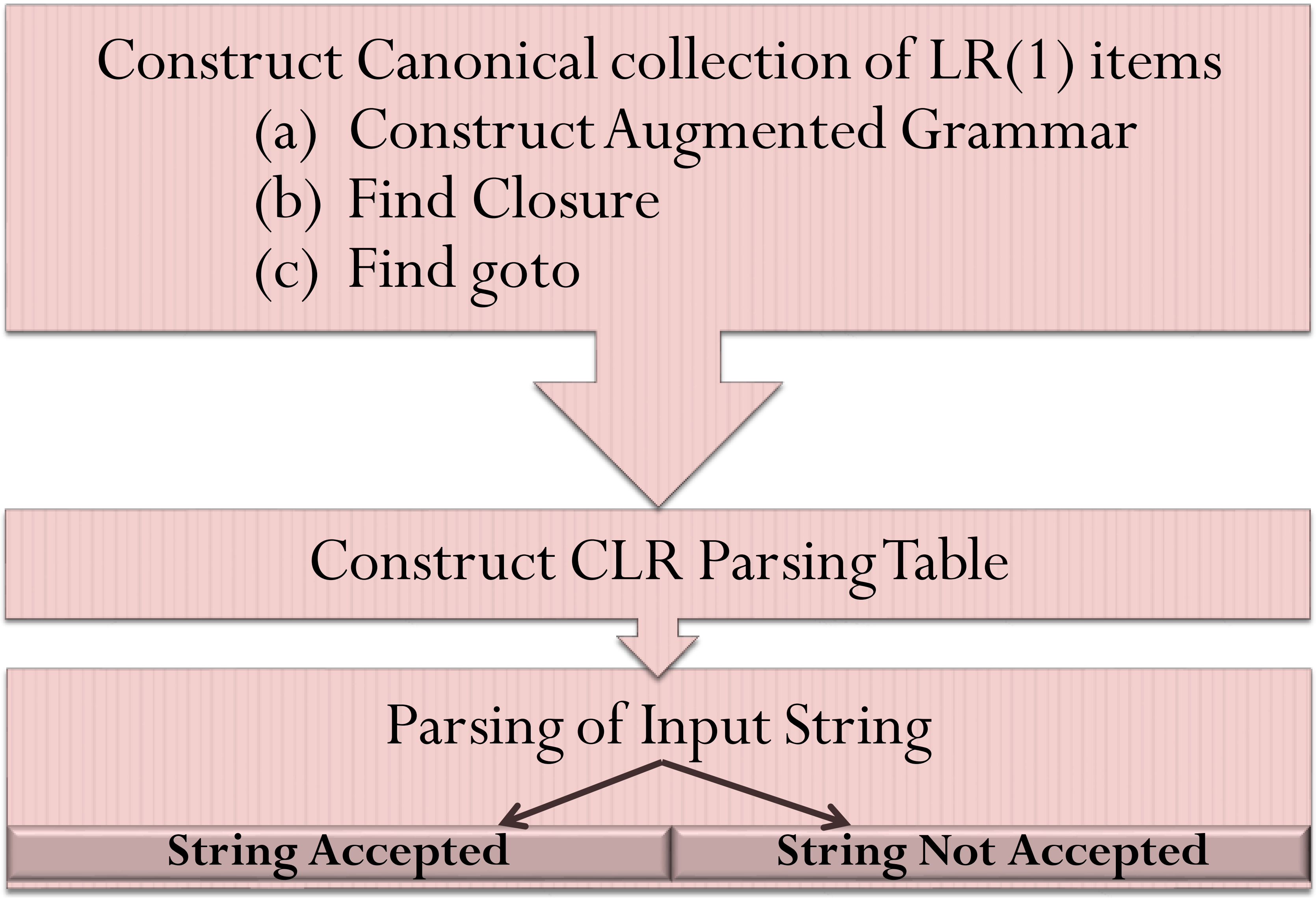


stack	input	action	output
0	id*id+id$	shift 5	
0id5	*id+id$	reduce by F→id	F→id
0F3	*id+id$	reduce by T→F	T→F
0T2	*id+id$		
0T2*7	id+id$	shift 7	
0T2*7id5	+id$	reduce by F→id	F→id
0T2*7F10	+id$	reduce by T→T*F	T→T*F
0T2	+id$	reduce by E→T	E→T
0E1	+id$		
0E1+6	id$	shift 6	
0E1+6id5	$	reduce by F→id	F→id
0E1+6F3	$	reduce by T→F	T→F
0E1+6T9	$	reduce by E→E+T	E→E+T
0E1	$		

CLR (Canonical LR Parser)

- SLR fails for unambiguous grammar.
- To overcome this limitation, CLR can be used.
- It parses the string using a look ahead symbol generated while constructing set of items.
- Therefore, the collection of set of items is called LR (1)

