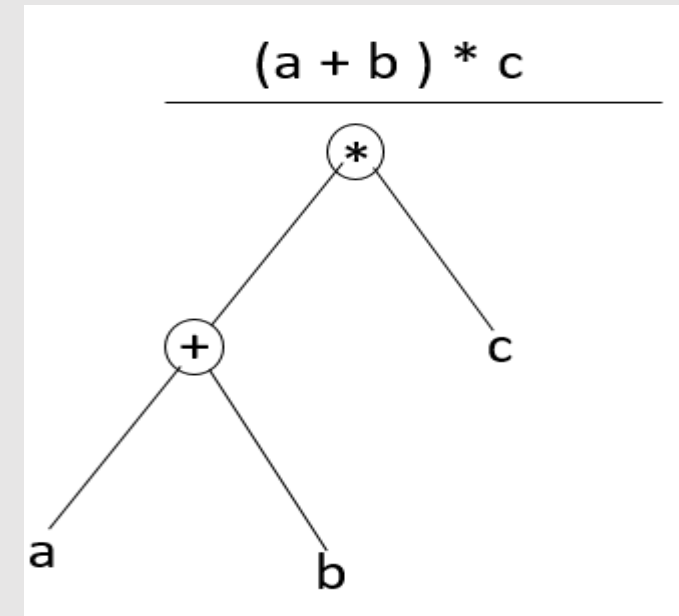


Parsing or Syntax Analysis

Parsing is a process which takes input string and produces either a parse tree or generates syntax error.

- Second phase of compilation
- Syntax Analyser takes tokens from Lexical Analyser & group them in a program structure, if any syntax cannot be recognized then it will give syntactical error.

Example :- for the given program statement $(a + b) * c$ parser produces the following parse tree



Role of Parser

We need to construct parse tree based on derivation process.

Derivation process is a process of deriving a input string by applying production rules

1. Parser scan input from left to right.
2. Parser makes use of production rules for deriving the given input string .
3. And finally a parse tree is constructed.

Ex:- $x = a + b * c;$



Role of Parser

1. It verifies the structure generated by the tokens based on the grammar.
2. It constructs the parse tree.
3. It reports the errors.
4. It performs error recovery.

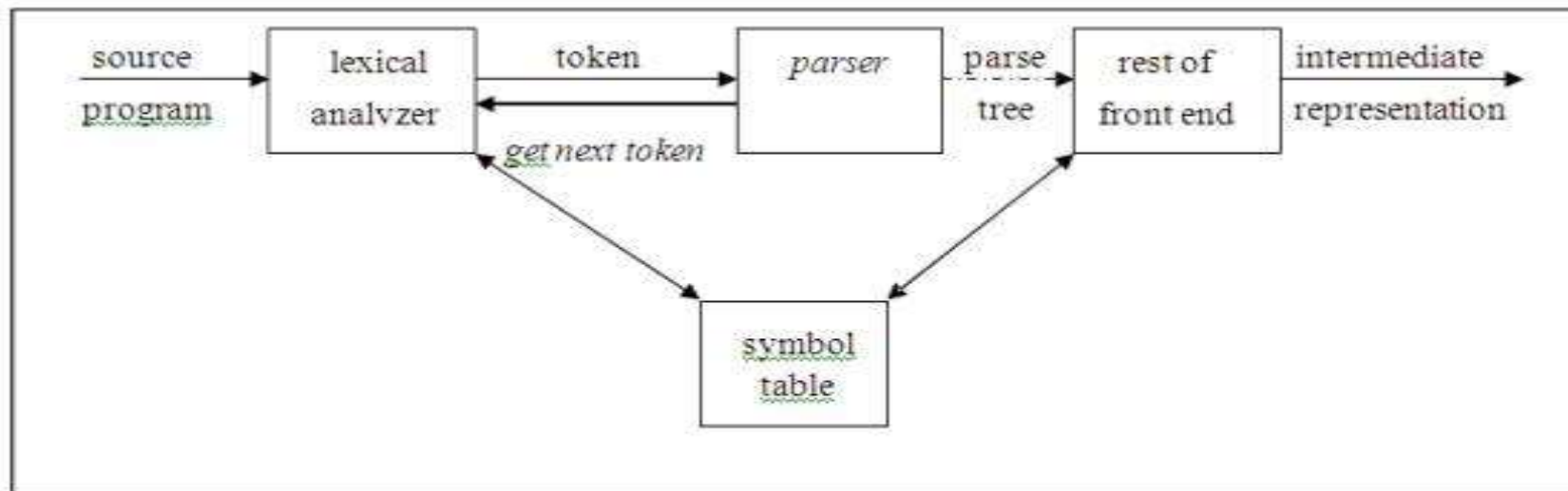
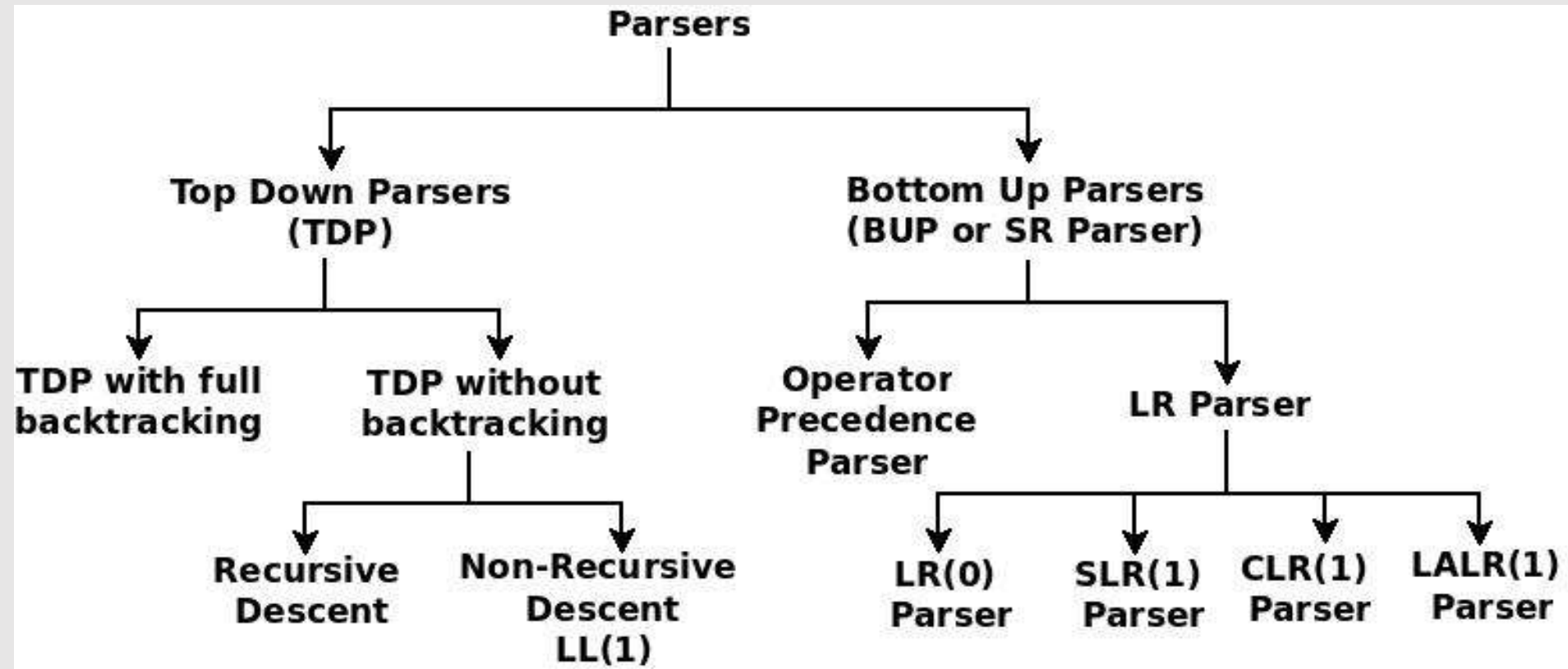


Fig. 2.1 Position of parser in compiler model

Parsing Techniques

Depending upon how the parse tree is built, parsing techniques are classified

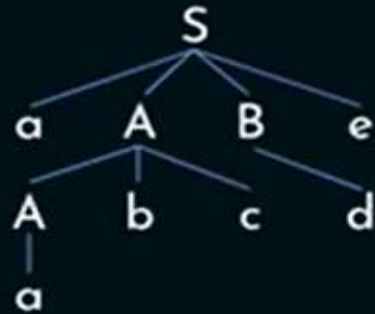


Top Down vs Bottom Up Parsing

Generation of Parse Tree: $S \rightarrow aABe$, $A \rightarrow Abc \mid a$, $B \rightarrow d$

aabcde

Top down approach:

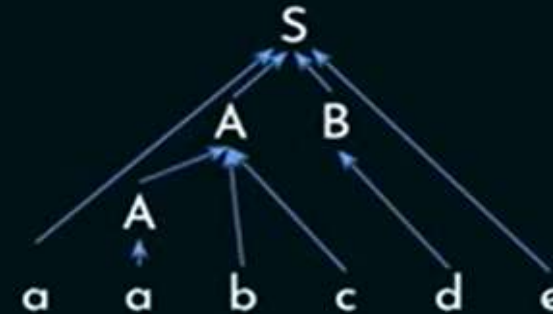


$S \Rightarrow aABe$
 $\Rightarrow aAbcBe$
 $\Rightarrow aabcBe$
 $\Rightarrow aabcde$

Decision:
Which production
to use.

(Left most Derivation)

Bottom up approach:



$S \Rightarrow aABe$
 $\Rightarrow aAde$
 $\Rightarrow aAbcde$
 $\Rightarrow aabcde$

Decision:
When to reduce.

(Right most Derivation) - In reverse.

Problems with Top Down Parsing

- The parse tree is generated from top to bottom (from root to leaves).
- The leftmost derivation is applied at each derivation step.
- In top-down parsing selection of proper production rule is very important.

Problems with Top Down Parsing

S

1. Backtracking
2. Left Recursion
3. Left Factoring
4. Ambiguity

Top Down Parsing- Backtracking

Backtracking:- It is a technique in which non-terminal symbols are expanded on trial & error basis until a match for input string is found.

