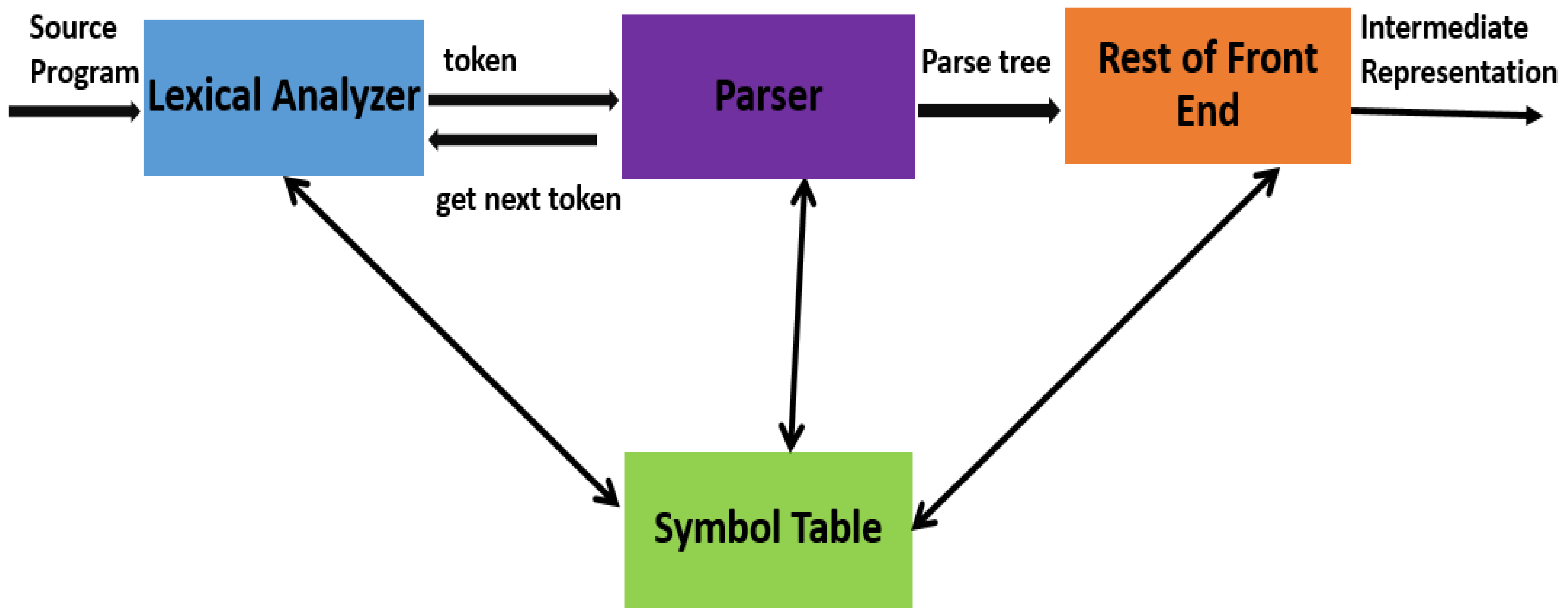
Unit-II

Syntax Analysis

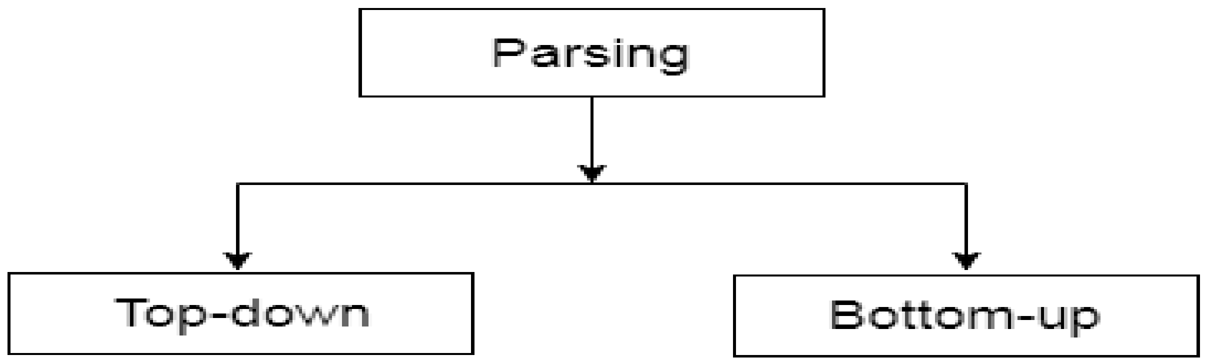
Syntax Analysis

- Syntax Analysis or Parsing is the second phase after lexical analysis.
- It checks the syntactical structure of the given input, i.e. whether the given input is in the correct syntax or not by building a data structure, called a Parse tree or Syntax tree.
- The parse tree is constructed by using the pre-defined Grammar of the language and the input string.
- If the given input string can be produced with the help of the syntax tree (in the derivation process), the input string is found to be in the correct syntax. If not, error is reported by syntax analyzer.

Syntax Analysis



Types of Parsers



- As implied by their names, top-down methods build parse trees from the top (root) to the bottom (leaves),.
- While bottom-up methods start from the leaves and work their way up to the root.
- In either case, the input to the parser is scanned from left to right, one symbol at a time.

Context Free Grammar

Context Free Grammar is a formal grammar which is used to generate all possible strings in a given formal language.

Context free grammar G can be defined by four tuples as:

$$\boxed{G = (V, T, P, S)}$$

Where,

G describes the grammar
T describes a finite set of terminal symbols.
V describes a finite set of non-terminal symbols
P describes a set of production rules
S is the start symbol.

Context Free Grammar Example

Example:

L= {wcw^R | w € (a, b)*}

Production rules:

1. $S \rightarrow aSa$
2. $S \rightarrow bSb$
3. $S \rightarrow c$

Now check that abbcbba string can be derived from the given CFG.

1. $S \Rightarrow aSa$
2. $S \Rightarrow abSba$
3. $S \Rightarrow abbSbba$
4. $S \Rightarrow abbcbba$

By applying the production $S \rightarrow aSa$, $S \rightarrow bSb$ recursively and finally applying the production $S \rightarrow c$, we get the string abbcbba.

Context Free Grammar Example

Example2: What kind of strings does the following grammar generate?

Consider a grammar G = (V , T , P , S) where-

V = { S }

T = { a , b }

P = { $S \rightarrow aSbS$, $S \rightarrow bSaS$, $S \rightarrow \in$ }

S = { S }

The above grammar generates the strings having equal number of a’s and b’s.

Derivation

Derivation is a sequence of production rules. It is used to get the input string through these production rules.

During parsing we have to take two decisions. These are as follows:

- We have to decide the non-terminal which is to be replaced.
- We have to decide the production rule by which the non-terminal will be replaced.

We have two options to decide which non-terminal to be replaced with production rule.

1. Left-most Derivation
2. Right-most Derivation

Derivation

In the **Left Most Derivation**, the input is scanned and replaced with the production rule from left to right. So we read the input string from left to right.

Example:

1. S = S + S
2. S = S - S
3. S = a | b |c

Input:

a - b + c

The left-most derivation is:

S = **S** + S ➔ S = **S** - S + S

S = a - **S** + S ➔ S = a - b + **S**

S = a - b + c

In the **Right Most Derivation**, the input is scanned and replaced with the production rule from right to left. So we read the input string from right to left.

Example:

1. S = S + S
2. S = S - S
3. S = a | b |c

Input:

a - b + c

The right-most derivation is:

S = S - **S** ➔ S = S - S + **S**

S = S - **S** + c ➔ S = **S** - b + c

S = a - b + c

Derivation Example

Find LMD and RMD for following grammar and the input string

$E \rightarrow E + E$

$E \rightarrow E * E$

$E \rightarrow id$

Input string: **id + id * id**

The left-most derivation is:

$E \rightarrow E * E$

$E \rightarrow E + E * E$

$E \rightarrow id + E * E$

$E \rightarrow id + id * E$

$E \rightarrow id + id * id$

The right-most derivation is:

$E \rightarrow E + E$

$E \rightarrow E + E * E$

$E \rightarrow E + E * id$

$E \rightarrow E + id * id$

$E \rightarrow id + id * id$

Parse Tree

- **Parse tree** is the graphical representation of symbol. The symbol can be terminal or non-terminal.
- In parsing, the string is derived using the start symbol. The root of the parse tree is the start symbol.
- The symbol in graphical representation can be terminals or non-terminals.
- Parse tree follows the precedence of operators.
- The deepest sub-tree traversed first. So, the operator in the parent node has less precedence over the operator in the sub-tree.

Parse Tree

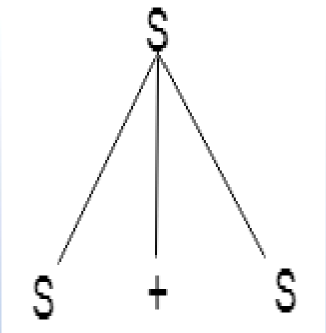
The parse tree follows these points:

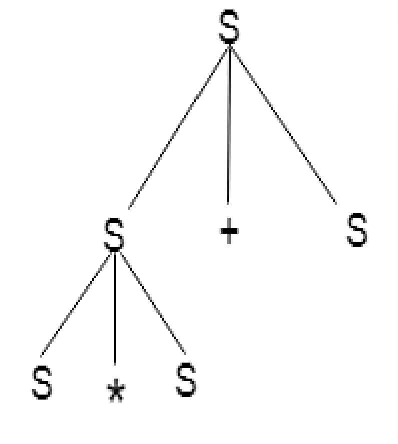
- All leaf nodes have to be terminals.
- All interior nodes have to be non-terminals.
- In-order traversal gives original input string.

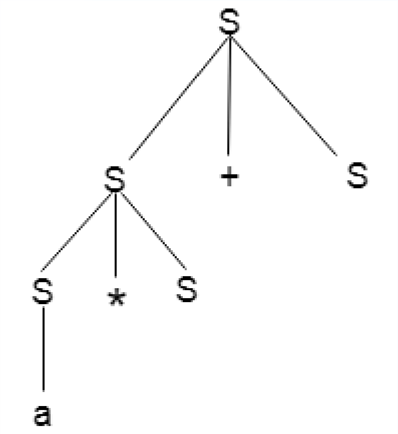
Example:

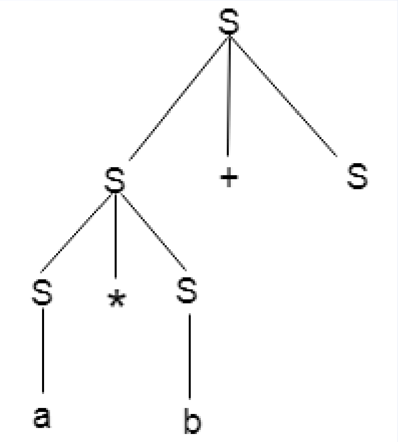
1. T= S + S | S * S
2. T = a|b|c

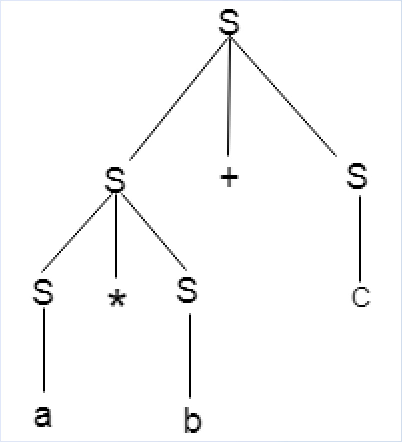
Input: a * b + c











Parse Tree Example

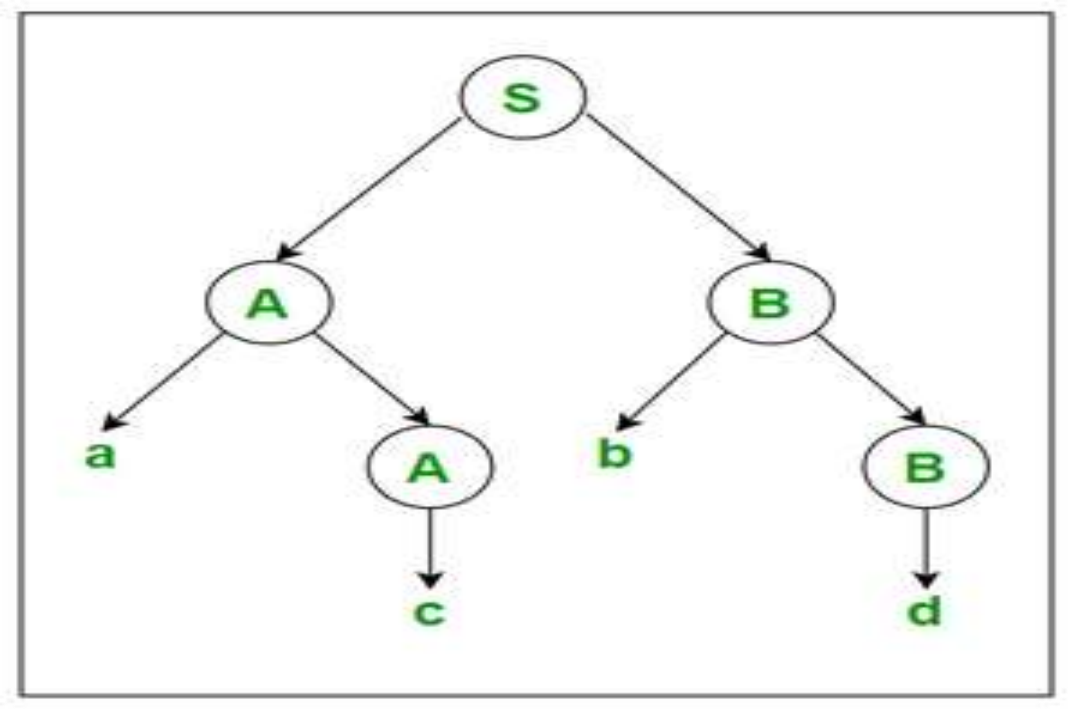
For the following production rules and input string the parse tree is generated as

S -> AB

A -> c/aA

B -> d/bB

The input string is “acbd”



Ambiguity

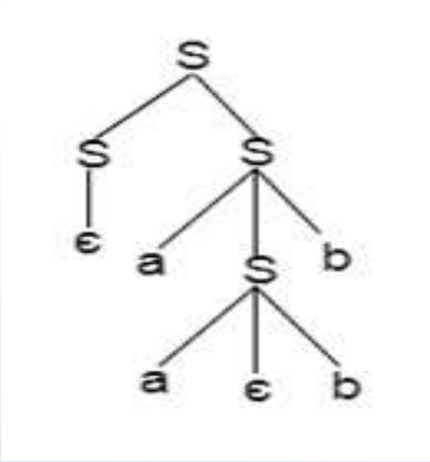
A **grammar** is said to be **ambiguous** if there exists more than one **leftmost derivation** or more than one **rightmost derivation** or more than one **parse tree** for the given input string.