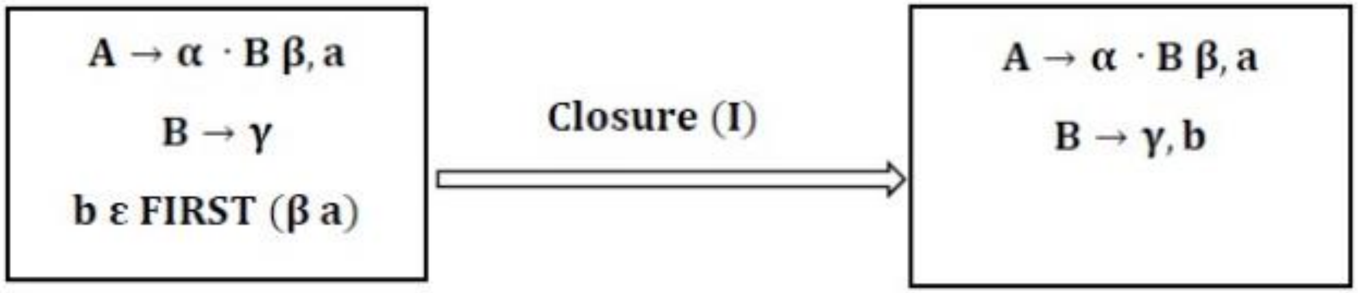
Working of CLR Parser


Construct Canonical collection of LR(1) items
(a) Construct Augmented Grammar
(b) Find Closure
(c) Find goto
Construct CLR Parsing Table
Parsing of Input String
String Accepted
String Not Accepted

Construction of Canonical LR(1) grammar

- Three things are required for constructing LR(1) grammar:
  - Augmented Grammar
    - if grammar G has start symbol S, augmented grammar is new grammar G' with new start symbol S'.
    - It will contain the production **S'->S**
  - Closure function
    - **for** each item $A \rightarrow \alpha \cdot B\, \beta, a$ in I,

```
      each production  $B \rightarrow \gamma$ and
      each terminal  $b \in FIRST\ (\beta\ a)$
      If $B \rightarrow \cdot\ \gamma$ is not in I
          Add $B \rightarrow \cdot\ \gamma, b$ to I
```



$A \rightarrow \alpha \cdot B\, \beta, a$
$B \rightarrow \gamma$
$b\ \varepsilon$ FIRST $(\beta\ a)$
Closure (I)
$A \rightarrow \alpha \cdot B\, \beta, a$
$B \rightarrow \gamma, b$

Construction of Canonical LR(1) grammar

- goto function
  - If there is a production **A-> α . X β , a** in I
  - then **goto (I,X)** is closure of set of items of **A-> α . X β , a**

Construction of Canonical LR Parsing Table Algorithm

- **Input :** An Augmented Grammar G.′
- **Output :** CLR Parsing Table
- **Method**

1. Initially construct set of items $C = \{I_0, I_1, I_2 \ldots \ldots I_n\}$ where C is a collection of LR (1) items for G.
2. Parsing actions are based on each item or state I_1.

   Various Actions are −

   a) If $A \rightarrow \alpha \cdot a\ \beta$ is in I_i and goto (I_i, a) = Ij then create an entry in Action table Action [i, a] = shift j".

   b) If $A \rightarrow \alpha\ \cdot$, a is in I_i then set in Action table Action [i, a] to reduce $A \rightarrow \alpha$. " Here, A should not be S′.

   c) If $S' \rightarrow S \cdot$ is in I_i then Action [i, \$] = accept".
3. The goto part of the SLR table can be filled as −

   If goto (I_i, A) = I_j then goto [i, A] = j
4. All entries not defined by rules 2 and 3 are considered to be "error. "

LALR (Look Ahead LR Parser)

- LALR Parser is Look Ahead LR Parser.
- It is intermediate in power between SLR and CLR parser.
- It is the compaction of CLR Parser.
  - Therefore, tables obtained in this will be smaller than CLR Parsing Table.

- First of all, LR (1) items are constructed.
- Next, we will look for the items having the same first component
- These items are merged to form a single set of items.
- The states have the same first component, but the different second component can be integrated into a single state or item.

Working of LALR Parser

Construction of set of LR(1) items with look ahead

(a) Construct Augmented Grammar

(b) Find Closure

(c) Find goto

Merge the items or states with same first component but different look ahead

Construct Parsing Table

Parse the Input String

String Accepted | **String Not Accepted**

For Example

- Suppose if

  I_4: $C \rightarrow d \cdot , c \mid d$

  I_7: $C \rightarrow d \cdot , \$$

  In I_4 and I_7,
  - First component in both items: $d\cdot$
  - Second component:
    - I_4 : $c \mid d$
    - in I_7: $\$$

  These states can be merged to give

  I_{47}: $C \rightarrow d \cdot , c \mid d \mid \$$