Disadvantages:

1. This powerful technique
2. This technique is slower and requires exponential time in general.
3. Hence Backtracking is not preferred for practical compilers.

Top Down Parsing- Backtracking

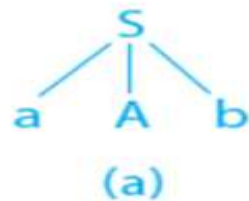
example:-Consider the input string 'acb' for the following grammar

$S \rightarrow aAb$
 $A \rightarrow cd \mid c$

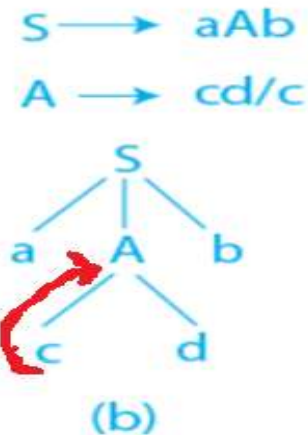
While constructing parse tree ,

(i) if input string is matched with the left most leaf of parse tree then the input pointer advances.

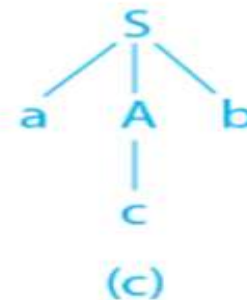
(ii) if input string is matched with the left most leaf of parse tree then another alternative of NT is applied.



To match the input string 'acb', we apply the production rule of 'A' i.e, $A \rightarrow cd$



As the input string 'acb' does not match with the leaf nodes 'acdb' Here we backtrack and apply other production rule of 'A' i.e, $A \rightarrow c$



Now the input string 'acb' matches the leaf nodes 'acb'

Therefore for the given input string 'acb' parse tree is produced successfully.

Top Down Parsing-Left Recursion

Left recursion is considered to be a problematic situation for Top down parsers where as Right recursion and general recursion does not create any problem for the Top down parsers.

Therefore, left recursion has to be eliminated from the grammar.

$$A \rightarrow A\alpha / \beta$$

(Left Recursive Grammar)

Convert to



$$A \rightarrow \beta A' \\ A' \rightarrow \alpha A' / \epsilon$$

(Right Recursive Grammar)

$S \rightarrow aB \mid AA$
 $A \rightarrow a \mid aBA \mid AAA$
 $B \rightarrow b$
 $X \rightarrow a$

To remove left recursion ($A \rightarrow AAA$) from $A \rightarrow AAA \mid a \mid aBA$,

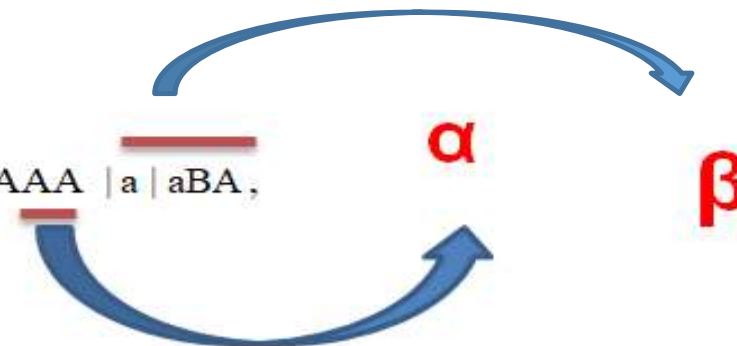
We get : $S \rightarrow aB \mid AA$

$A \rightarrow aC \mid aBAC$

$C \rightarrow AAC \mid \epsilon$

$B \rightarrow b$

$X \rightarrow a$



Top Down Parsing- Left Factoring

Left Factoring is a process to transform the grammar with common prefixes.

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

if $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$
are two A -productions,
left-factored productions become

$$\begin{aligned} A &\rightarrow \alpha A' \\ A' &\rightarrow \beta_1 \mid \beta_2 \end{aligned}$$

Problem:

Do left factoring in the following grammar-

$$S \rightarrow iEtS \mid iEtSeS \mid a$$

$$E \rightarrow b$$

Solution-

The left factored grammar is-

$$S \rightarrow iEtSS' \mid a$$

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

$$E \rightarrow b$$

Top Down Parsing- Ambiguity

A grammar is said to be ambiguous if for any string generated by it, it produces more than one Parse tree

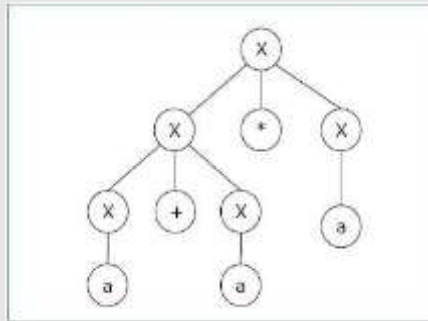
Or derivation tree

Or syntax tree

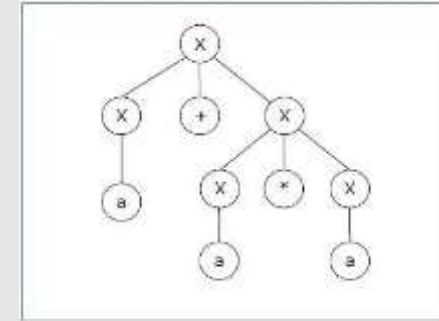
Or leftmost derivation

Or rightmost derivation

Example - $X \rightarrow X+X \mid X*X \mid X \mid a$



$$\begin{aligned} X &\rightarrow X+X \\ &\rightarrow a+X \\ &\rightarrow a+X*X \\ &\rightarrow a+a*X \\ &\rightarrow a+a*a \end{aligned}$$



$$\begin{aligned} X &\rightarrow X*X \\ &\rightarrow X+X*X \\ &\rightarrow a+X*X \\ &\rightarrow a+a*X \\ &\rightarrow a+a*a \end{aligned}$$

Here we are able to create 2 parse trees for given production, so the given grammar is ambiguous.

Recursive Descent

- It is a kind of Top-Down Parser.
- It uses collection of recursive procedures for parsing given input string (for each non terminal)
- CFG is used to build recursive routines.
- Each nonterminal in the grammar is implemented as a function.
- Begin with the start symbol S of the grammar by calling the function S().
- Based on the first token received, apply the appropriate grammar rule for S.
- Continue in this manner until S is “satisfied.”

Advantages:-

- Simple to build

Disadvantages:-

- Error-recovery is difficult.
- They are not able to handle as large a set of grammars as other parsing methods.

Recursive Descent- procedure

The procedure will start from top root node and we have to apply several production rules to get bottom leaf nodes.

Step1: If input is a Non-terminal then call the corresponding procedure of that Non-terminal

Setp2: If input is a terminal then compare terminal with input string
if it is a match the pointer advances

Step3: If a Non-terminal produces more than one productions then all the production code should be written in the corresponding function.

Step4: There is no need to define main() . If we define main() then we have to call start symbol function in main().

Recursive Descent- procedure

Example:-

$E \rightarrow i E'$

$E' \rightarrow + i E' \mid \epsilon$

There are 2 Non-terminals (E & E') so we define 2 functions

```
E ()
{
    if (input == 'i' )
        input++;
    Eprime();
}

Eprime()
{
    if (input == '+' )
    {
        input++;
        if (input == 'i' )
            input++;
        Eprime();
    }
    else
        return;
}
```

sample input string = i+i \$

First & Follow functions

Compiler design is the process of creating software that translates source code written in a high-level programming language into machine code that can be executed by a computer. Parsing is one of the crucial steps in the process of compiler design that involves breaking down the source code into smaller pieces and analyzing their syntax to generate a parse tree.

FIRST and FOLLOW in Compiler Design functions are used to generate a predictive parser, which is a type of syntax analyzer used to check whether the input source code follows the syntax of the programming language.

The FIRST and FOLLOW in Compiler Design sets are used to construct a predictive parser, which can predict the next token in the input. First tells which terminal can start production whereas the follows tells the parser what terminal can follow a non-terminal.

By using FIRST and FOLLOW the compiler come to know in advance, “ which character of the string is produced by applying which production rule”.

First & Follow functions

To compute the first set of a nonterminal, we must consider all possible productions of the nonterminal symbol and compute the first set of the symbols that can appear as the first token in the right-hand side of each production. If any of these symbols can derive the empty string (i.e., the ϵ symbol), then we must also include the First set of the next symbol in the production.

RULES:-

1. If $A \rightarrow A \rightarrow a\alpha$, $\alpha \in (V \cup T)^*$ then **FIRST(A) = { a }**
2. If $A \rightarrow \epsilon$ then **FIRST(A) = { ϵ }**
3. If $A \rightarrow BC$ then
 - a) **FIRST(A) = FIRST(B)**, if FIRST(B) doesnot contain ϵ
 - b) **FIRST(A) = FIRST(B) U FIRST(C)**, if FIRST(B) contains ϵ

Example:-

$E \rightarrow TE'$

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

$T \rightarrow FT'$

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

$F \rightarrow (E) / id$

$FIRST(E) = \{ (, id \}$

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

$FIRST(T) = \{ (, id \}$

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

$FIRST(F) = \{ (, id \}$

First & Follow functions

The Follow function of nonterminal symbols represents the set of terminal symbols that can appear immediately after the nonterminal in any valid derivation.

To compute the Follow set of a nonterminal, we must consider all the productions in which the nonterminal symbol appears and add the Follow set of the nonterminal's parent to the Follow set of the nonterminal. We must also consider the case where the nonterminal appears at the end of production and add the Follow set of the nonterminal's parent to the Follow set of the nonterminal.