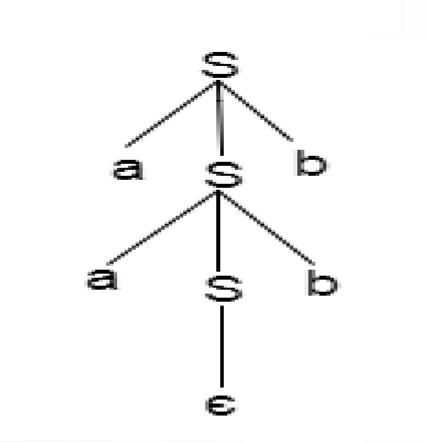
Example:

S = aSb | SS

S = ∈

For the string **aabb**, the above grammar generates two parse trees:





If the grammar has ambiguity then it is not good for a compiler construction.

Ambiguity Example

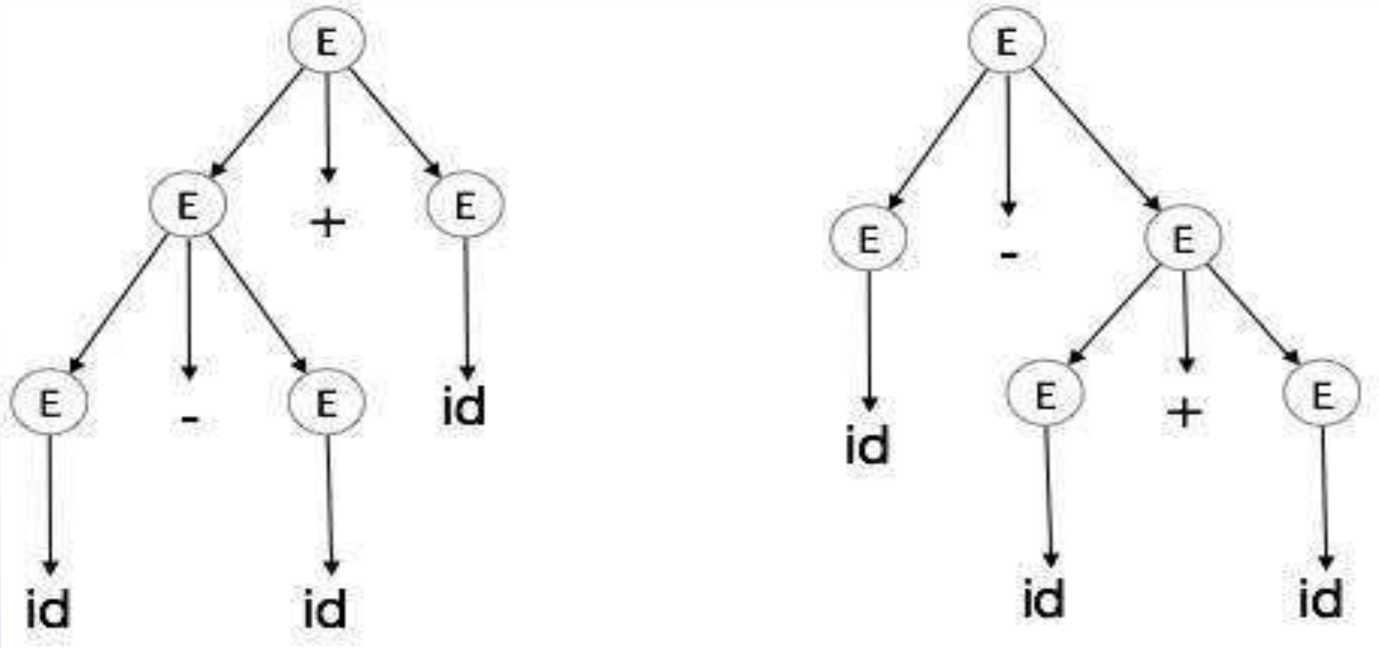
Check if the grammar is ambiguous for the following productions and input string.

$E \rightarrow E + E$

$E \rightarrow E - E$

$E \rightarrow id$

For the string **id + id – id**

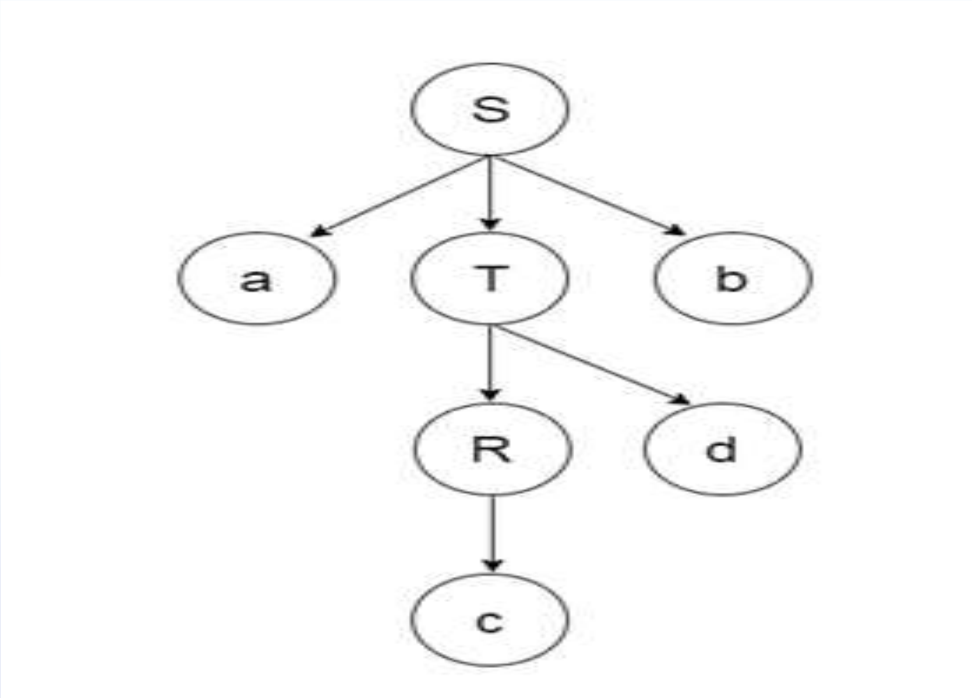


The grammar is ambiguous

Top down Parsing

- The top down parsing is known as recursive parsing or predictive parsing.
- In the top down parsing, the parsing starts from the start symbol and transform it into the input symbol.

Parse Tree representation of input string "acdb" is as follows:



Problems Associated with Top down Parsing

- Left recursion
- Left factoring
- Backtracking
- Ambiguity

Left Recursion

Let *G* be a context-free grammar. A production of *G* is said *left recursive* if it has the form

$$\boxed{A \rightarrow A\alpha \,/\beta}$$

- Here the leftmost variable of its RHS is same as variable of its LHS.
- A grammar containing a production having left recursion is called as Left Recursive Grammar.
- Left recursion is considered to be a problematic situation for Top down parsers.
- Therefore, left recursion has to be eliminated from the grammar.

Elimination of Left Recursion

In order to eliminate Left recursion, replace the following grammar

$$\boxed{A \rightarrow A\alpha \,/\, \beta}$$

with

$$\boxed{\begin{array}{l} A \rightarrow \beta A' \\ A' \rightarrow \alpha A' \,/\, \epsilon \end{array}}$$

Elimination of Left Recursion

Eliminate Left recursion from the following grammars

$A \to ABd \,/\, Aa \,/\, a$
$B \to Be \,/\, b$

$A \to aA'$
$A' \to BdA' \,/\, aA' \,/\, \epsilon$
$B \to bB'$
$B' \to eB' \,/\, \epsilon$

$E \to E + T \,/\, T$

$T \to T * F \,/\, F$

$F \to id$

$E \to TE'$

$E' \to +TE' \,/\, \epsilon$

$T \to FT'$

$T' \to *FT' \,/\, \epsilon$

$F \to id$

$E \to E + E \,/\, E * E \,/\, a$

$E \to aE'$
$E' \to +EE' \,/\, *EE' \,/\, \epsilon$

Elimination of Left Recursion

Eliminate Left recursion from the following grammars

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

↓

$S \rightarrow (L)$ / a
$L \rightarrow SL'$
$L' \rightarrow ,SL'$ / $\in$

$S \rightarrow A$
$A \rightarrow Ad$ / Ae / aB / ac
$B \rightarrow bBc$ / f

↓

$S \rightarrow A$
$A \rightarrow aBA'$ / acA'
$A' \rightarrow dA'$ / eA' / $\in$
$B \rightarrow bBc$ / f

$A \rightarrow Ba$ / Aa / c
$B \rightarrow Bb$ / Ab / d

↓

$A \rightarrow BaA'$ / cA'
$A' \rightarrow aA'$ / $\in$
$B \rightarrow cA'bB'$ / dB'
$B' \rightarrow bB'$ / aA'bB' / $\in$

Left Factoring

If RHS of more than one production starts with the same symbol, then such a grammar is called as Grammar With Common Prefixes

$$\boxed{\mathbf{A \rightarrow \alpha\beta1 \ / \ \alpha\beta2 \ / \ \alpha\beta3}}$$

(Grammar with common prefixes)

- This kind of grammar creates a problematic situation for Top down parsers.
- Top down parsers can not decide which production must be chosen to parse the string in hand.
- To remove this confusion, we use **left factoring**.

Left Factoring-

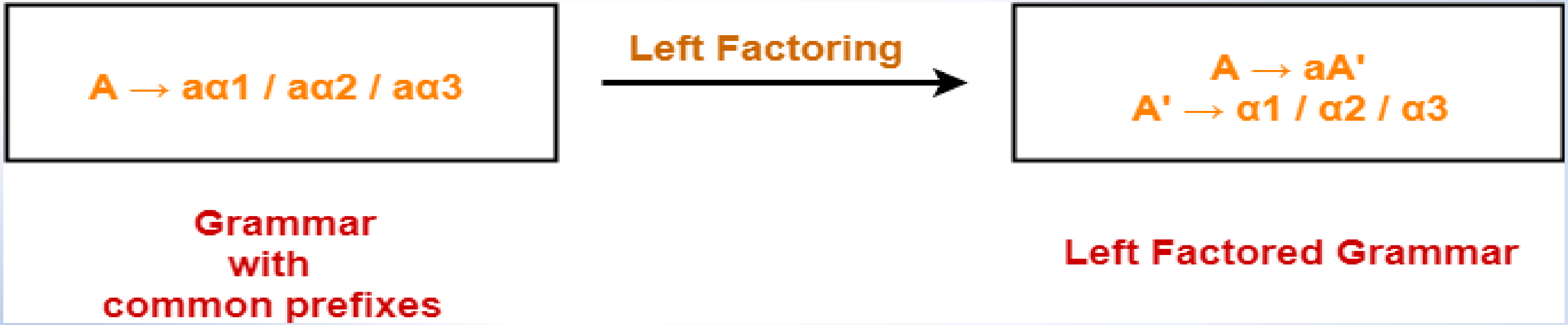
Left factoring is a process by which the grammar with common prefixes is transformed to make it useful for Top down parsers.

Left Factoring

Procedure:

- We make one production for each common prefixes.
- The common prefix may be a terminal or a non-terminal or a combination of both.
- Rest of the derivation is added by new productions.

The grammar obtained after the process of left factoring is called as **Left Factored Grammar**.