S

RULES:-

1. If 'S' is a start symbol then **FOLLOW(S) = { \$ }**
2. If $A \rightarrow \alpha B \beta$ then
 - a) **FOLLOW(B) = FIRST(β)** , if $\text{FIRST}(\beta) \neq \epsilon$
 - b) **FOLLOW(B) = FOLLOW(A)** , if $\beta \rightarrow \epsilon$
1. If $A \rightarrow \alpha B$ then **FOLLOW(B) = FOLLOW(A)**

First & Follow functions

Example:-

$E \rightarrow TE'$

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

$T \rightarrow FT'$

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

$F \rightarrow (E) / id$

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

- $\$$ included as E is start symbol
- $)$ included as it's the following symbol after E in $F \rightarrow (E)$

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

- By applying rule3 to $E \rightarrow TE'$, $FOLLOW(E') = FOLLOW(E)$

3) $FOLLOW(T) = \{ +, \$,) \}$

- By applying rule2 (a) to $E \rightarrow TE'$, $FOLLOW(E') = FIRST(E')$
- $FIRST(E') = \{ +, \epsilon \}$ -----1, But Follow should not contain ϵ
- So substitute ϵ for E' in $E \rightarrow TE'$ resulting $E \rightarrow T$
- For $E \rightarrow T$, $FOLLOW(T) = FOLLOW(E) = \{ \$,) \}$ -----2
- Combining 1 & 2 we get $FOLLOW(T) = \{ +, \$,) \}$

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

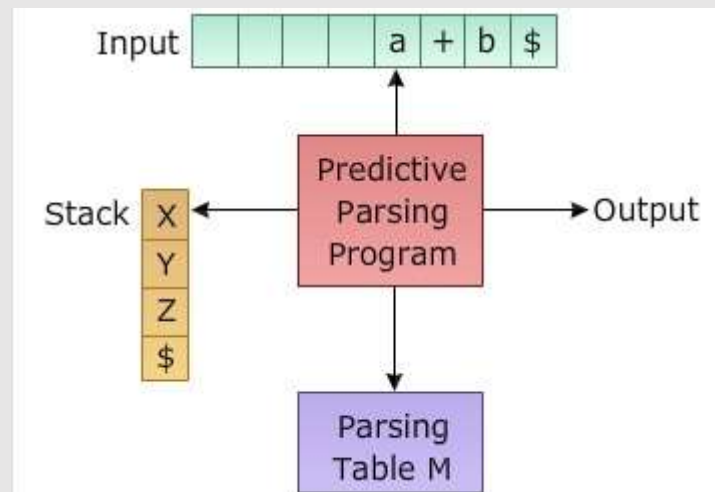
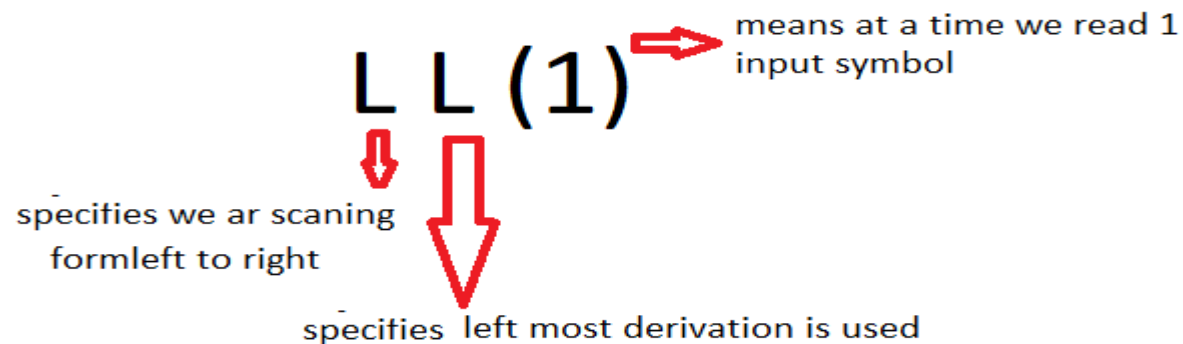
- By applying rule3 to $T \rightarrow FT'$, $FOLLOW(T') = FOLLOW(T)$

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

- Apply same procedure as for 3 rd point

LLR(1) Parser

- LL(1) parser / Predictive Parser / Non-Recursive Descent Parser
- parser program works with the following 3 components to produce output
 - 1) **INPUT**: Contains string to be parsed with \$ as it's end marker
 - 2) **STACK**: Contains sequence of grammar symbols with \$ as it's bottom marker. Initially stack contains only \$
 - 3) **PARSING TABLE**: A two dimensional array $M[A,a]$, where A is a non-terminal and a is a Terminal
It is a tabular implementation of the recursive descent parsing, where a stack is maintained by the parser table than the language in which parser is written



LLR(1) Parser

Construction of LL(1) parser

STEP1:- Elimination of Left Recursion

STEP2:- Elimination of Left Factoring

STEP3:- Calculation of First and Follow s

STEP4:- Construction Parsing table

STEP5:- String Validation, check whether Input string is accepted by parser or not

LLR(1) Parser

Example:-

$E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow (E) \mid id$

STEP1:-Elimination of Left Recursion

$E \rightarrow E+T \mid T$

Converted to

$E \rightarrow TE'$

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

$T \rightarrow T*F \mid F$

Converted to

$T \rightarrow FT'$

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

After removing Left Recursion we get

$E \rightarrow TE'$

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

$T \rightarrow FT'$

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

$F \rightarrow (E) \mid id$



LLR(1) Parser

STEP2:- Elimination of Left Factoring

$E \rightarrow TE'$

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

$T \rightarrow FT'$

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

$F \rightarrow (E) \mid id$

The above grammar has no left factoring.

if $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$
are two A -productions,
left-factored productions become

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

LLR(1) Parser

STEP3:- Calculation of First and Follow

$E \rightarrow TE'$

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

$T \rightarrow FT'$

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

$F \rightarrow (E) \mid id$

$FIRST(E) = \{ (, id \}$

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

$FIRST(T) = \{ (, id \}$

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

$FIRST(F) = \{ (, id \}$

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

- $\$$ included as E is start symbol
- $)$ included as it's the following symbol after E in $F \rightarrow (E)$

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

- By applying rule3 to $E \rightarrow TE'$, $FOLLOW(E') = FOLLOW(E)$

3) $FOLLOW(T) = \{ +, \$,) \}$

- By applying rule2 (a) to $E \rightarrow TE'$, $FOLLOW(E') = FIRST(E')$
- $FIRST(E') = \{ +, \epsilon \}$ -----1, But Follow should not contain ϵ
- So substitute ϵ for E' in $E \rightarrow TE'$ resulting $E \rightarrow T$
- For $E \rightarrow T$, $FOLLOW(T) = FOLLOW(E) = \{ \$,) \}$ -----2
- Combining 1 & 2 we get $FOLLOW(T) = \{ +, \$,) \}$

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

- By applying rule3 to $T \rightarrow FT'$, $FOLLOW(T') = FOLLOW(T)$

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

- Apply same procedure as for 3rd point

LLR(1) Parser

STEP4:- Construction Parsing table

- Rows are Non Terminals
- Columns are Terminals
- All the **Null Productions** of the Grammars will go under the Follow elements and the remaining productions will lie under the elements of the First set.

Predictive parsing table :

NON- TERMINAL	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	$F \rightarrow id$			$F \rightarrow (E)$		

LLR(1) Parser

STEP5:- String Validation, check whether Input string is accepted by parser or not
Actions are decided by mapping Top of stack & input string in parsing Table

Stack implementation:

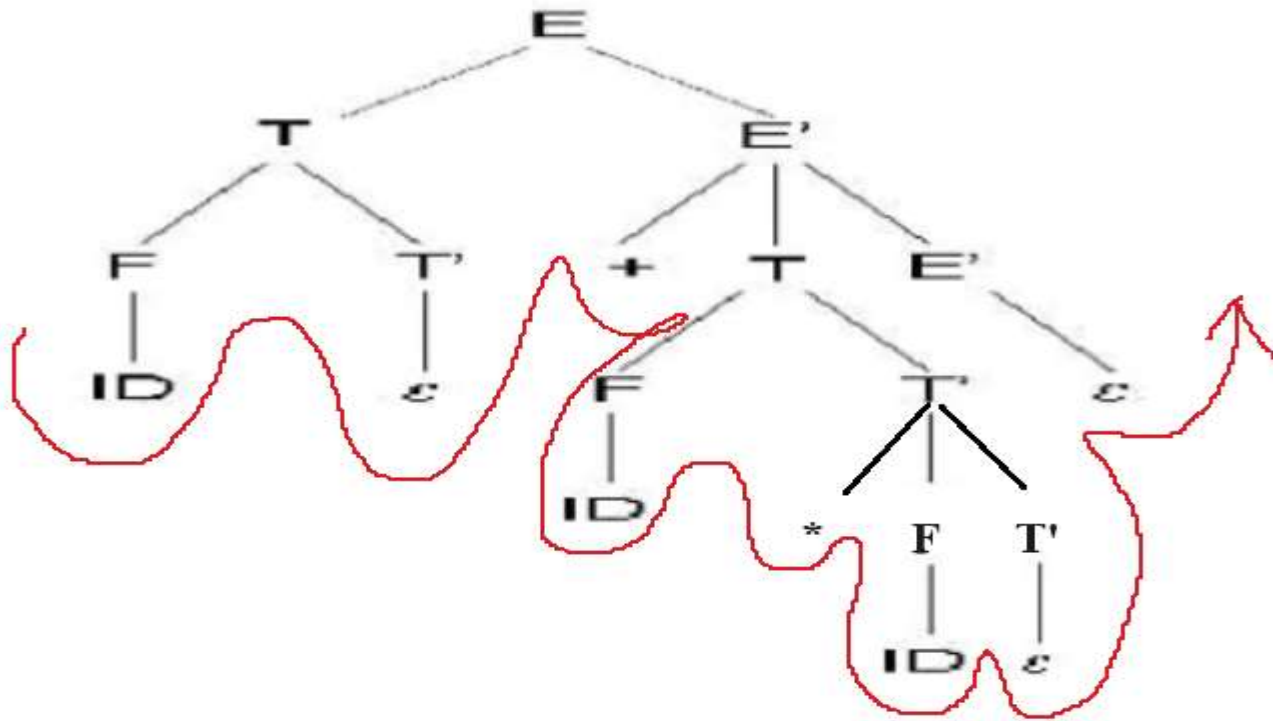
stack	Input	Output
\$E	id+id*id \$	
\$E'T	id+id*id \$	$E \rightarrow TE'$
\$E'T'F	id+id*id \$	$T \rightarrow FT'$
\$E'T'id	id+id*id \$	$F \rightarrow id$
\$E'T'	+id*id \$	pop id
\$E'	+id*id \$	$T' \rightarrow \epsilon$
\$E'T+	+id*id \$	$E' \rightarrow +TE'$
\$E'T	id*id \$	pop +

stack	Input	Output
\$E'T'F	id*id \$	$T \rightarrow FT'$
\$E'T'id	id*id \$	$F \rightarrow id$
\$E'T'	*id \$	pop id
\$E'T'F*	*id \$	$T' \rightarrow *FT'$
\$E'T'F	id \$	pop *
\$E'T'id	id \$	$F \rightarrow id$
\$E'T'	\$	pop id
\$E'	\$	$T' \rightarrow \epsilon$
\$	\$	$E' \rightarrow \epsilon$

After parsing entire string if the stack contains only \$ then we can say that the string is accepted by parser.

LLR(1) Parser

Now draw the parse tree from the Action column of the parsing table.



BOTTOM UP PARSING

Reduction, it is the activity in which the parser tries to match substring of input string with RHS of production rule and replace it by corresponding LHS.

Handle, is the substring of input string that matches with RHS of production

It is the process of reducing the input string to the start symbol of the grammar i.e, Right most derivation in reverse order.

1) The input string is first taken



2) Then we try to reduce substring of input string with the help of grammar.



3) Finally we try to obtain the start symbol.

Parse tree is constructed starting from leaf nodes.

Leaf nodes are reduced further to internal nodes.

These internal nodes are further reduced & eventually a root node is obtained.

BOTTOM UP PARSING

Example-

$S \rightarrow aABe$

$A \rightarrow Abc \mid b$

$B \rightarrow d$

Let input string is "abbcede"

STEP1:

abbcede"

STEP2:

"aAbcde"

Here substring of input string 'b' matches with the production $A \rightarrow b$, so replace substring 'b' with corresponding LHS A .

STEP3:

"aAde"

Here substring of input string 'Abc' matches with the production $A \rightarrow Abc$, so replace substring 'Abc' with corresponding LHS A .

STEP4:

"aABe"

Here substring of input string 'd' matches with the production $B \rightarrow d$, so replace substring 'd' with corresponding LHS B .

STEP5:

S

Here substring of input string 'aABe' matches with the production $S \rightarrow aABe$, so replace substring 'aABe' with corresponding LHS S .

BOTTOM UP PARSING

Handle pruning, it is the process of obtaining the Right most derivation in reverse order.

Example-

$S \rightarrow aABe$

$A \rightarrow Abc \mid b$

$B \rightarrow d$

$S \rightarrow aA\textcolor{red}{B}e$


$S \rightarrow aA\textcolor{red}{d}e$

$S \rightarrow aA\textcolor{red}{b}cde$

$S \rightarrow abbcde$

Let input string is "abbcde"

Right sentential form	Handle	Production
abbcde	b	$A \rightarrow b$
aAbcde	Abc	$A \rightarrow Abc$
aAde	d	$B \rightarrow d$
aABe	aABe	$S \rightarrow aABe$
S		



Here Bottom up parsing is producing Right most derivation in reverse order.

Shift Reduce Parser

Shift Reduce parser a bottom-up parsing technique i.e. the parse tree is constructed from leaves(bottom) to the root(up).

This parser requires some data structures i.e.

- An **input buffer** for storing the input string& string is terminated with \$
- A **stack** for storing Symbols of production rules, Initially bottom of stack stores \$



Then string is said to be accepted

1. **Shift:** This involves push operation pushing symbols from the input buffer onto the stack.
2. **Reduce:** When a match for the handle(RHS of production) is found at the top of the stack, the reduce operation applies the applicable production rules, i.e., pops out the RHS of the production rule from the stack and pushes the LHS of the production rule onto the stack.
3. **Accept:** If only the start symbol is present in the stack and the input buffer is empty then, the parsing action is called accept. When accepted action is obtained, it means successful parsing is done.
4. **Error:** This is the situation in which the parser can neither perform shift action nor reduce action and not even accept action.

Shift Reduce Parser-example1

Example 1 – Consider the grammar $S \rightarrow S + S \mid S * S \mid id$
Perform Shift Reduce parsing for input string “id + id + id”.

Stack	Input Buffer	Parsing Action
\$	id+id+id\$	Shift
\$id	+id+id\$	Reduce $S \rightarrow id$
\$S	+id+id\$	Shift
\$S+	id+id\$	Shift
\$S+id	+id\$	Reduce $S \rightarrow id$
\$S+S	+id\$	Reduce $S \rightarrow S+S$
\$S	+id\$	Shift
\$S+	id\$	Shift
\$S+id	\$	Reduce $S \rightarrow id$
\$S+S	\$	Reduce $S \rightarrow S+S$
\$S	\$	Accept

Shift Reduce Parser-example2

Example 3 – Consider the grammar $S \rightarrow (L) \mid a$ & $L \rightarrow L, S \mid S$
 Perform Shift Reduce parsing for input string “(a, (a, a))”.

Stack	Input Buffer	Parsing Action
\$	(a , (a , a)) \$	Shift
\$ (a , (a , a)) \$	Shift
\$ (a	, (a , a)) \$	Reduce $S \rightarrow a$
\$ (S	, (a , a)) \$	Reduce $L \rightarrow S$
\$ (L	, (a , a)) \$	Shift
\$ (L ,	(a , a)) \$	Shift
\$ (L , (a , a)) \$	Shift
\$ (L , (a	, a)) \$	Reduce $S \rightarrow a$
\$ (L , (S	, a)) \$	Reduce $L \rightarrow S$
\$ (L , (L	, a)) \$	Shift

Shift Reduce Parser-example2

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

LR Parser or LR(k) parser

LR parsing is one type of bottom up parsing. It is used to parse the large class of grammars.

In the LR parsing, "L" stands for left-to-right scanning of the input.

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

"K" is the number of input symbols of the look ahead used to make number of parsing decision.

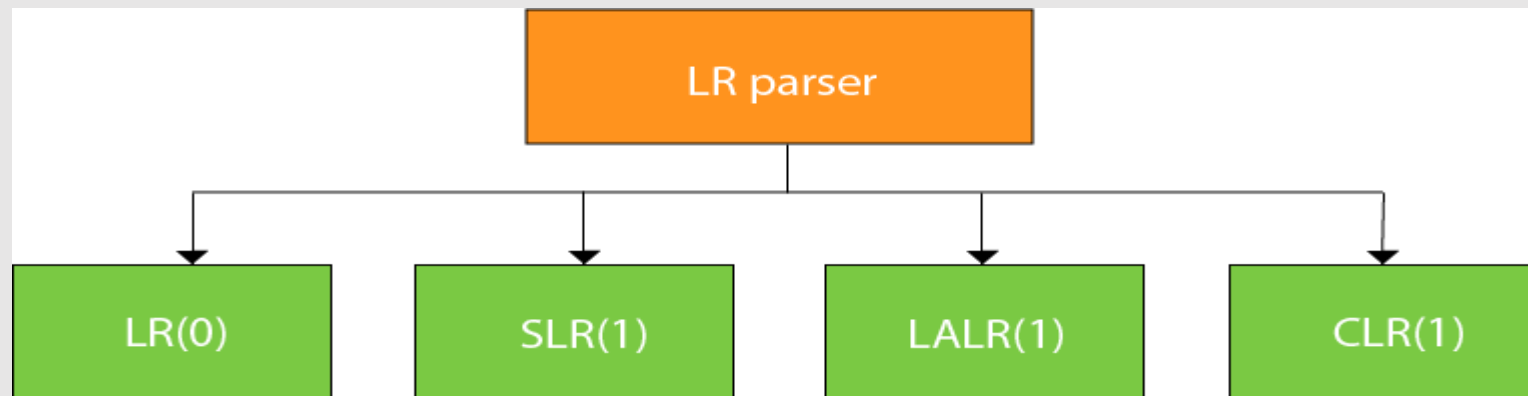


Fig: Types of LR parser

NOTE:-

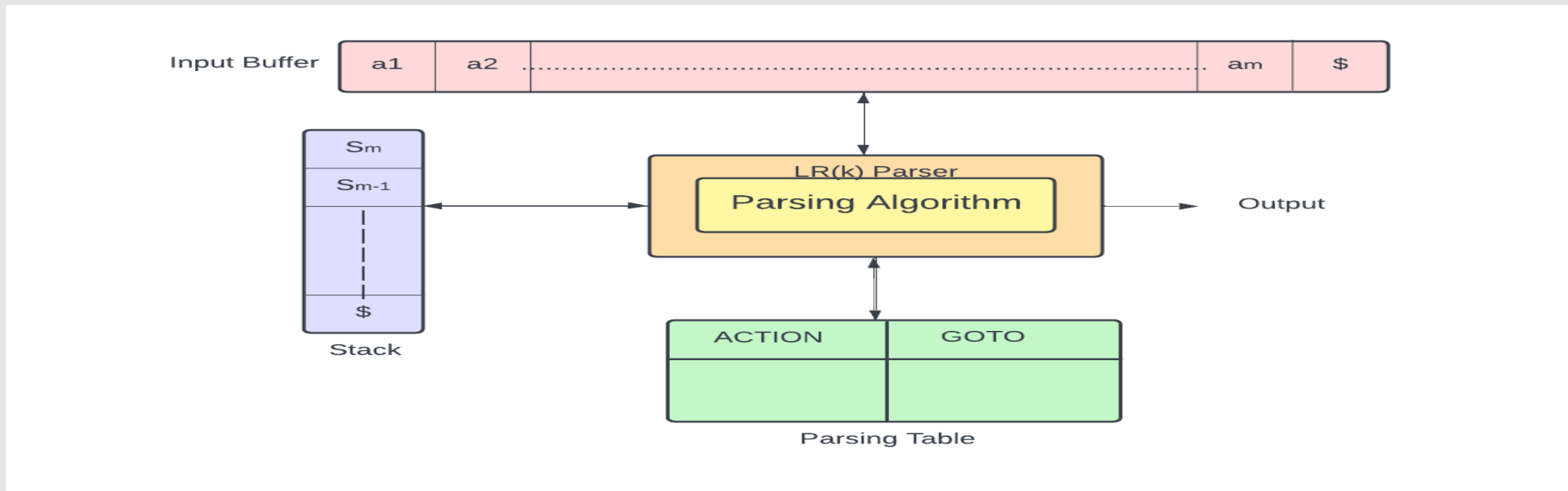
LR(0) and SLR(1) uses canonical collection of LR(0) items

LALR(1) and CLR(1) uses canonical collection of LR(1) items

LR Parser or LR(k) parser

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

NOTE:-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.