Left Factoring Examples

Do Left factoring for the following grammars

$S \to iEtS$ / $iEtSeS$ / a
$E \to b$

$S \to iEtSS'$ / a
$S' \to eS$ / ϵ
$E \to b$

$A \to aAB$ / aBc / aAc

$A \to aA'$
$A' \to AB$ / Bc / Ac

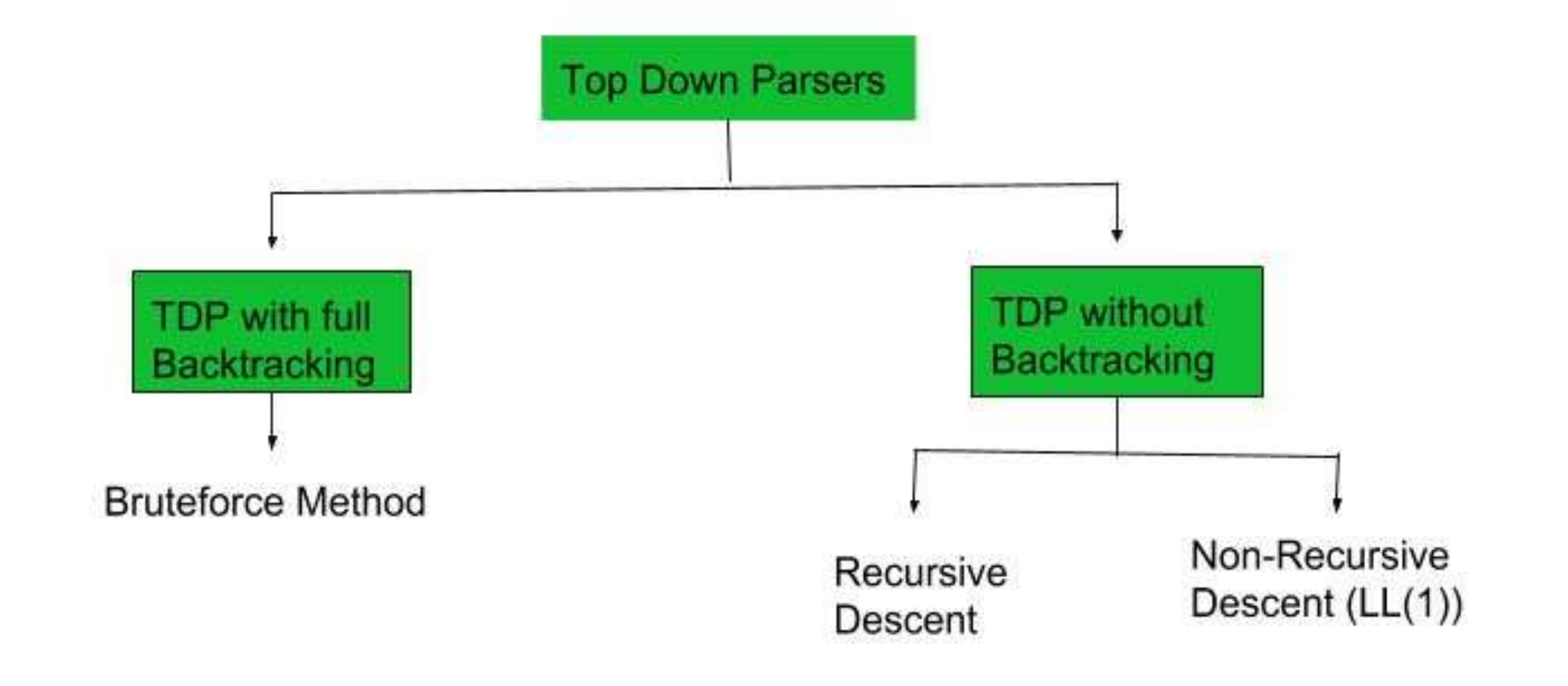
$A \to aA'$
$A' \to AD$ / Bc
$D \to B$ / c

$S \to aSSbS$ / $aSaSb$ / abb / b

$S \to aS'$ / b
$S' \to SSbS$ / $SaSb$ / bb

$S \to aS'$ / b
$S' \to SA$ / bb
$A \to SbS$ / aSb

Top down Parsing



Back Tracking

Backtracking is a systematic way of trying out different sequences of decisions until we find one that "works.“

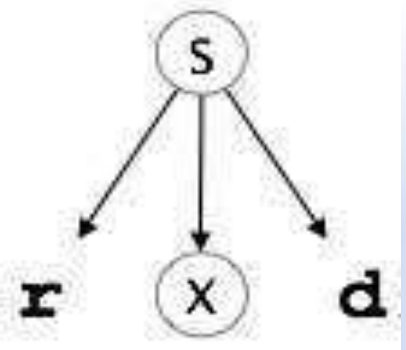
Top- down parsers start from the root node (start symbol) and match the input string against the production rules to replace them (if matched). To understand this, take the following example of CFG:

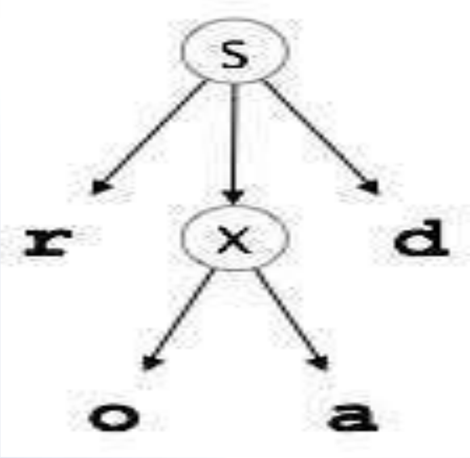
S → rXd | rZd

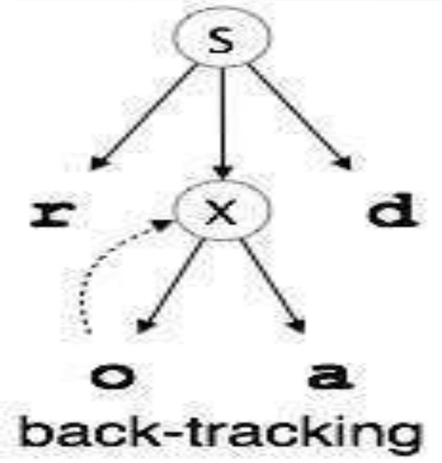
X → oa | ea

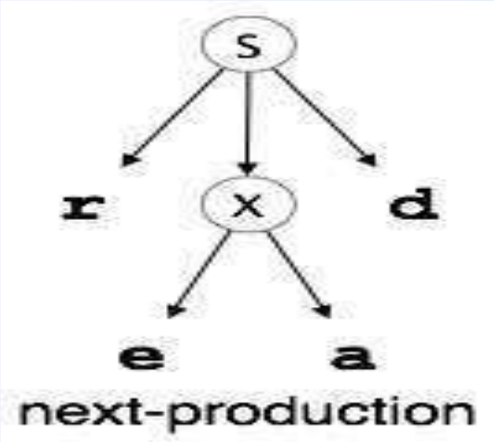
Z → ai

For an input string: **read**









Recursive Descent Parser

- It is a kind of Top-Down Parser. A top-down parser builds the parse tree from the top to down, starting with the start non-terminal.

- After eliminating left recursion and left factoring from grammar only the recursive descent parser will parse the grammar.

Recursive Descent Parser

Example:

Grammar: E --> i E'

E' --> + i E' | ε

```
E()
{
    if (l == 'i') {
        match('i');
        E'();
    }
}
```

```
match(char t)
{
    if (l == t) {
        l = getchar();
    }
    else
        printf("Error");
}
```

```
E'()
{
    if (l == '+') {
        match('+');
        match('i');
        E'();
    }
    else
        return ();
}
```

```
int main()
{
        E();
        if (l == '$')
        printf("Parsing Successful");
}
```

Non-Recursive Descent Parser

The **Predictive parsing** is a special form of recursive descent parsing, where no backtracking is required, so this can predict which production to use to replace the input string.

Non-recursive predictive parsing is also known as **LL(1) parser**. This parser follows the leftmost derivation (LMD).

LL(1):

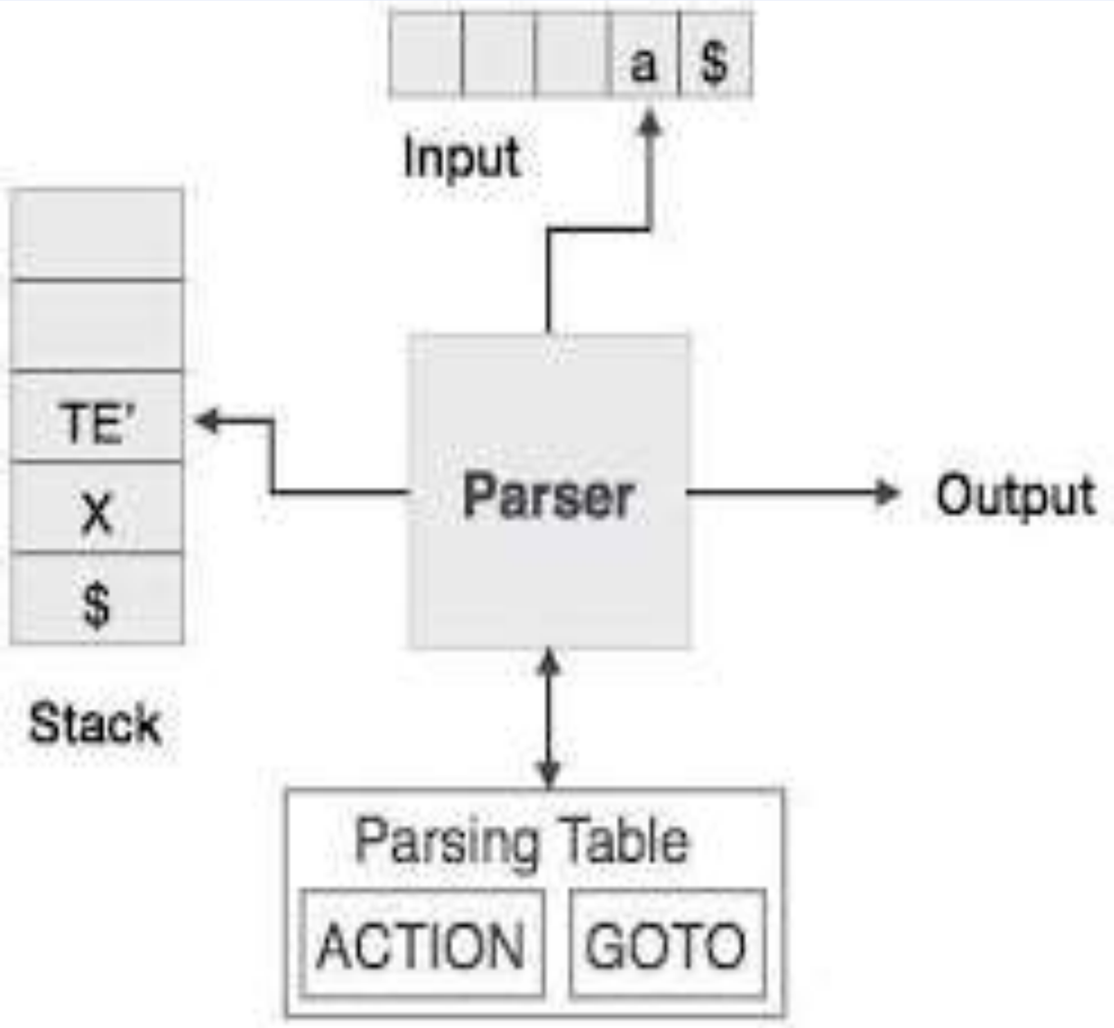
here, first L is for Left to Right scanning of inputs,

the second L is for left most derivation procedure,

1 = Number of Look Ahead Symbols

- The main problem during predictive parsing is that of determining the production to be applied for a non-terminal.
- This non-recursive parser looks up which production to be applied in a parsing table.

LL(1) Parser



The LL(1) parser has the following components:

(1) buffer: an input buffer which contains the string to be passed

(2) stack: a pushdown stack which contains a sequence of grammar symbols

(3) A parsing table: a 2d array M[A, a] where

A->non-terminal, a->terminal or $

(4) output stream:

end of the stack and an end of the input symbols are both denoted with $

Calculating First and Follow Sets

First and **Follow** sets are needed so that the parser can properly apply the needed production rule at the correct position.

First Function-

First(α) is a set of terminal symbols that begin in strings derived from α.

Example-

Consider the production rule-

A → abc / def / ghi

Then, we have-

First(A) = { a , d , g }

Rules for Calculating First Sets

Rules For Calculating First Function-

Rule-01:

For a production rule $X \rightarrow \in$,

$First(X) = \{ \in \}$

Rule-02:

For any terminal symbol 'a',

$First(a) = \{ a \}$

Rules for Calculating First Sets

Rule-03:

For a production rule $X \rightarrow Y1Y2Y3$,

Calculating First(X)

- If $\in \notin$ First(Y1), then First(X) = First(Y1)
- If $\in \in$ First(Y1), then First(X) = { First(Y1) – $\in$ } $\cup$ First(Y2Y3)

Calculating First(Y2Y3)

- If $\in \notin$ First(Y2), then First(Y2Y3) = First(Y2)
- If $\in \in$ First(Y2), then First(Y2Y3) = { First(Y2) – $\in$ } $\cup$ First(Y3)

Similarly, we can make expansion for any production rule $X \rightarrow Y1Y2Y3\ldots..Yn$.

Examples Calculating First Sets

Calculate the first functions for the given grammar-

$S \rightarrow aBDh$
$B \rightarrow cC$
$C \rightarrow bC \mid \epsilon$
$D \rightarrow EF$
$E \rightarrow g \mid \epsilon$
$F \rightarrow f \mid \epsilon$

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

$S \rightarrow A$
$A \rightarrow aB \mid Ad$
$B \rightarrow b$
$C \rightarrow g$

$S \rightarrow A$
$A \rightarrow aBA'$
$A' \rightarrow dA' \mid \epsilon$
$B \rightarrow b$
$C \rightarrow g$

$\text{First}(S) = \text{First}(A) = \{ a \}$
$\text{First}(A) = \{ a \}$
$\text{First}(A') = \{ d , \epsilon \}$
$\text{First}(B) = \{ b \}$
$\text{First}(C) = \{ g \}$

Examples Calculating First Sets

Calculate the first functions for the given grammar-

S → (L) / a
L → SL'
L' → ,SL' / ∈

First(S) = { (, a }
First(L) = First(S) = { (, a }
First(L') = { , , ∈ }

S → AaAb / BbBa
A → ∈
B → ∈

First(S) = { First(A) – ∈} ∪ First(a) ∪ { First(B) – ∈} ∪ First(b) = {a,b}
First(A) = { ∈ }
First(B) = { ∈ }

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

E → TE'
E' → + TE' / ∈
T → FT'
T' → * FT' / ∈
F → (E) / id

First(E) = First(T) = First(F) ={(, id }
First(E') = { + , ∈ }
First(T) = First(F) = { (, id }
First(T') = { * , ∈ }
First(F) = { (, id }

Rules for Calculating Follow Sets

Follow(α) is a set of terminal symbols that appear immediately to the right of α.

Rules For Calculating Follow Function-

Rule-01:

For the start symbol S, place $ in Follow(S).

Rule-02:

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

Follow(B) = Follow(A)

Rule-03:

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

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

Important Note

Note-01:

∈ will never appear in the follow function of a non-terminal.

Note-02:

Before calculating the first and follow functions, eliminate Left Recursion from the grammar, if present.

Note-03:

We calculate the follow function of a non-terminal by looking where it is present on the RHS of a production rule.

Examples for Calculating Follow Sets

Calculate the follow functions for the given grammar-

$S \rightarrow aBDh$
$B \rightarrow cC$
$C \rightarrow bC \,/\, \in$
$D \rightarrow EF$
$E \rightarrow g \,/\, \in$
$F \rightarrow f \,/\, \in$

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

$S \rightarrow A$
$A \rightarrow aB \,/\, Ad$
$B \rightarrow b$
$C \rightarrow g$

$S \rightarrow A$
$A \rightarrow aBA'$
$A' \rightarrow dA' \,/\, \in$
$B \rightarrow b$
$C \rightarrow g$

Follow(S) = { \$ }
Follow(A) = Follow(S) = { \$ }
Follow(A’) = Follow(A) = { \$ }
Follow(B) = { First(A’) – ∈ } ∪ Follow(A) = { d , \$ }
Follow(C) = NA

Examples Calculating Follow Sets

Calculate the follow functions for the given grammar-

S → (L) / a
L → SL’
L’ → ,SL’ / ∈

Follow(S)= { $ }∪{ First(L’) – ∈ }∪ Follow(L) ∪Follow(L’) ={ $, , ,) }
Follow(L) = {) }
Follow(L’) = Follow(L) = {) }

S → AaAb / BbBa
A → ∈
B → ∈

Follow(S) = { $ }
Follow(A) = First(a) ∪ First(b) = { a , b }
Follow(B) = First(b) ∪ First(a) = { a , b }

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

E → TE’
E’ → + TE’ / ∈
T → FT’
T’ → * FT’ / ∈
F → (E) / id

Follow(E) = { $,) }
Follow(E’) = Follow(E) = { $,) }
Follow(T) = { First(E’) – ∈ } ∪ Follow(E) ∪
Follow(E’) = { + , $,) }
Follow(T’) = Follow(T) = { + , $,) }
Follow(F) = { First(T’) – ∈ } ∪ Follow(T) ∪
Follow(T’) = { * , + , $,) }

Steps for Constructing LL(1) Parse Table

To construct the Parsing table, we have two functions:

- **First():** If there is a variable, and from that variable if we try to drive all the strings then the beginning *Terminal Symbol* is called the first.
- **Follow():** What is the *Terminal Symbol* which follow a variable in the process of derivation.
- Now we have to make entries into the Parse table, the Rows will contain the Non-Terminals and the column will contain the Terminal Symbols.
- All the **Null Productions** of the Grammars will go under the Follow elements and the remaining productions will lie under the elements of First set.

Steps for Constructing LL(1) Parse Table

Example: Consider the following Grammar

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

E → TE’
E’ → + TE’ / ∈
T → FT’
T’ → * FT’ / ∈
F → (E) / id

	First	Follow
E –> TE’	{ id, (}	{ $,) }
E’ –> +TE’/e	{ +, e }	{ $,) }
T –> FT’	{ id, (}	{ +, $,) }
T’ –> *FT’/e	{ *, e }	{ +, $,) }
F –> id/(E)	{ id, (}	{ *, +, $,) }

Making Entries into Parse Table

	First	Follow
E –> TE’	{ id, (}	{ $,) }
E’ –> +TE’/e	{ +, e }	{ $,) }
T –> FT’	{ id, (}	{ +, $,) }
T’ –> *FT’/e	{ *, e }	{ +, $,) }
F –> id/(E)	{ id, (}	{ *, +, $,) }

	id	+	*	(	)	$
E	E –> TE’			E –> TE’		
E’		E’ –> +TE’			E’ –> e	E’ –> e
T	T –> FT’			T –> FT’		
T’		T’ –> e	T’ –> *FT’		T’ –> e	T’ –> e
F	F –> id			F –> (E)		

Example

Consider the string **“id+id$”**

Stack	Input String	Action
$E	id + id$	Pop and Push
$E’T	id + id$	Pop and Push
$E’T’F	id + id$	Pop and Push
$E’T’id	id + id$	Pop and Bypass
$E’T’	+ id $	Pop and Push
$E’	+ id $	Pop and Push
$E’T+	+ id $	Pop and Bypass
$E’T	id $	Pop and Push
$E’T’F	id $	Pop and Push
$E’T’id	id $	Pop and Bypass
$E’T’	$	Pop and Push
$E’	$	Pop and Push
$	$	Accept

Constructing LL(1) Parse Table Example

Example: Consider the following Grammar

S --> A | a
A --> a

	First	Follow
S –> A/a	{ a }	{ $ }
A –>a	{ a }	{ $ }

	a	**$**
S	S –> A, S –> a	
A	A –> a	

Here, we can see that there are two productions into the same cell. Hence, this grammar is not feasible for LL(1) Parser.

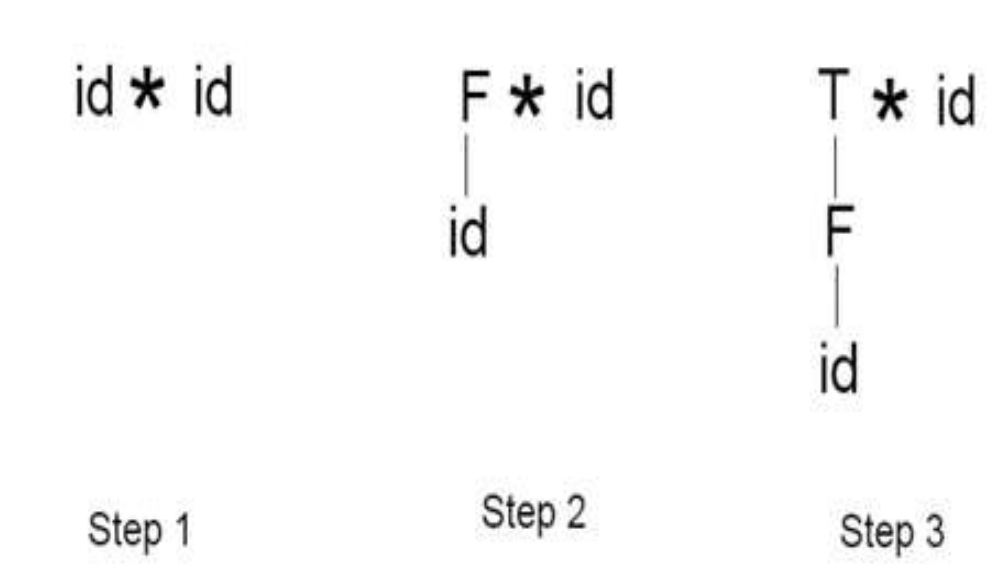
Bottom Up Parsing

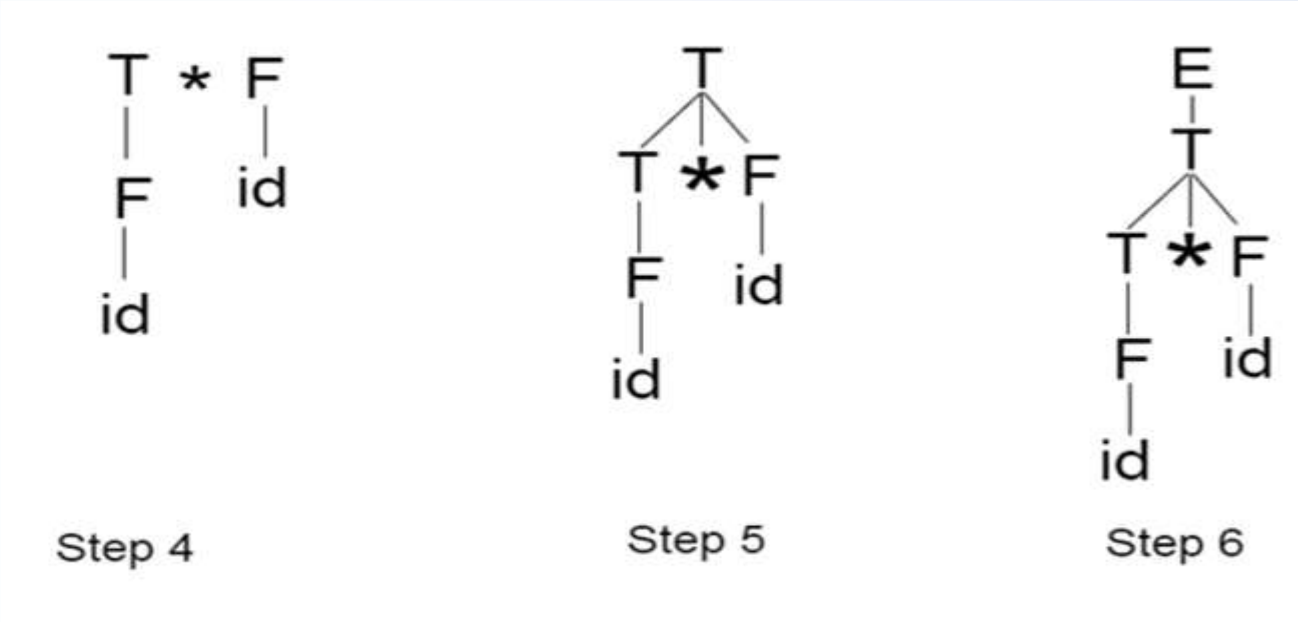
- Bottom up parsing is also known as shift-reduce parsing.
- Bottom up parsing is used to construct a parse tree for an input string.
- In the bottom up parsing, the parsing starts with the input symbol and construct the parse tree up to the start symbol by tracing out the rightmost derivations of string in reverse.

Bottom Up Parsing Example

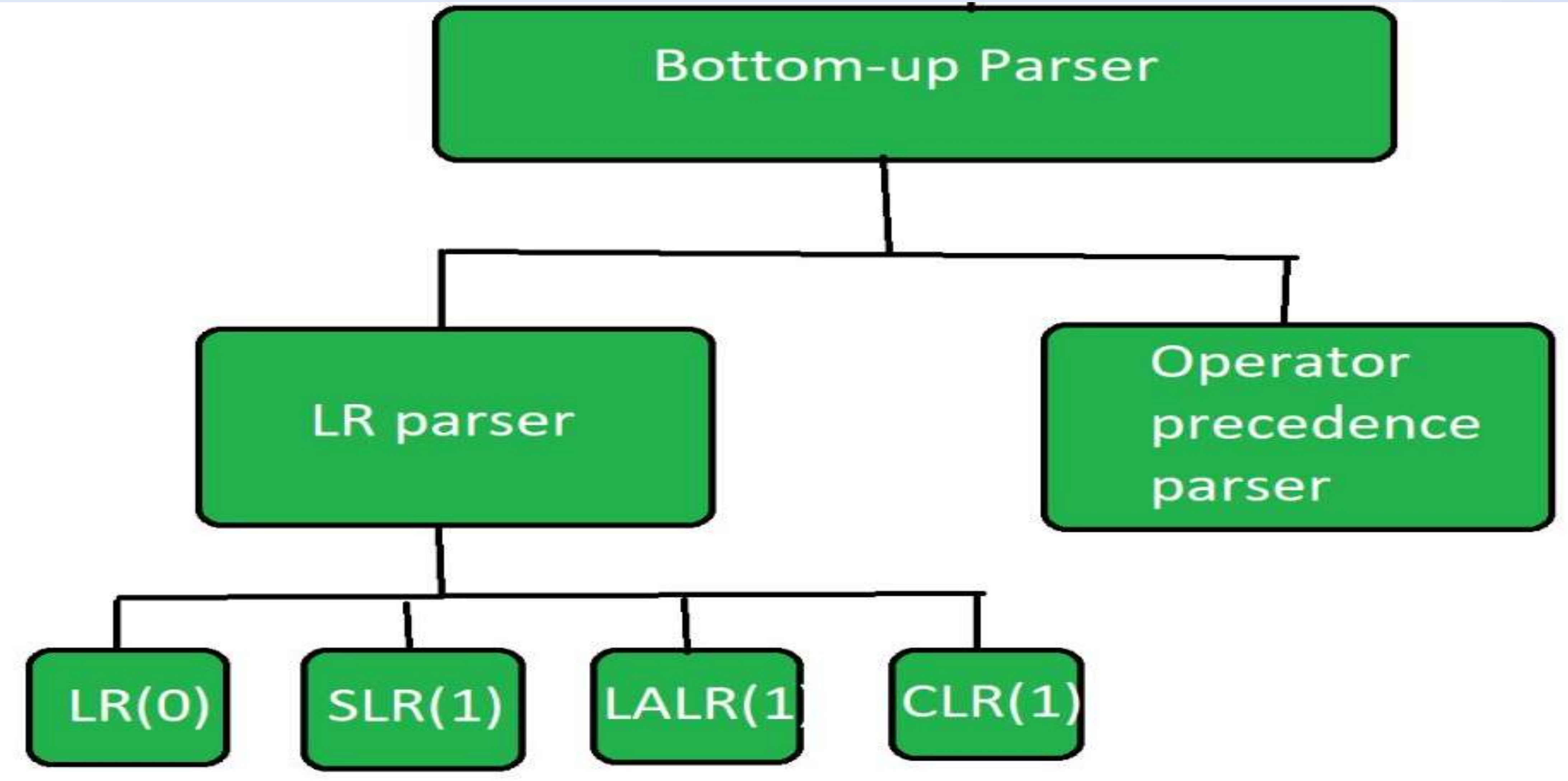
$E \rightarrow E + T \,/\, T$
$T \rightarrow T * F \,/\, F$
$F \rightarrow (E) \,/\, id$