LR Parser or LR(k) parser-- terminology

The **dot(.)** is useful to indicate that how much of the input has been scanned up to a given point in the process of parsing.

Item , is any production rule with a dot(.) at the beginning of RHS of production.

LR (0) item , is any production rule with a dot(.) at some position of RHS of production.

Final Item , is any production rule with a dot(.) at the end of RHS of production.

Ex:-

$S \rightarrow .A$ // item

$S \rightarrow A.A$ // LR(0) item

$S \rightarrow AA.$ // final item

S

Canonical collection represents the set of valid states for the LR parser

- a) Canonical LR (0) collection
- b) Canonical LR (1) collection

Canonical LR (0) collection helps to construct LR parsers like LR(0) & SLR parsers

To create Canonical LR (0) collection for Grammar, 3 things are required –

1. Augmented Grammar
2. Closure Function
3. goto Function

LR Parser or LR(k) parser-- terminology

Augmented grammar is a production rule where the start symbol appears only on the LHS of productions. It is used to identify when to stop parser & declare string as accepted.

Ex:- For the grammar $S \rightarrow AA$ & $A \rightarrow SA$

$S' \rightarrow S$ is the augmented grammar, here S' appears only on LHS

Closure(), it helps to find states of the automaton and to compute closure we need to add dot(.) to the RHS of production.

If after dot(.) we have to write Non-terminal symbol productions and we need to add dot(.) in RHS of these productions also.

Ex:-
 $S' \rightarrow S$
 $S \rightarrow AA$
 $A \rightarrow aA/b$

Closure($S' \rightarrow S$) = $S' \rightarrow .S$ → non terminal. So add production of S
 $S \rightarrow .AA$ → non terminal after dot. So add production of A
 $A \rightarrow .aA/.b$

Goto, it will help in constructing transitions of the automaton.

It performs to move on the dot(.) to the RHS of the symbol.

Syntax:----- Goto(old_state, NT)

Ex:-

Goto($A \rightarrow aA, a$) = $A \rightarrow a.A$ → apply closure
 $A \rightarrow .aA/.b$

non terminal after dot. So add production of A

LR Parser or LR(k) parser-- implementation

STEPS to implement LR(0) parser

1. For given Input string write CFG
2. Check the Ambiguity of the grammar
3. Add augment production to the given grammar
4. Create canonical collection of LR(0) items
5. Draw a data flow diagram
6. Construct LR(0) parsing table
7. Parse the input string

LR Parser or LR(k) parser-- Example

Example:-

$S \rightarrow AA$

$A \rightarrow aA \mid b$

STEP 1&2 :- CFG & ambiguity

Already satisfied

STEP3:- Add augment grammar

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

$S' \rightarrow \bullet S$ // Augment grammar

$S \rightarrow \bullet AA$

$A \rightarrow \bullet aA$

$A \rightarrow \bullet b$

LR Parser or LR(k) parser-- Example

STEP4:- canonical collection of LR(0) items

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 = $S \rightarrow A\bullet A$

S

$A \rightarrow \bullet aA$ //Add all productions of A in to I2 State because " \bullet " is followed by the non-terminal

$A \rightarrow \bullet b$

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

I3 = $A \rightarrow a\bullet A$ //Add all productions of A in to I2 State because " \bullet " is followed by the non-terminal

$A \rightarrow \bullet aA$

$A \rightarrow \bullet b$

LR Parser or LR(k) parser-- Example

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

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

Go to (I2, b) = Closure ($A \rightarrow b\bullet$) = (same as I4)

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

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

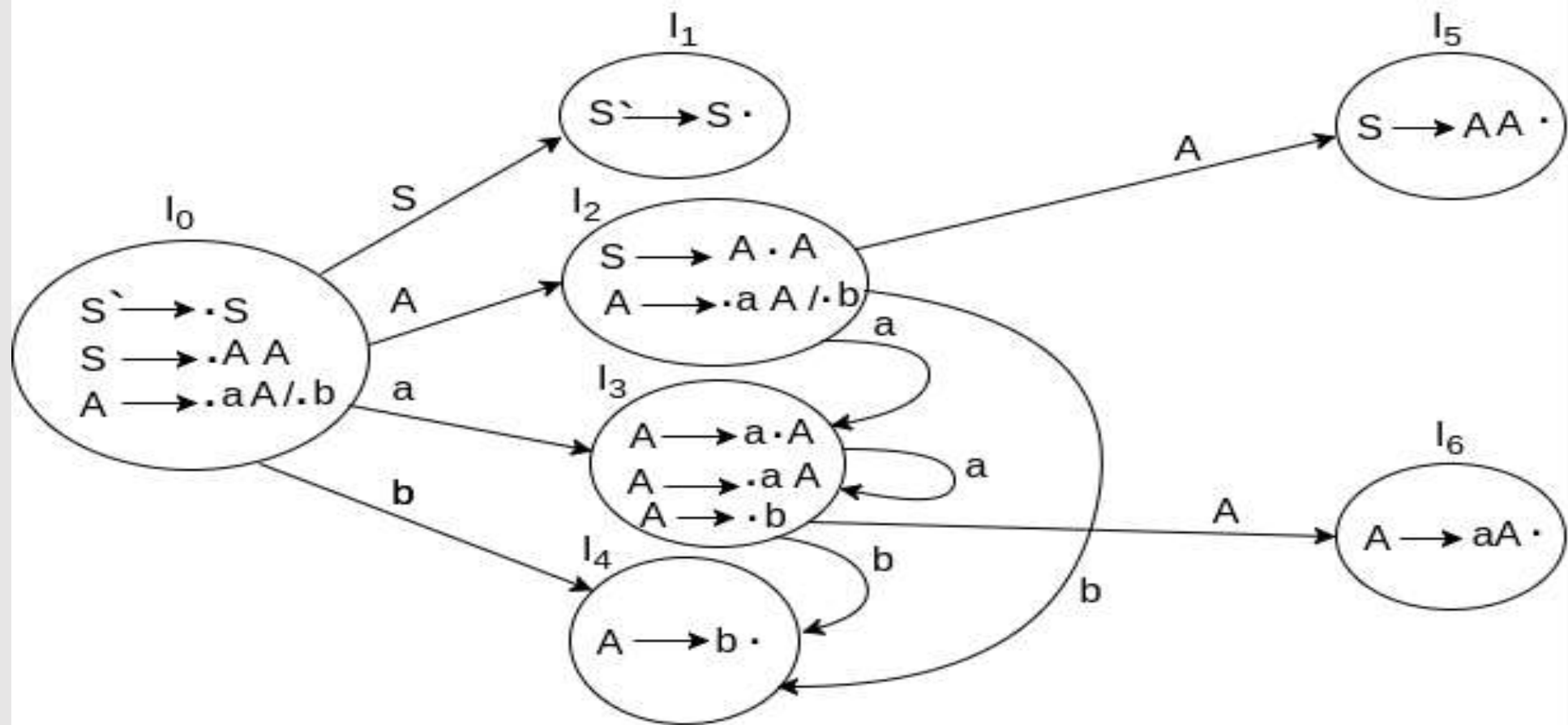
S

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

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

LR Parser or LR(k) parser-- Example

STEP5:-Draw the Data Flow Diagram



LR Parser or LR(k) parser-- Example

STEP6:-construct LR(0) Parsing Table

- If a state is going to some other state on a terminal then it correspond to a shift move.
- If a state is going to some other state on a non- terminal then it correspond to go to move.
- If a state contain the final item in the particular row then write the reduce node completely.

S

let

$S' \rightarrow S$ -----Accepted

$S \rightarrow AA$ -----r1

$A \rightarrow aA$ -----r2

$A \rightarrow b$ -----r3

LR Parser or LR(k) parser-- Example

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		

LR Parser or LR(k) parser-- Example

- For every push operation the pointer advances to next symbol

Map the stack top & start symbol of input string.

① if its a **shift** perform push operation **perform PUSH** the terminal & shift number on to the stack

② if its a **reduce** perform pop operation **perform POP 2 times the symbol on RHS of reduce**

NOTE:- After PUSH increment the pointer to point to next symbol in input string.

0 is pushed on to stack as its the initial state 0

let

$S' \rightarrow S$ -----accepted

$S \rightarrow AA$ -----r1

$A \rightarrow aA$ -----r2

$A \rightarrow b$ -----r3

As r3 is $A \rightarrow b$

we have to perform

POP 2 times the symbol on RHS of r3 i.e, $2 \times 1 = 2$ symbols to be popped. and push the LHS of r3

resulting 0a3aA

but top of stack should always be a number

so map the last two symbols 0a3aA and add the result (6) on top of stack 0a3aA6

Stack	Input	Action
0	aabb\$	$10 \rightarrow a, S3$
0a3	aa b \$	$13 \rightarrow a, S3$
0a3a3	aa b \$	$13 \rightarrow b, S4$
0a3a3b4	aa b \$	$14 \rightarrow b, r3$
0a3a3A6	aa b \$	$16 \rightarrow b, r2$
0a3A6	aa b \$	$16 \rightarrow b, r2$
0A2	aa b \$	$12 \rightarrow b, s4$
0A2b4	aa b \$	$14 \rightarrow \$, r3$
0A2A5	aa b \$	$15 \rightarrow \$, r1$
0s1	aa b \$	accept

① **PUSH** the terminal & shift number on to the stack

①

①

②

② r2 is $A \rightarrow aA$,
POP 4 symbols

①

② r3 is $A \rightarrow b$,
POP 2 symbols

② r1 is $S \rightarrow AA$,
POP 4 symbols

Mapping 1 on \$ we get accept

LR Parser or LR(k) parser-- Example

Explanation

- I0 on S is going to I1 so write it as 1.
- I0 on A is going to I2 so write it as 2.
- I2 on A is going to I5 so write it as 5.
- I3 on A is going to I6 so write it as 6.
- I0, I2 and I3 on a are going to I3 so write it as S3 which means that shift 3.
- I0, I2 and I3 on b are going to I4 so write it as S4 which means that shift 4.
- I4, I5 and I6 all states contains the final item because they contain • in the right most end. So rate the production as production number.
- I1 contains the final item which drives($S' \rightarrow S\bullet$), so action {I1, \$} = Accept.
- I4 contains the final item which drives $A \rightarrow b\bullet$ and that production corresponds to the production number 3 so write it as r3 in the entire row.
- I5 contains the final item which drives $S \rightarrow AA\bullet$ and that production corresponds to the production number 1 so write it as r1 in the entire row.
- I6 contains the final item which drives $A \rightarrow aA\bullet$ and that production corresponds to the production number 2 so write it as r2 in the entire row.