Parse Tree representation of input string "id * id" is as follows:



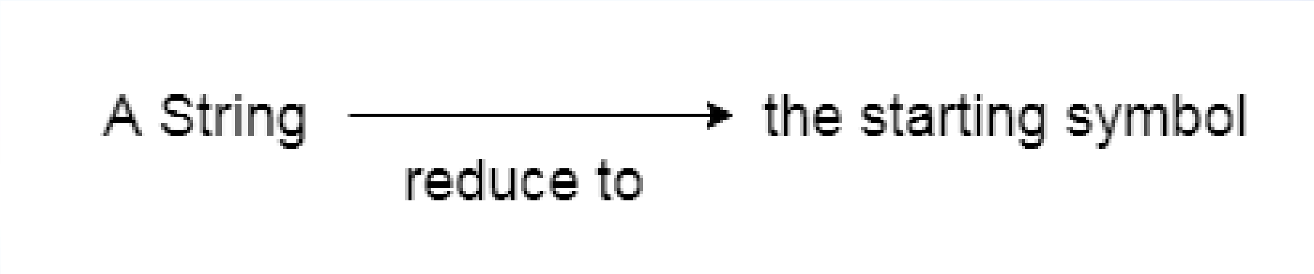


Bottom Up Parsing Classification



Shift-Reduce Parsing

- Shift reduce parsing is a process of reducing a string to the start symbol of a grammar.
- Shift reduce parsing uses a stack to hold the grammar and an input tape to hold the string.



Shift-Reduce Parsing

A shift-reduce parser can possibly make the following four actions-

1. **Shift-** In a shift action, The next symbol is shifted onto the top of the stack.

2. **Reduce-** In a reduce action, The handle appearing on the stack top is replaced with the appropriate non-terminal symbol.

3. **Accept-** In an accept action, The parser reports the successful completion of parsing.

4. **Error-** In this state,
The parser becomes confused and is not able to make any decision.
It can neither perform shift action nor reduce action nor accept action.

Shift-Reduce Parsing

Consider the following grammar-

$S \rightarrow (L) \mid a$

$L \rightarrow L , S \mid S$

Parse the input string **(a , (a , a))** using a shift-reduce parser.

Shift-Reduce Parsing

Stack	Input Buffer	Parsing Action
$	(a , (a , a)) $	Shift
$ (	a , (a , a)) $	Shift
$ (a	, (a , a)) $	Reduce S → a
$ (S	, (a , a)) $	Reduce L → S
$ (L	, (a , a)) $	Shift
$ (L ,	(a , a)) $	Shift
$ (L , (	a , a)) $	Shift
$ (L , (a	, a)) $	Reduce S → a
$ (L , (S	, a)) $	Reduce L → S
$ (L , (L	, a)) $	Shift
$ (L , (L ,	a)) $	Shift
$ (L , (L , a	)) $	Reduce S → a
$ (L , (L , S	)) $	Reduce L → L , S
$ (L , (L	)) $	Shift
$ (L , (L)	) $	Reduce S → (L)
$ (L , S	) $	Reduce L → L , S
$ (L	) $	Shift
$ (L)	$	Reduce S → (L)
$ S	$	Accept

LR Parser

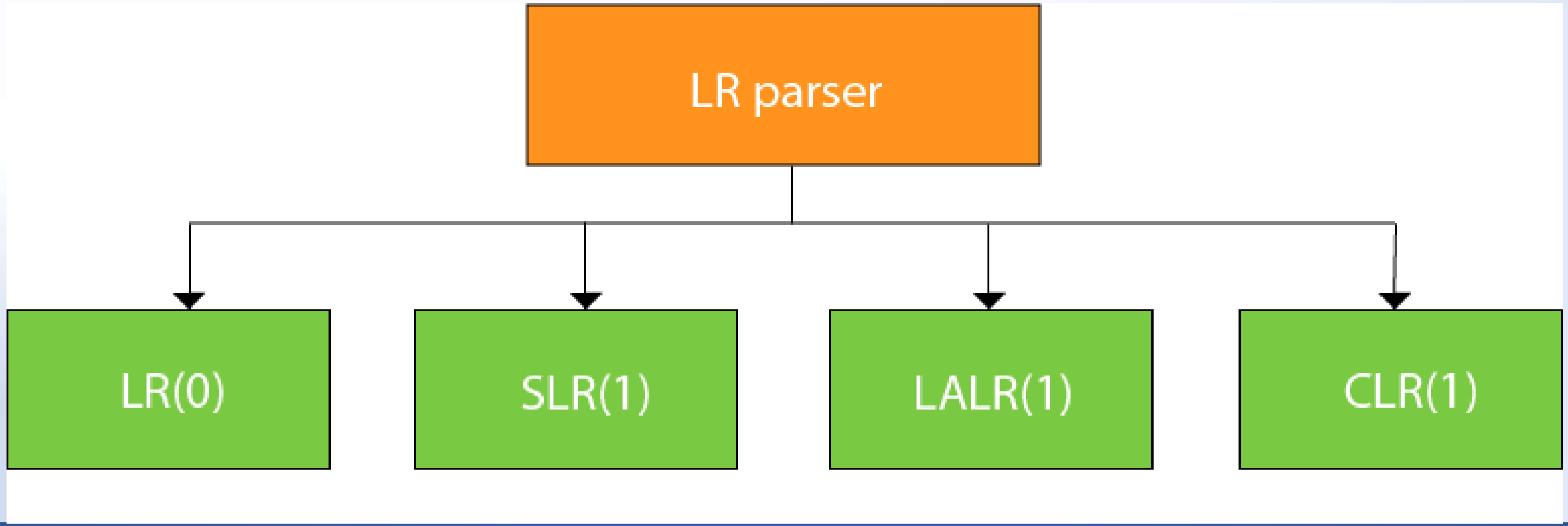
LR parsing is one type of bottom up parsing.

In the LR parsing,

"L" stands for left-to-right scanning of the input.

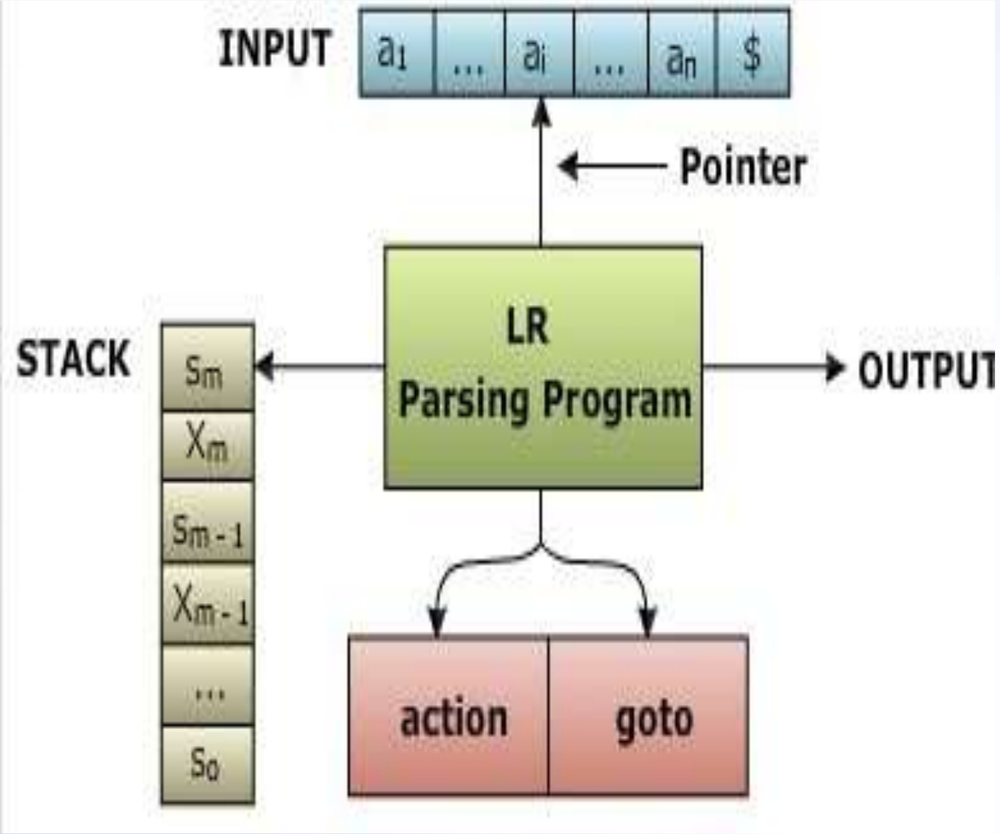
"R" stands for constructing a right most derivation in reverse.

LR parsing is divided into four parts: LR (0) parsing, SLR parsing, CLR parsing and LALR parsing.



LR Algorithm:

The LR algorithm requires stack, input, output and parsing table. In all type of LR parsing, input, output and stack are same but parsing table is different.



- Input buffer contains the string to be parsed and it is followed by a $ Symbol.
- A stack is used to contain a sequence of grammar symbols with a $ at the bottom of the stack.
- Parsing table is a two dimensional array. It contains two parts: Action part and Go To part.

LR(1) Parsing

Various steps involved in the LR (1) Parsing:

- For the given input string write a context free grammar.
- Check the ambiguity of the grammar.
- Add Augment production in the given grammar.
- Create Canonical collection of LR (0) items.
- Draw a data flow diagram (DFA).
- Construct a LR (1) parsing table.

Augmented Grammar

Augmented grammar G’ will be generated if we add one more production in the given grammar G.

It helps the parser to identify when to stop the parsing and announce the acceptance of the input.

Example: For a given grammar

$S \rightarrow AA$

$A \rightarrow aA \mid b$

The Augment grammar G’ is represented by

$S' \rightarrow S$

$S \rightarrow AA$

$A \rightarrow aA \mid b$

Canonical Collection of LR(0) items

- An LR (0) item is a production G with dot at some position on the right side of the production.
- LR(0) items is useful to indicate that how much of the input has been scanned up to a given point in the process of parsing.
- In the LR (0), we place the reduce node in the entire row.

Example

Given grammar:

$S \rightarrow AA$
$A \rightarrow aA \mid b$

Add Augmented Production and insert '•' symbol at the first position for every production in G.

1. $S` \rightarrow \bullet S$
2. $S \rightarrow \bullet AA$
3. $A \rightarrow \bullet aA$
4. $A \rightarrow \bullet b$

I0 State:

Add Augment production to the I0 State and Compute the Closure

I0 = Closure ($S` \rightarrow \bullet S$)

Add all productions starting with S in to I0 State because "•" is followed by the non-terminal. So, the I0 State becomes

I0 = $S` \rightarrow \bullet S$
$S \rightarrow \bullet AA$

Add all productions starting with "A" in modified I0 State because "•" is followed by the non-terminal. So, the I0 State becomes.