SLR parser

- SLR is simple LR.
- It is the smallest class of grammar having few number of states.
- SLR is very easy to construct and is similar to LR parsing.
- The only difference between SLR parser and LR(0) parser is that in LR(0) parsing table, we place **Reduce move only in the FOLLOW of LHS** not the entire row as in LR(0).

Steps for constructing the SLR parsing table :

1. For given Input string write CFG
2. Check the Ambiguity of the grammar
3. Writing augmented grammar
4. LR(0) collection of items to be found
5. Draw a data flow diagram
6. Find FOLLOW of LHS of production
7. Constructing parsing table
8. Parse the input string

s

SLR Parser- Example

Example:-

$S \rightarrow AA$

$A \rightarrow aA \mid b$

STEP 1&2 :- CFG & ambiguity

Already satisfied

STEP3:- Add augment grammar

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

$S' \rightarrow \bullet S$ // Augment grammar

$S \rightarrow \bullet AA$

$A \rightarrow \bullet aA$

$A \rightarrow \bullet b$

SLR Parser- Example

STEP4:- canonical collection of LR(0) items

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 = $S \rightarrow A \bullet A$

S

$A \rightarrow \bullet aA$ //Add all productions of A in to I2 State because " \bullet " is followed by the non-terminal

$A \rightarrow \bullet b$

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

I3 = $A \rightarrow a \bullet A$ //Add all productions of A in to I2 State because " \bullet " is followed by the non-terminal

$A \rightarrow \bullet aA$

$A \rightarrow \bullet b$

SLR Parser- Example

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

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

Go to (I2, b) = Closure ($A \rightarrow b\bullet$) = (same as I4)

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

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

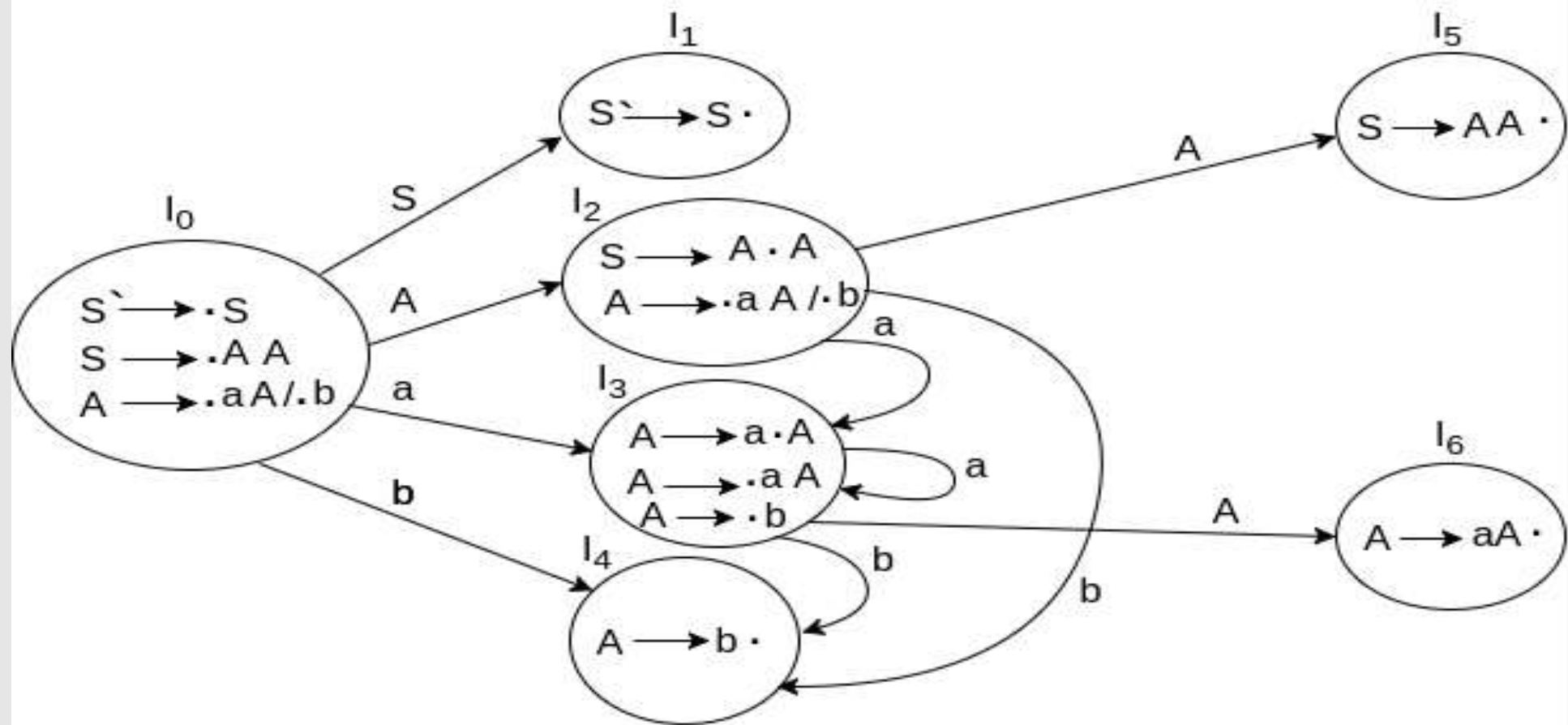
S

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

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

SLR Parser- Example

STEP5:-Draw the Data Flow Diagram



SLR Parser-Example

STEP6: –

Find FOLLOW of LHS of production

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

$\text{FOLLOW}(A) = a, b, \$$

let

$S' \rightarrow S$ ----- Accepted

$S \rightarrow AA$ -----r1

$A \rightarrow aA$ -----r2

$A \rightarrow b$ -----r3

S

SLR Parser- Example

STEP7:-construct SLR(0) Parsing Table

- If a state is going to some other state on a **terminal** then it correspond to a **shift move(S_n)**
- If a state is going to some other state on a **non-terminal** then it correspond to **go to move(n)**
- If a state contain the final item in the particular row then write the **reduce move only in the FOLLOW of LHS.**

States	Action			Go to	
	a	b	S	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		
I ₆	r2	r2	r2		

LR Parser or LR(k) parser-- Example

- For every push operation the pointer advances to next symbol

Map the stack top & start symbol of input string.

① if its a **shift** perform push operation **perform PUSH** the terminal & shift number on to the stack

② if its a **reduce** perform pop operation **perform POP 2 times the symbol on RHS of reduce**

NOTE:- After PUSH increment the pointer to point to next symbol in input string.

0 is pushed on to stack as its the initial state 0

let

$S' \rightarrow S$ -----accepted

$S \rightarrow AA$ -----r1

$A \rightarrow aA$ -----r2

$A \rightarrow b$ -----r3

As r3 is $A \rightarrow b$

we have to perform

POP 2 times the symbol on RHS of r3 i.e, $2 \times 1 = 2$ symbols to be popped. and push the LHS of r3

resulting 0a3aA

but top of stack should always be a number

so map the last two symbols 0a3aA and add the result (6) on top of stack 0a3aA6

Stack	Input	Action
0	aabb\$	$10 \rightarrow a, S3$
0a3	aa a bb\$	$13 \rightarrow a, S3$
0a3a3	aa b bb\$	$13 \rightarrow b, S4$
0a3a3b4	aa b b\$	$14 \rightarrow b, r3$
0a3a3A6	aa b b\$	$16 \rightarrow b, r2$
0a3A6	aa b b\$	$16 \rightarrow b, r2$
0A2	aa b b\$	$12 \rightarrow b, s4$
0A2b4	aa b \$	$14 \rightarrow \$, r3$
0A2A5	aa b \$	$15 \rightarrow \$, r1$
0s1	aa b \$	accept

① **PUSH** the terminal & shift number on to the stack

①

①

②

② r2 is $A \rightarrow aA$,
POP 4 symbols

①

② r3 is $A \rightarrow b$,
POP 2 symbols

② r1 is $S \rightarrow AA$,
POP 4 symbols

Mapping 1 on \$ we get accept

CLR Parser

- The CLR parser stands for canonical LR parser.
- It is a more powerful LR parser.
- It makes use of look-ahead symbols.
- This method uses a large set of items called LR(1) items.

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

RULES For Calculating Look-ahead

1. The look ahead for the argument production is always \$ symbol

Ex:-

$S' \rightarrow .S, \$$ // look-ahead is \$ as it's the augment production

2. When we have to calculate Look-ahead of Non-terminal(**NT**) in a production
See (**.NT**) in the previous production and calculate FIRST (remaining part after **.NT**)

Ex:-

$S' \rightarrow .S, \$$ // previous production .NT is .S

$S' \rightarrow .S, \$$

$S \rightarrow .AA, ?$ // **look-ahead = FIRST**(remaining part after .NT in previous production) i.e, FIRST(\$)= \$
resulting $S \rightarrow .AA, \$$

CLR Parser-- Implementation

STEPS to implement CLR(1) parser

1. For given Input string write CFG
2. Check the Ambiguity of the grammar
3. Add augment production to the given grammar
4. Create canonical collection of LR(1) items
5. Draw a data flow diagram
6. Construct CLR(1) parsing table
7. Parse the input string

CLR Parser-Example

Example:-

$S \rightarrow AA$

$A \rightarrow aA \mid b$

STEP 1&2 :- CFG & ambiguity

Already satisfied

STEP3:- Add augment grammar

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

$S' \rightarrow \bullet S$ // Augment grammar

$S \rightarrow \bullet AA$

$A \rightarrow \bullet aA$

$A \rightarrow \bullet b$

CLR Parser-- Example

STEP4:- canonical collection of LR(1) items

I0 = $S' \rightarrow \bullet S, \$$ // as it's the argument production

$S \rightarrow \bullet AA, \$$ //look-ahead= FIRST(remaining part after .NT in previous production) i.e, FIRST($\$$)= $\$$