I0= $S` \rightarrow \bullet S$
$S \rightarrow \bullet AA$
$A \rightarrow \bullet aA$
$A \rightarrow \bullet b$

I0= $S` \rightarrow \bullet S$

$S \rightarrow \bullet AA$

$A \rightarrow \bullet aA$

$A \rightarrow \bullet b$

I1= Go to (I0, S) = closure ($S` \rightarrow S\bullet$) = $S` \rightarrow S\bullet$

I2= Go to (I0, A) = closure ($S \rightarrow A\bullet A$)

I2 =$S\rightarrow A\bullet A$

$A \rightarrow \bullet aA$

$A \rightarrow \bullet b$

I3= Go to (I0,a) = Closure ($A \rightarrow a\bullet A$)

I3= $A \rightarrow a\bullet A$

$A \rightarrow \bullet aA$

$A \rightarrow \bullet b$

I4= Go to (I0, b) = closure ($A \rightarrow b\bullet$) = $A \rightarrow b\bullet$

I5= Go to (I2, A) = Closure ($S \rightarrow AA\bullet$) = $S \rightarrow A A\bullet$

Go to(I2,a) = Closure($A \rightarrow a\bullet A$) = **I3**

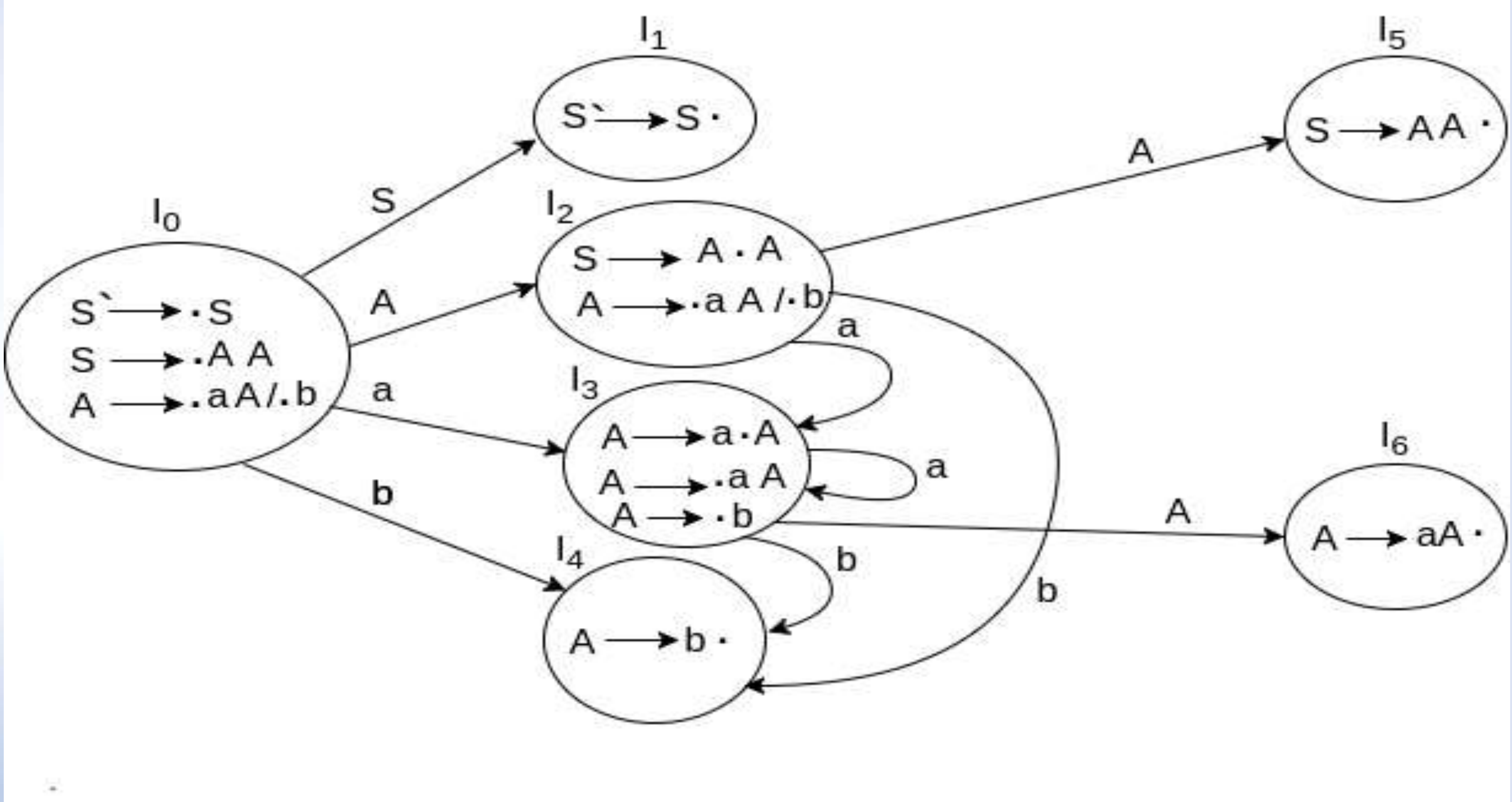
Go to(I2,b) = closure ($A \rightarrow b\bullet$) = **I4**

I6= Go to (I3, A) = Closure ($A \rightarrow aA\bullet$)= $A \rightarrow aA\bullet$

Go to (I3, a) = Closure ($A \rightarrow a\bullet A$) = **I3**

Go to (I3, b) = Closure ($A \rightarrow b\bullet$) = **I4**

Data Flow Diagram



LR(0) Table

1. Construct F = {I0, I1, …In}
2. a) if $\{A \rightarrow \alpha\bullet\}$ ε Ii and A != S' then action[i, _] := **reduce A → α**

   b) if $\{S' \rightarrow S\bullet\}$ ε Ii then action[i,$] := **accept**

   c) if $\{A \rightarrow \alpha \bullet a\beta\}$ ε Ii and Successor(Ii ,a) = Ij then action[i,a] := **shift j**

3. if Successor(Ii ,A) = Ij then goto[i,A] := **j**

LR(0) Table

States	Action			Go to	
	a	b	$	A	S
I_0	S3	S4		2	1
I_1			accept		
I_2	S3	S4		5	
I_3	S3	S4		6	
I_4	r3	r3	r3		
I_5	r1	r1	r1		
I_6	r2	r2	r2		

SLR(1) Parsing

SLR (1) refers to simple LR Parsing. It is same as LR(0) parsing. The only difference is in the parsing table.

To construct SLR (1) parsing table, we use canonical collection of LR (0) item.

In the SLR (1) parsing, we place the reduce move only in the follow of left hand side.

Various steps involved in the SLR (1) Parsing:

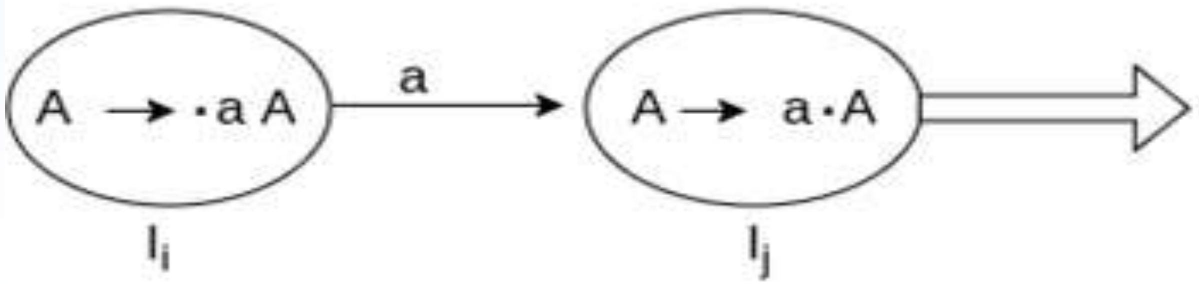
- For the given input string write a context free grammar
- Check the ambiguity of the grammar
- Add Augment production in the given grammar
- Create Canonical collection of LR (0) items
- Draw a data flow diagram (DFA)
- Construct a SLR (1) parsing table

SLR(1) Table Construction

SLR (1) Table Construction

The steps which use to construct SLR (1) Table is given below:

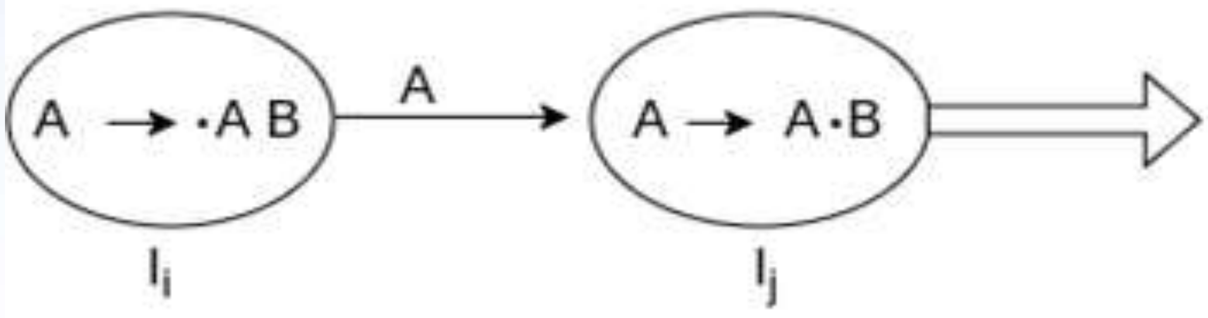
If a state (I_i) is going to some other state (I_j) on a terminal then it corresponds to a shift move in the action part.



States	Action		Go to
	a	$	A
I_i	Sj		
I_j			

SLR(1) Table Construction

If a state (I_i) is going to some other state (I_j) on a variable then it correspond to go to move in the Go to part.



States	Action		Go to
	a	$	A
I_i			j
I_j			

SLR(1) Table Construction

If a state (I_i) contains the final item like A → ab• which has no transitions to the next state then the production is known as reduce production. For all terminals X in FOLLOW (A), write the reduce entry along with their production numbers.

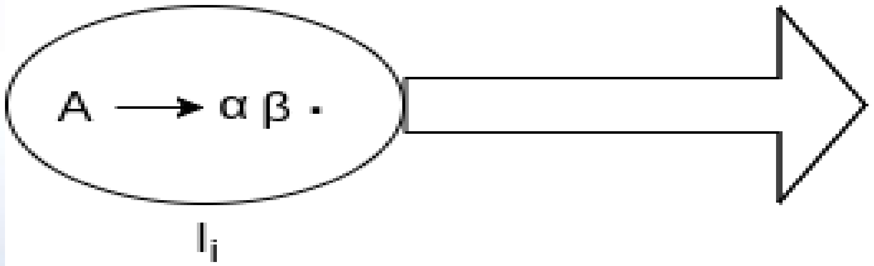
Example:

S -> •Aa

A->αβ•

Follow(S) = {$}

Follow (A) = {a}



States	Action			Go to	
	a	b	$	S	A
I_i	r2				

SLR(1) Table Construction

Construct SLR (1) Parse Table for the following Grammar

$S \rightarrow E$

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow id$

Add Augment Production and insert '•' symbol at the first position for every production in G

$S` \rightarrow \bullet E$

$E \rightarrow \bullet E + T$

$E \rightarrow \bullet T$

$T \rightarrow \bullet T * F$

$T \rightarrow \bullet F$

$F \rightarrow \bullet id$

SLR(1) Table Construction

I0 State: Add Augment production to the I0 State and Compute the Closure

I0 = Closure (S\` → •E)

Add all productions starting with E in to I0 State because "." is followed by the non-terminal. So, the I0 State becomes

I0 = S\` → •E
E → •E + T
E → •T

Add all productions starting with T and F in modified I0 State because "." is followed by the non-terminal. So, the I0 State becomes.

I0= S\` → •E
E → •E + T
E → •T
T → •T * F
T → •F
F → •id

SLR(1) Table Construction

VISHNU UNIVERSAL LEARNING

I0= S` → •E
E → •E + T
E → •T
T → •T * F
T → •F
F → •id

I5= Go to (I1, +)
= Closure (E → E +•T)
I5 = E → E +•T
T → •T * F
T → •F
F → •id

I6= Go to (I2, *) = Closure (T → T * •F)
I6 = T → T * •F
F → •id

I1= Go to (I0, E) = closure (S` → E•, E → E• + T)

I2= Go to (I0, T) = closure (E → T•,T → T• * F)

I3= Go to (I0, F) = Closure (T → F•) = T → F•

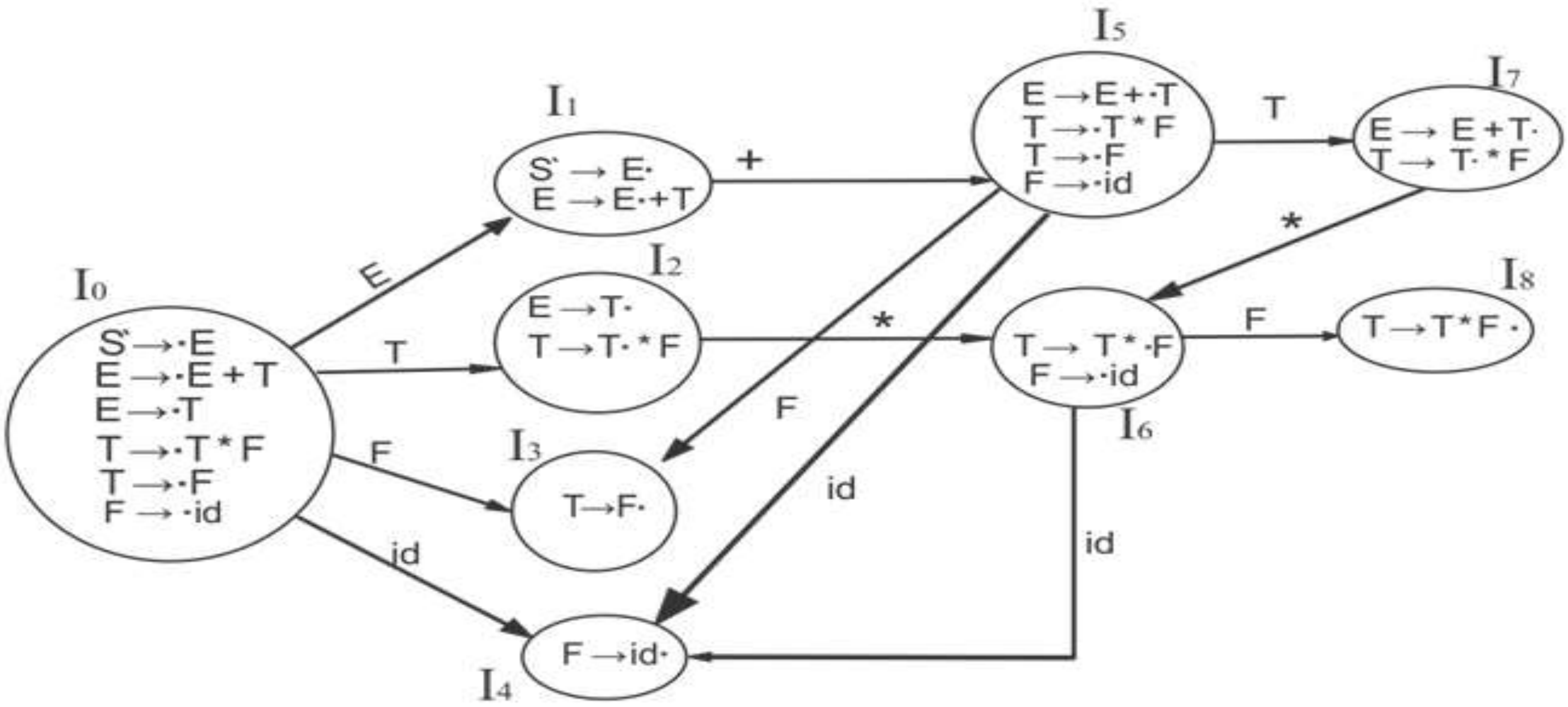
I4= Go to (I0, id) = closure (F → id•) = F → id•

I7= Go to (I5, T)
= Closure (E → E + T•) = E → E + T•
= Closure(T → T• * F) = I2
= Go to(I5,F) = Closure (T → F•) = I3
= Go to(I5,id) =Closure (F → id•) = I4

I8 = Go to (I6, F)
= Closure (T → T * F•) = T → T * F•
= Go to(I6,id) = Closure (F → id•) = I4

Go to (I7, *) = Closure (T → T * •F) = I6

Data Flow Diagram



SLR(1) Parse Table

States	Action				Go to		
	id	+	*	$	E	T	F
I_0	S_4				1	2	3
I_1		S_5		Accept			
I_2		R_2	S_6	R2			
I_3		R_4	R_4	R4			
I_4		R_5	R5	R5			
I_5	S4					7	3
I_6	S4						8
I_7		R1	S6	R1			
I_8		R3	R3	R3			

CLR(1) Parser

- CLR refers to canonical lookahead.
- CLR parsing use the canonical collection of LR (1) items to build the CLR (1) parsing table.
- CLR (1) parsing table produces the more number of states as compare to the SLR (1) parsing.
- In the CLR (1), we place the reduce node only in the lookahead symbols.

CLR(1) Parser

Various steps involved in the CLR (1) Parsing:

- For the given input string write a context free grammar
- Check the ambiguity of the grammar
- Add Augment production in the given grammar
- Create Canonical collection of LR (0) items
- Draw a data flow diagram (DFA)
- Construct a CLR (1) parsing table

CLR(1) Parser

LR (1) item

LR (1) item is a collection of LR (0) items and a look ahead symbol.

LR (1) item = LR (0) item + look ahead

- The look ahead is used to determine that where we place the final item.
- The look ahead always add $ symbol for the argument production.

CLR(1) Parse Table Construction

Example

CLR (1) Grammar

S → AA

A → aA

A → b

Add Augment Production, insert '•' symbol at the first position for every production in G and also add the lookahead.

S` → •S, $

S → •AA, $

A → •aA, a/b

A → •b, a/b

CLR(1) Parse Table Construction

I0 State:

Add Augment production to the I0 State and Compute the Closure

I0 = Closure (S\` → •S)

Add all productions starting with S in to I0 State because "." is followed by the non-terminal. So, the I0 State becomes

I0 = S\` → •S, $
S → •AA, $

Add all productions starting with A in modified I0 State because "." is followed by the non-terminal. So, the I0 State becomes.

I0= S\` → •S, $
S → •AA, $
A → •aA, a/b
A → •b, a/b

CLR(1) Parse Table Construction

VISHNU UNIVERSAL LEARNING

I0 = S` → •S, \$
S → •AA, \$
A → •aA, a/b
A → •b, a/b

I4= Go to (I0, b) = Closure (A → b•, a/b) = A → b•, a/b

I5= Go to (I2, A) = Closure (S → AA•, \$)
=S → AA•, \$

I6= Go to (I2, a) = Closure (A → a•A, \$)
I6 = A → a•A, \$
A → •aA, \$
A → •b, \$

I1= Go to (I0, S) = Closure (S` → S•, \$) = S` → S•, \$

I2= Go to (I0, A) = Closure (S → A•A, \$)
I2= S → A•A, \$
A → •aA, \$
A → •b, \$

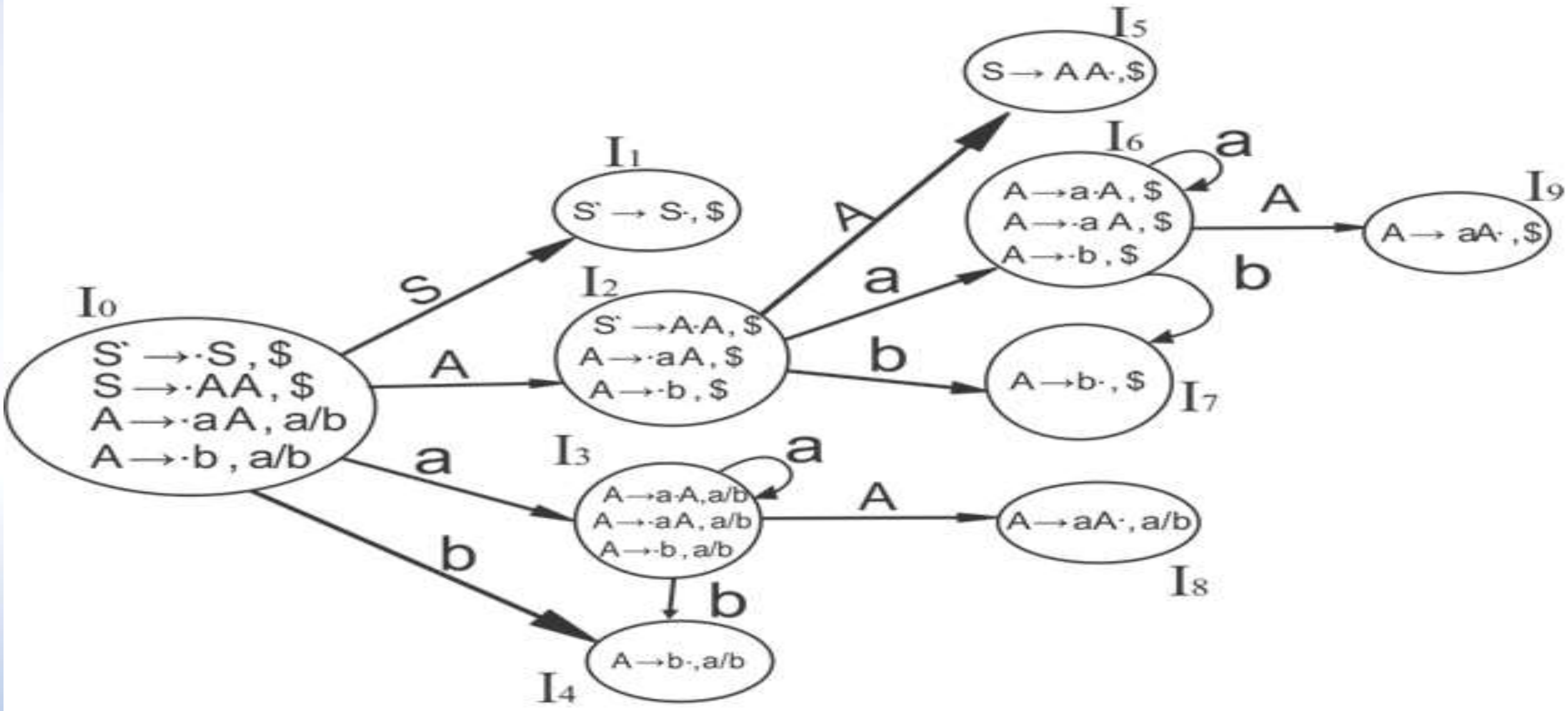
I7= Go to (I2, b) = Closure (A → b•, \$) = A → b•, \$

I8= Go to (I3, A) = Closure (A → aA•, a/b) = A → aA•, a/b
Go to (I3, a) = Closure (A → a•A, a/b) = I3
Go to (I3, b) = Closure (A → b•, a/b) = I4

I3= Go to (I0, a) = Closure (A → a•A, a/b)
I3= A → a•A, a/b
A → •aA, a/b
A → •b, a/b

I9= Go to (I6, A) = Closure (A → aA•, \$) = A → aA•, \$
Go to (I6, a) = Closure (A → a•A, \$) = I6
Go to (I6, b) = Closure (A → b•, \$) = I7

Data Flow Diagram



CLR(1) Parse Table

States	a	b	$	S	A
I_0	S_3	S_4		1	2
I_1			Accept		
I_2	S_6	S_7			5
I_3	S_3	S_4			8
I_4	R_3	R_3			
I_5			R_1		
I_6	S_6	S_7			9
I_7			R_3		
I_8	R_2	R_2			
I_9			R_2		

LALR(1) Parser

- LALR refers to the lookahead LR. To construct the LALR (1) parsing table, we use the canonical collection of LR (1) items.
- In the LALR (1) parsing, the LR (1) items which have same productions but different look ahead are combined to form a single set of items
- LALR (1) parsing is same as the CLR (1) parsing, only difference in the parsing table.

LALR(1) Parser Example

Example

S → AA

A → aA

A → b

Add Augment Production, insert '•' symbol at the first position for every production in G and also add the look ahead.

I0= S\` → •S, $

S → •AA, $

A → •aA, a/b

A → •b, a/b

LALR(1) Parse Table Construction

I0 = S` → •S, \$
S → •AA, \$
A → •aA, a/b
A → •b, a/b

I1= Go to (I0, S) = Closure (S` → S•, \$) = S` → S•, \$

I2= Go to (I0, A) = Closure (S → A•A, \$)
I2= S → A•A, \$
A → •aA, \$
A → •b, \$

I3= Go to (I0, a) = Closure (A → a•A, a/b)
I3= A → a•A, a/b
A → •aA, a/b
A → •b, a/b

I4= Go to (I0, b) = Closure (A → b•, a/b) = A → b•, a/b

I5= Go to (I2, A) = Closure (S → AA•, \$)
=S → AA•, \$

I6= Go to (I2, a) = Closure (A → a•A, \$)
I6 = A → a•A, \$
A → •aA, \$
A → •b, \$

I7= Go to (I2, b) = Closure (A → b•, \$) = A → b•, \$

I8= Go to (I3, A) = Closure (A → aA•, a/b) = A → aA•, a/b
Go to (I3, a) = Closure (A → a•A, a/b) = I3
Go to (I3, b) = Closure (A → b•, a/b) = I4

I9= Go to (I6, A) = Closure (A → aA•, \$) = A → aA•, \$
Go to (I6, a) = Closure (A → a•A, \$) = I6
Go to (I6, b) = Closure (A → b•, \$) = I7

LALR(1) Parser Example

If we analyze then LR (0) items of I3 and I6 are same but they differ only in their lookahead.

I3 = $A \rightarrow a\bullet A$, a/b
$A \rightarrow \bullet aA$, a/b
$A \rightarrow \bullet b$, a/b

I6= $A \rightarrow a\bullet A$, \$
$A \rightarrow \bullet aA$, \$
$A \rightarrow \bullet b$, \$

Clearly I3 and I6 are same in their LR (0) items but differ in their lookahead, so we can combine them and called as I36.