$A \rightarrow \bullet aA, a/b$ //look-ahead= FIRST(remaining part after .NT in previous production) i.e, FIRST($A, \$$)= a/b

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

I1= Go to (**I0**, S) = closure ($S' \rightarrow S\bullet$) = $S' \rightarrow S\bullet, \$$ //write the look-ahead as it is in **I0**

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

S

I2 = $S \rightarrow A\bullet A, \$$ //write the look-ahead as it is in **I0**

$A \rightarrow \bullet aA, \$$ //Add all productions of A in to I2 State because " \bullet " is followed by the non-terminal

$A \rightarrow \bullet b, \$$ // look-ahead= FIRST(remaining part after .NT in previous production) i.e, FIRST($\$$)= $\$$

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

I3= $A \rightarrow a\bullet A, a/b$ //write the look-ahead as it is in **I0**

$A \rightarrow \bullet aA, a/b$ //Add all productions of A in to I2 State because " \bullet " is followed by the non-terminal

$A \rightarrow \bullet b, a/b$ // look-ahead= FIRST(remaining part after .NT in previous production) i.e, FIRST(a/b)= a/b

CLR Parser-- Example

I4= Go to (I0, b) = closure ($A \rightarrow b\bullet$) = $A \rightarrow b\bullet$, a/b //write the look-ahead as it is in **I0**

I5= Go to (I2, A) = Closure ($S \rightarrow AA\bullet$) = $S \rightarrow AA\bullet$, \$ //write the look-ahead as it is in **I2**

I6=Go to (I2,a) = Closure ($A \rightarrow a\bullet A$) = $A \rightarrow a\bullet A$, \$ //write the look-ahead as it is in **I2**

$A \rightarrow \bullet aA$, \$ // look-ahead=

$A \rightarrow \bullet d$, \$ //FIRST(remaining part after .NT in previous production)
i.e, FIRST(\$)= \$

S

I7=Go to (I2, b) = Closure ($A \rightarrow b\bullet$) = $A \rightarrow b\bullet$, \$ //write the look-ahead as it is in **I2**

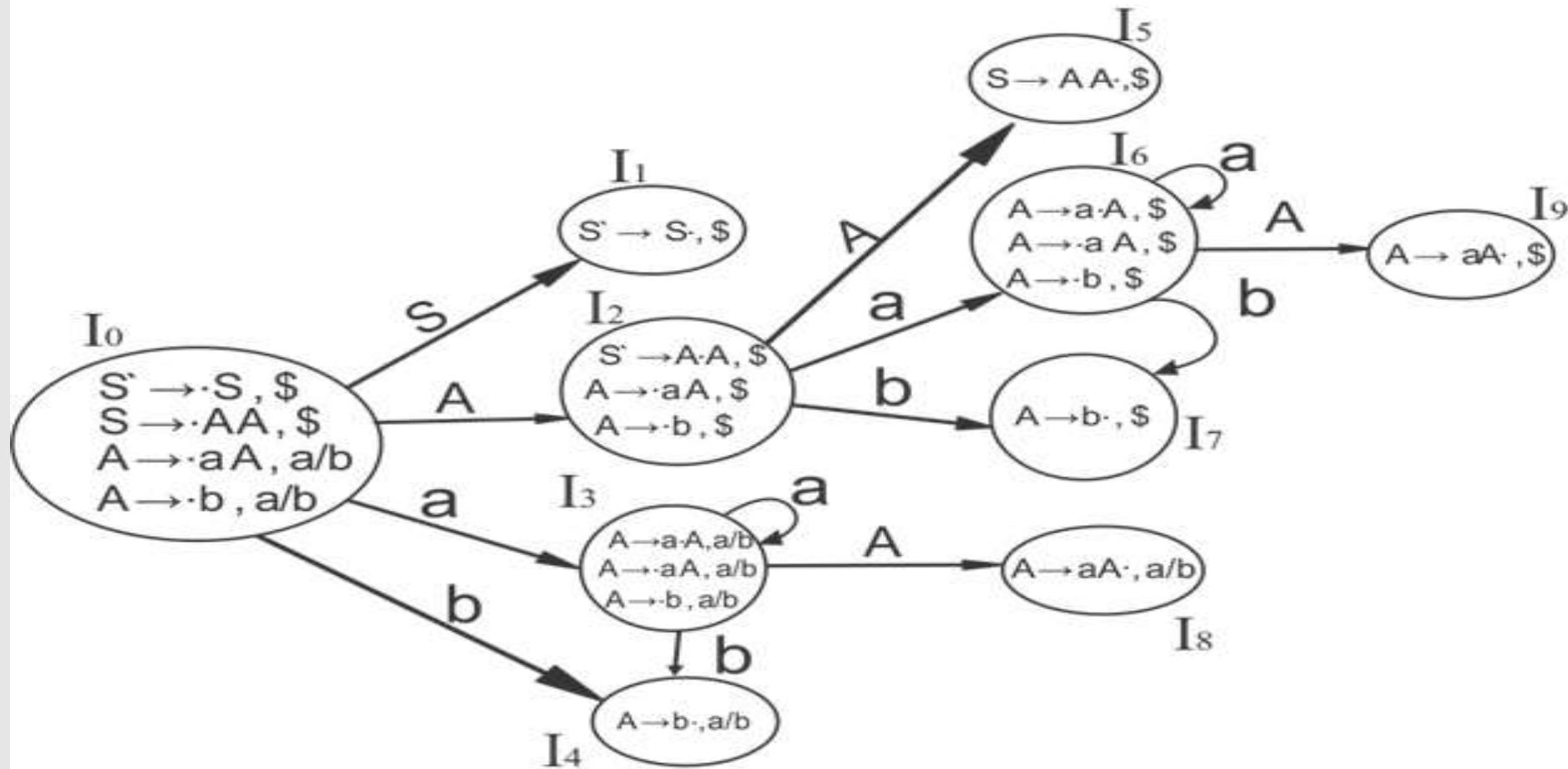
Go to (I3, a) = Closure ($A \rightarrow a\bullet A$) = (same as I3)

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

I8= Go to (I3, A) = Closure ($A \rightarrow aA\bullet$) = $A \rightarrow aA\bullet$, a/b //write the look-ahead as it is in **I3**

CLR Parser-- Example

STEP5:-Draw the Data Flow Diagram



CLR Parser--Example

STEP6:-construct CLR(1) Parsing Table

- If a state is going to some other state on a terminal then it correspond to a shift move.
- If a state is going to some other state on a non- terminal then it correspond to go to move.
- If a state contain the final item in the particular row then write the reduce node completely.

The only difference between SLR parser and CLR(1) parser is that in CLR(1) parsing table, we place **Reduce move only in the look-ahead symbols** not in the FOLLOW of LHS.

let

$S' \rightarrow S$ -----Accepted
 $S \rightarrow AA$ -----r1
 $A \rightarrow aA$ -----r2
 $A \rightarrow b$ -----r3

CLR Parser -- Example

States	a	b	\$	\$	A
I ₀	S ₃	S ₄			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		

CLR Parser -- Example

Explanation

The placement of shift node in CLR (1) parsing table is same as the SLR (1) parsing table. Only difference in the placement of reduce node.

I4 contains the final item which drives ($A \rightarrow b\bullet, a/b$), so action {I4, a} = R3, action {I4, b} = R3.

I5 contains the final item which drives ($S \rightarrow AA\bullet, \$$), so action {I5, \$} = R1.

I7 contains the final item which drives ($A \rightarrow b\bullet, \$$), so action {I7, \$} = R3.

I8 contains the final item which drives ($A \rightarrow aA\bullet, a/b$), so action {I8, a} = R2, action {I8, b} = R2.

I9 contains the final item which drives ($A \rightarrow aA\bullet, \$$), so action {I9, \$} = R2.

CLR Parser-- Example

Step7:-
parsing the input string

Map the stack top & start symbol of input string.

- ① if its a **shift** perform push operation **perform PUSH** the terminal & shift number on to the stack
- ② if its a **reduce** perform pop operation **perform POP** 2 times the symbol on RHS of reduce

NOTE:- After PUSH increment the pointer to point to next symbol in input string.

0 is pushed on to stack
as its the initial state I0

let

$S' \rightarrow S$ -----accepted
 $S \rightarrow AA$ -----r1
 $A \rightarrow aA$ -----r2
 $A \rightarrow b$ -----r3

As r3 is $A \rightarrow b$
 we have to perform
 POP 2 times the symbol
 on RHS of r3 i.e, $2 \times 1 = 2$ symbols to
 be popped, and push the LHS of r3
 resulting 0a3aA
 but top of stack should always be a
 number
 so map the last two symbols 0a3aA
 and add the result (8) on top of stack
 0a3aA8

- For every push operation the pointer advances to next symbol

Stack	Input	Action
0	aabb\$	I0 → a, S3
0a3	aabb\$	I3 → a, S3
0a3a3	aabb\$	I3 → b, S4
0a3a3b4	aabb\$	I4 → b, r3
0a3a3A8	aabb\$	I8 → b, r2
0a3A8	aabb\$	I8 → b, r2
0A2	aabb\$	I2 → b, s7
0A2b7	aabb\$	I7 → \$, r3
0A2A5	aabb\$	I5 → \$, r1
0s1	aabb\$	accept

①
PUSH the terminal & shift
number on to the stack

①

①

②

②
r2 is $A \rightarrow aA$,
POP 4 symbols

①

②
r3 is $A \rightarrow b$,
POP 2 symbols

②
r1 is $S \rightarrow AA$,
POP 4 symbols

Mapping 1 on \$ we get
accept

LALR Parser

- LALR refers to the look-ahead LR.
- It is the most powerful parser.
- It can handle large classes of grammar.
- To construct the LALR (1) parsing table, we use the canonical collection of LR (1) items.
- LALR (1) parsing is same as the CLR (1) parsing, only difference in the parsing table.

STEPS to implement LALR(1) parser

1. For given Input string write CFG
2. Check the Ambiguity of the grammar
3. Add augment production to the given grammar
4. Create canonical collection of LR(1) items
5. Draw a data flow diagram
6. Construct LALR(1) parsing table
7. Parse the input string

LALR Parser-Example

Example:-

$S \rightarrow AA$

$A \rightarrow aA \mid b$

STEP 1&2 :- CFG & ambiguity

Already satisfied

STEP3:- Add augment grammar

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

$S' \rightarrow \bullet S$ // Augment grammar

$S \rightarrow \bullet AA$

$A \rightarrow \bullet aA$

$A \rightarrow \bullet b$

LALR Parser-- Example

STEP4:- canonical collection of LR(1) items

I0 = $S' \rightarrow \bullet S, \$$ // as it's the argument production

$S \rightarrow \bullet AA, \$$ //look-ahead= FIRST(remaining part after .NT in previous production) i.e, FIRST($\$$)= $\$$

$A \rightarrow \bullet aA, a/b$ //look-ahead= FIRST(remaining part after .NT in previous production) i.e, FIRST($A, \$$)= a/b

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

I1= Go to (**I0**, S) = closure ($S' \rightarrow S\bullet$) = $S' \rightarrow S\bullet, \$$ //write the look-ahead as it is in **I0**

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

S

I2 = $S \rightarrow A\bullet A, \$$ //write the look-ahead as it is in **I0**

$A \rightarrow \bullet aA, \$$ //Add all productions of A in to I2 State because " \bullet " is followed by the non-terminal

$A \rightarrow \bullet b, \$$ // look-ahead= FIRST(remaining part after .NT in previous production) i.e, FIRST($\$$)= $\$$

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

I3= $A \rightarrow a\bullet A, a/b$ //write the look-ahead as it is in **I0**

$A \rightarrow \bullet aA, a/b$ //Add all productions of A in to I2 State because " \bullet " is followed by the non-terminal

$A \rightarrow \bullet b, a/b$ // look-ahead= FIRST(remaining part after .NT in previous production) i.e, FIRST(a/b)= a/b

LALR Parser-- Example

I4= Go to (I0, b) = closure ($A \rightarrow b\bullet$) = $A \rightarrow b\bullet$, a/b //write the look-ahead as it is in **I0**

I5= Go to (I2, A) = Closure ($S \rightarrow AA\bullet$) = $S \rightarrow AA\bullet$, \$ //write the look-ahead as it is in **I2**

I6=Go to (I2,a) = Closure ($A \rightarrow a\bullet A$) = $A \rightarrow a\bullet A$, \$ //write the look-ahead as it is in **I2**

$A \rightarrow \bullet aA$, \$ // look-ahead=

$A \rightarrow \bullet d$, \$ //FIRST(remaining part after .NT in previous production)
i.e, FIRST(\$)= \$

S

I7=Go to (I2, b) = Closure ($A \rightarrow b\bullet$) = $A \rightarrow b\bullet$, \$ //write the look-ahead as it is in **I2**

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

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

I8= Go to (I3, A) = Closure ($A \rightarrow aA\bullet$) = $A \rightarrow aA\bullet$, a/b //write the look-ahead as it is in **I3**

LALR 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 look-ahead, 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/\$
 }

s

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

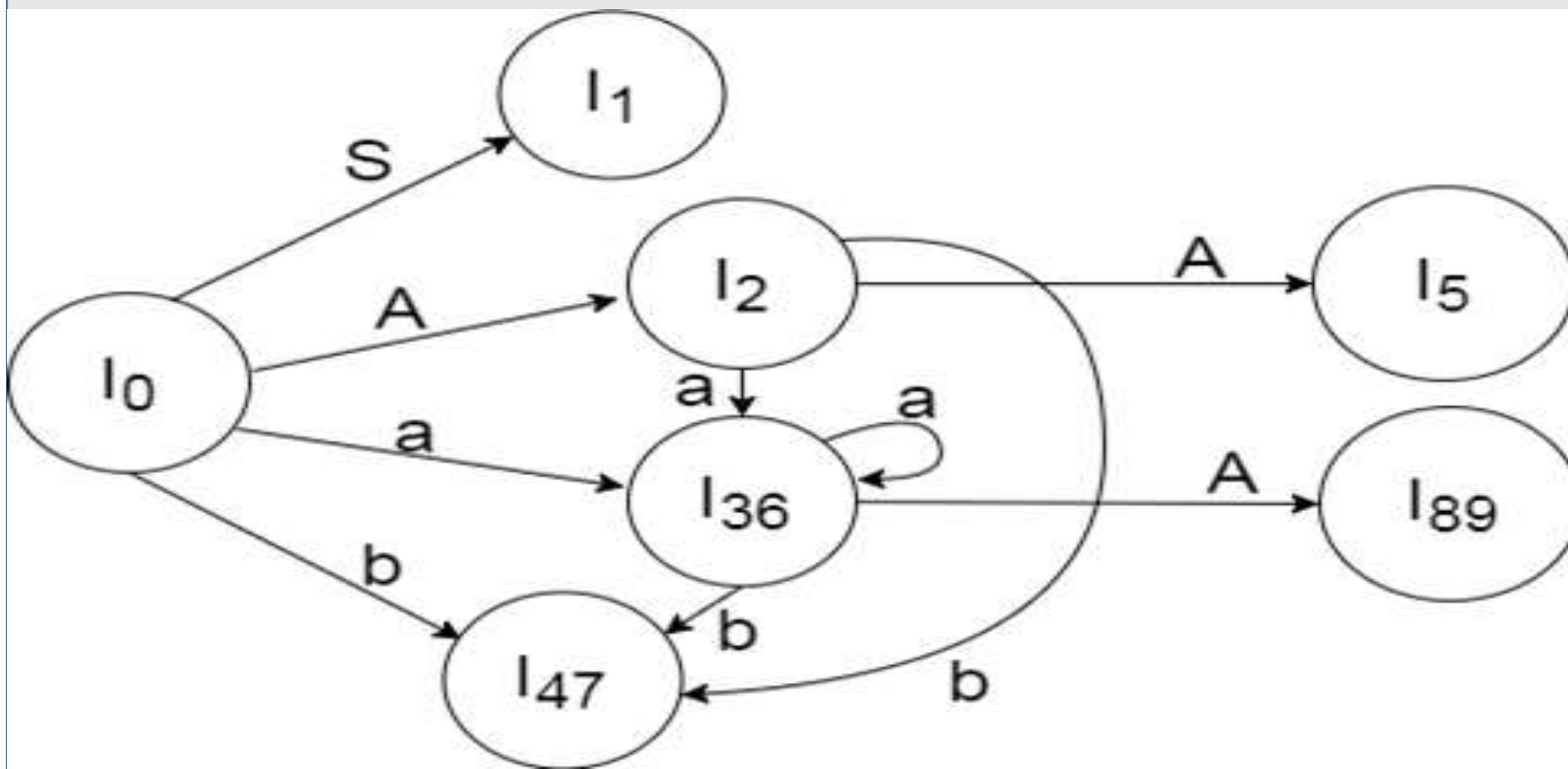
I47 = { $A \rightarrow b \bullet$, a/b/\$ }

The 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 Parser-- Example

STEP5:-Draw the Data Flow Diagram



LALR Parser--Example

STEP6:-construct LALR(1) Parsing Table

- If a state is going to some other state on a terminal then it correspond to a shift move.
- If a state is going to some other state on a non- terminal then it correspond to go to move.
- If a state contain the final item in the particular row then write the reduce node completely.

The only difference between CLR(1) parser and LALR(1) parser is that in LALR(1) parsing table, **we combine the two similar states but with different look-ahead.**

Ex:- **I4** = {A → b•, a/b} & **I7** = {A → b•, \$ } = **I47** = {A → b•, a/b/\$ }

let

S' → S -----Accepted

S → AA -----r1

A → aA -----r2

A → b -----r3

LALR Parser -- Example

States	a	b	S	S	A
I ₀	S ₃₆	S ₄₇		1	2
I ₁	accept				
I ₂	S ₃₆	S ₄₇			5
I ₃₆	S ₃₆	S ₄₇			89
I ₄₇	R ₃	R ₃	R ₃		
I ₅			R ₁		
I ₈₉	R ₂	R ₂	R ₂		

LALR Parser -- Example

Explanation

The placement of shift node in LALR (1) parsing table is same as the CLR (1) parsing table. Only difference is in LALR parsing table construction , we merge these similar states.

I3 and I6 are similar except their look-ahead.

I4 and I7 are similar except their look-ahead.

I8 and I9 are similar except their look-ahead.

Wherever there is 3 or 6, make it 36(combined form)

Wherever there is 4 or 7, make it 47(combined form)

Wherever there is 8 or 9, make it 89(combined form)

LALR Parser-- Example

Step7:- parsing the input string

Map the stack top & start symbol of input string.

① if its a **shift** perform push operation **perform PUSH** the terminal & shift number on to the stack

② if its a **reduce** perform pop operation **perform POP 2 times the symbol on RHS of reduce**

NOTE:- After PUSH increment the pointer to point to next symbol in input string.

0 is pushed on to stack
as its the initial state 10

let

$S' \rightarrow S$ -----accepted

$S \rightarrow AA$ -----r1

$A \rightarrow aA$ -----r2

$A \rightarrow b$ -----r3

As r3 is $A \rightarrow b$
we have to perform
POP 2 times the symbol
on RHS of r3 i.e, $2 \times 1 = 2$ symbols to
be popped. and push the LHS of r3

resulting 0a36a36A
but top of stack should always be a
number
so map the last two symbols 0a36a36A
and add the result (89) on top of stack
0a36a36A 89

- For every push operation the pointer advances to next symbol

Stack	Input	Action
0	aabb\$	10 \rightarrow a, S36
0a36	aabb\$	136 \rightarrow a, S36
0a36a36	aabb\$	136 \rightarrow b, S47
0a36a36b47	aabb\$	147 \rightarrow b, r3
0a3a36A89	aabb\$	189 \rightarrow b, r2
0a3A89	aabb\$	189 \rightarrow b, r2
0A2	aabb\$	12 \rightarrow b, S47
0A2b47	aabb\$	147 \rightarrow \$, r3
0A2A5	aabb\$	15 \rightarrow \$, r1
0s1	aabb\$	accept

①
PUSH the terminal & shift
number on to the stack

①

①

②

②

①

②

②

Mapping 1 on \$ we get
accept

r2 is $A \rightarrow aA$,
POP 4 symbols

r3 is $A \rightarrow b$,
POP 2 symbols

r1 is $S \rightarrow AA$,
POP 4 symbols