I36 = $A \rightarrow a\bullet A$, a/b/\$
$A \rightarrow \bullet aA$, a/b/\$
$A \rightarrow \bullet b$, a/b/\$

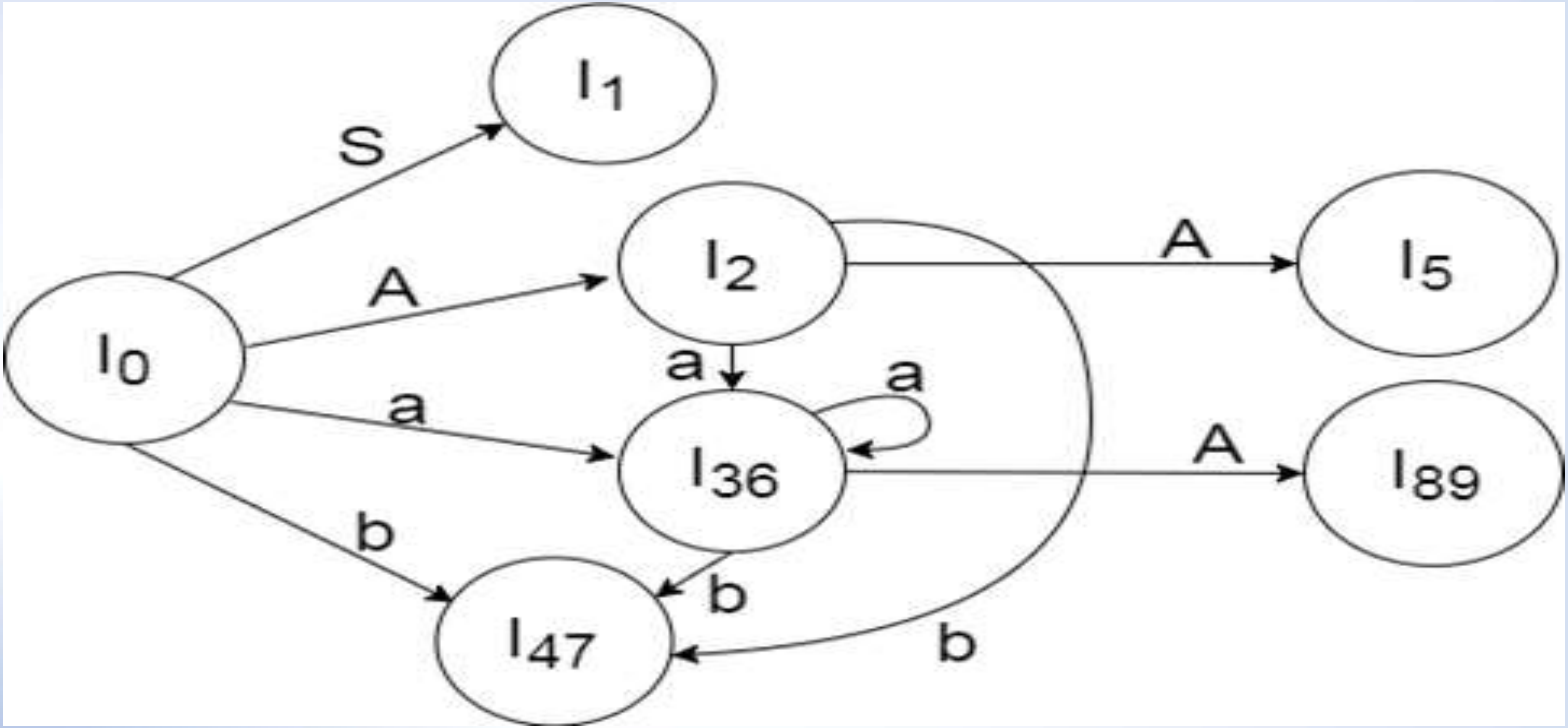
The states I4 and I7 are same but they differ only in their look ahead, so we can combine them and called as I47.

I47 = $A \rightarrow b\bullet$, a/b/\$

The states I8 and I9 are same but they differ only in their look ahead, so we can combine them and called as I89.

I89 = $A \rightarrow aA\bullet$, a/b/\$

LALR(1) Data Flow Diagram



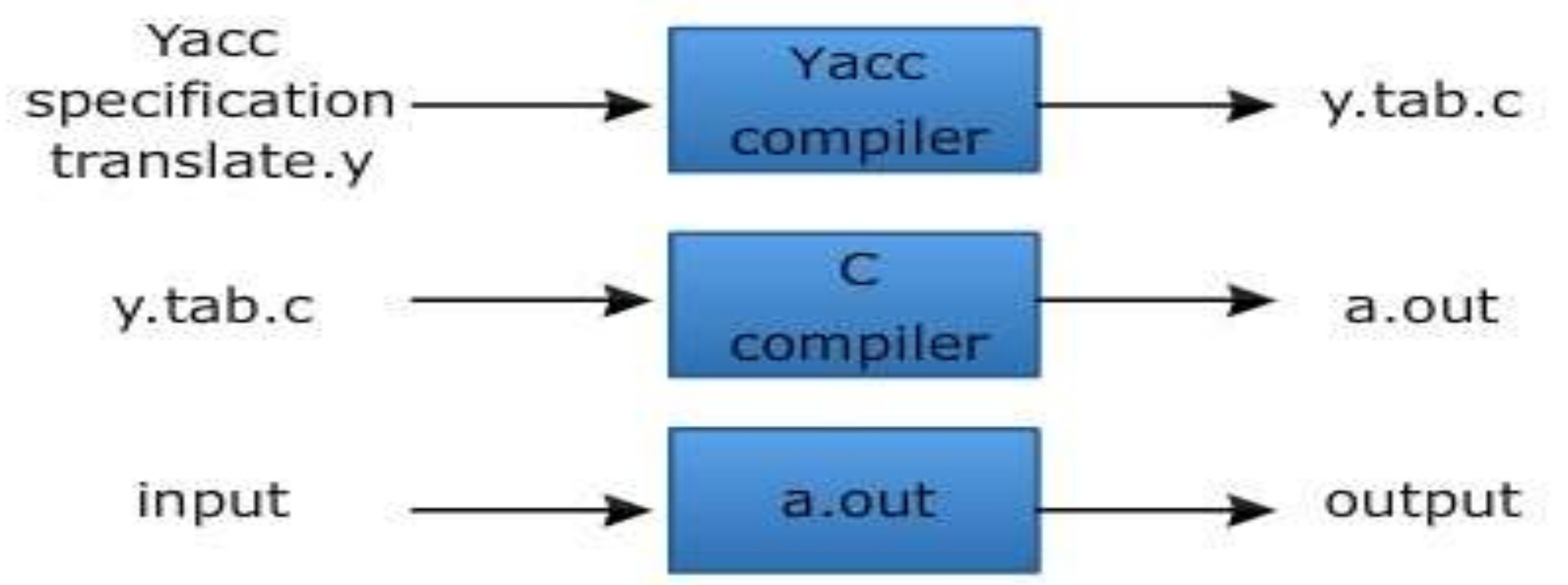
LALR(1) Parse Table

States	Action			Go to	
	a	b	$	S	A
I0	S36	S47		1	2
I1			Accept		
I2	S36	S47			5
I36	S36	S47			89
I47	R3	R3	R3		
I5			R1		
I89	R2	R2	R2		

Automatic Parser Generator

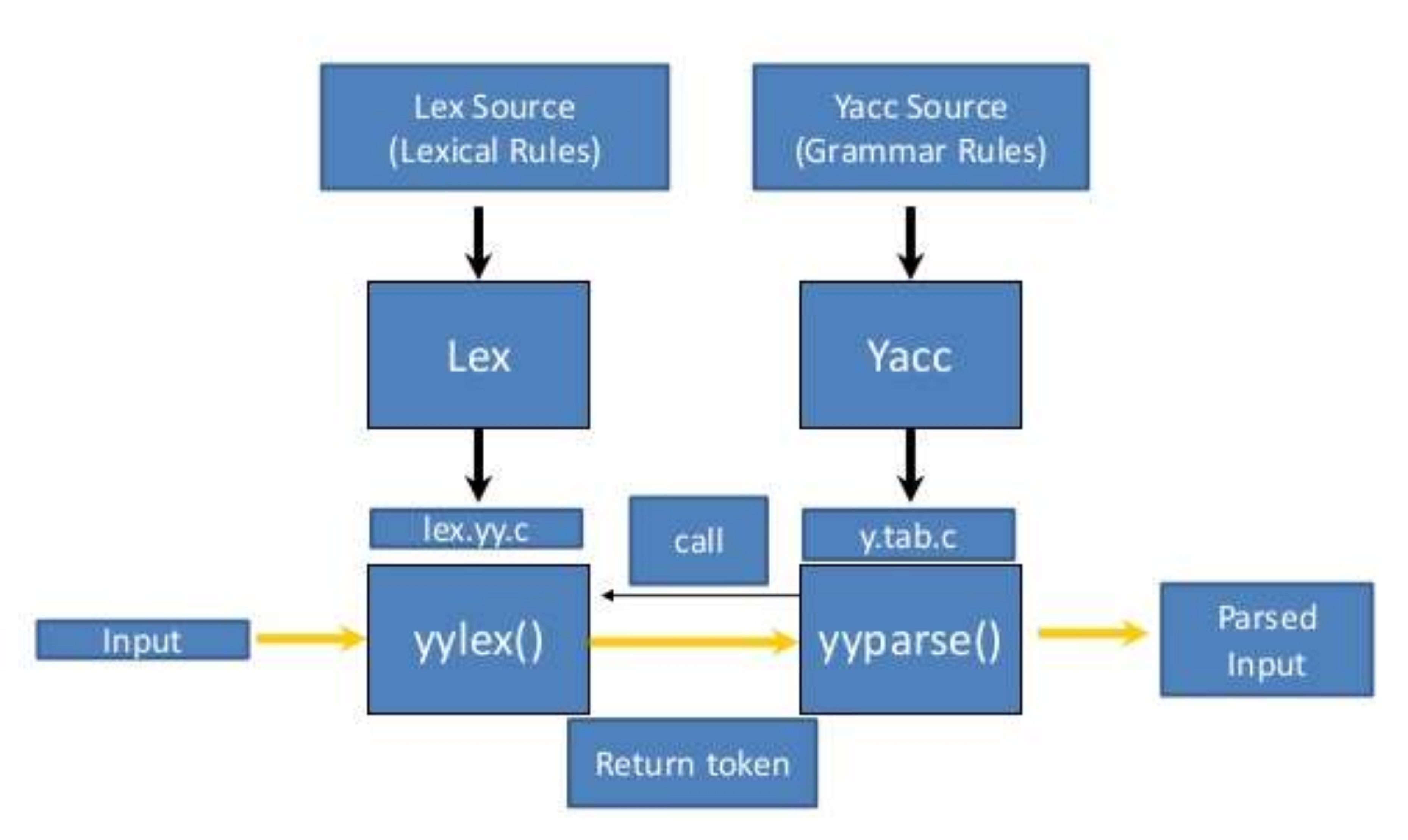
- YACC is an automatic tool that generates the parser program.
- YACC stands for **Yet Another Compiler Compiler**.
- YACC provides a tool to produce a parser for a given grammar.
- YACC is a program designed to compile a LALR (1) grammar.
- It is used to produce the source code of the syntactic analyzer of the language produced by LALR (1) grammar.
- The input of YACC is the rule or grammar and the output is a C program.

YACC



Creating an Input/Output Translator with YACC

Lex and YACC



Lex Source
(Lexical Rules)
Yacc Source
(Grammar Rules)
Lex
Yacc
lex.yy.c
call
y.tab.c
Input
yylex()
yyparse()
Parsed
Input
Return token

Structure of YACC

```
%{
    C declarations
%}
    yacc declarations
%%
    Grammar rules
%%
    Additional C code
```

– Comments enclosed in /* ... */ may appear in any of the sections.

Operator Precedence Parsing

Operator Precedence Grammar

A grammar that satisfies the following 2 conditions is called as Operator Precedence Grammar–

- There exists no production rule which contains ε on its RHS.
- There exists no production rule which contains two non-terminals adjacent to each other on its RHS.

Operator Precedence Parsing

Example

E → EAE | (E) | -E | id

A → + | - | x | / | ^

Operator Precedence Grammar

E → E + E | E - E | E x E | E / E | E ^ E | (E) | -E | id

Operator Precedence Grammar

Operator Precedence Parsing

Operator Precedence Parser:

A parser that reads and understand an operator precedence grammar is called as **Operator Precedence Parser**.

Designing Operator Precedence Parser-

In operator precedence parsing,

- Firstly, we define precedence relations between every pair of terminal symbols.
- Secondly, we construct an operator precedence table.

Defining Precedence Relations

Rule-01:

- If precedence of b is higher than precedence of a, then we define **a < b**
- If precedence of b is same as precedence of a, then we define **a = b**
- If precedence of b is lower than precedence of a, then we define **a > b**

Rule-02:

- An **identifier** is always given the higher precedence than any other symbol.
- **$** symbol is always given the lowest precedence.

Rule-03:

- If two operators have the same precedence, then we go by checking their associativity.

Operator Precedence Parsing Example

Consider the following grammar-

E → EAE | id

A → + | *

Construct the operator precedence parser and parse the string **id + id * id**.

<u>Solution:</u>

We convert the given grammar into operator precedence grammar.

The equivalent operator precedence grammar is-

E → E + E

E → E * E

E → id

The terminal symbols in the grammar are { id, + , * , $ }

Operator Precedence Parsing Example

We construct the operator precedence table as-

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

Operator Precedence Parsing Example

Stack	Input	Action	Comments
$	id + id * id$	$ <· id Shift	As $ is lesser than id, we shift
$id	+id*id$	id ·> + reduce E → id	id is greater than +, so we pop as E à id ,will lead to id as a handle
$	+id*id$	Shift	
$+	id*id$	Shift	
$ + id	*id$	id .> * reduce E → id	
$ +	* id $	Shift	
$ + *	id $	Shift	
$ + * id	$	id ·> $ reduce E → id	
$+ *	**$**	*** ·> $ reduce E → E*E**	*, + has greater than relation with $. So we keep popping the symbols.
$ +	$	+ ·> $ reduce E → E+E	The input is already consumed thus announcing a successful parsing.
$	$	Accept	

Thank You