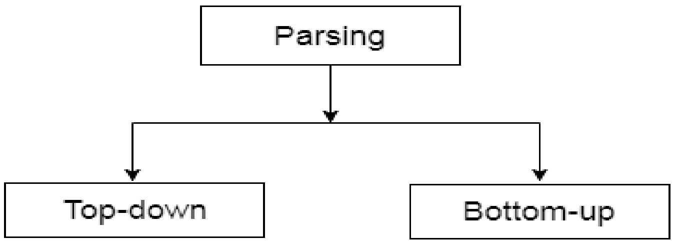
UNIT-2

PARSER

Parser is a compiler that is used to break the data into smaller elements coming from lexical analysis phase.

A parser takes input in the form of sequence of tokens and produces output in the form of parse tree.

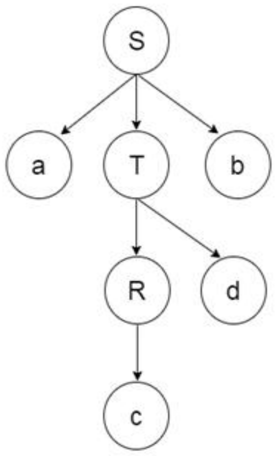
Parsing is of two types: top down parsing and bottom up parsing.



Top down paring

- The top down parsing is known as recursive parsing or predictive parsing.
- Bottom up parsing is used to construct a parse tree for an input string.
- 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:



Bottom up parsing

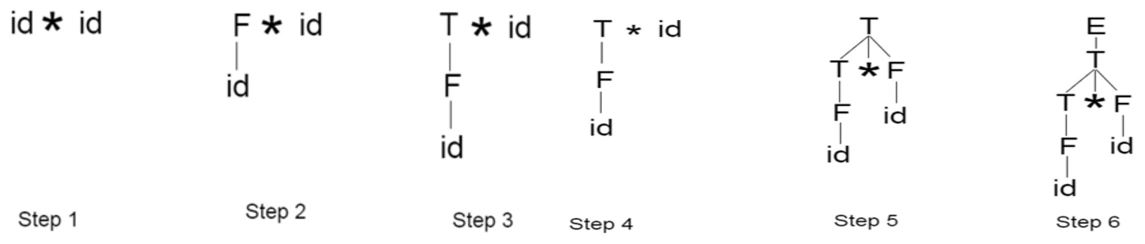
- Bottom up parsing is also known as shift-reduce parsing.
- Bottom up parsing is used to construct a parse tree for an input string.
- In the bottom up parsing, the parsing starts with the input symbol and construct the parse tree up to the start symbol by tracing out the rightmost derivations of string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

Example

Production

1. E → T
2. T → T * F
3. T → id
4. F → T
5. F → id

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



Bottom up parsing is classified in to various parsing. These are as follows:

1. Shift-Reduce Parsing
2. Operator Precedence Parsing
3. Table Driven LR Parsing

a. LR(1)
b. SLR(1)
c. CLR (1)
d. LALR(1)

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.

$$\text{A String} \xrightarrow[\text{reduce to}]{\qquad\qquad} \text{the starting symbol}$$

- Sift reduce parsing performs the two actions: shift and reduce. That's why it is known as shift reduces parsing.
- At the shift action, the current symbol in the input string is pushed to a stack.
- At each reduction, the symbols will be replaced by the non-terminals. The symbol is the right side of the production and non-terminal is the left side of the production.

Example:

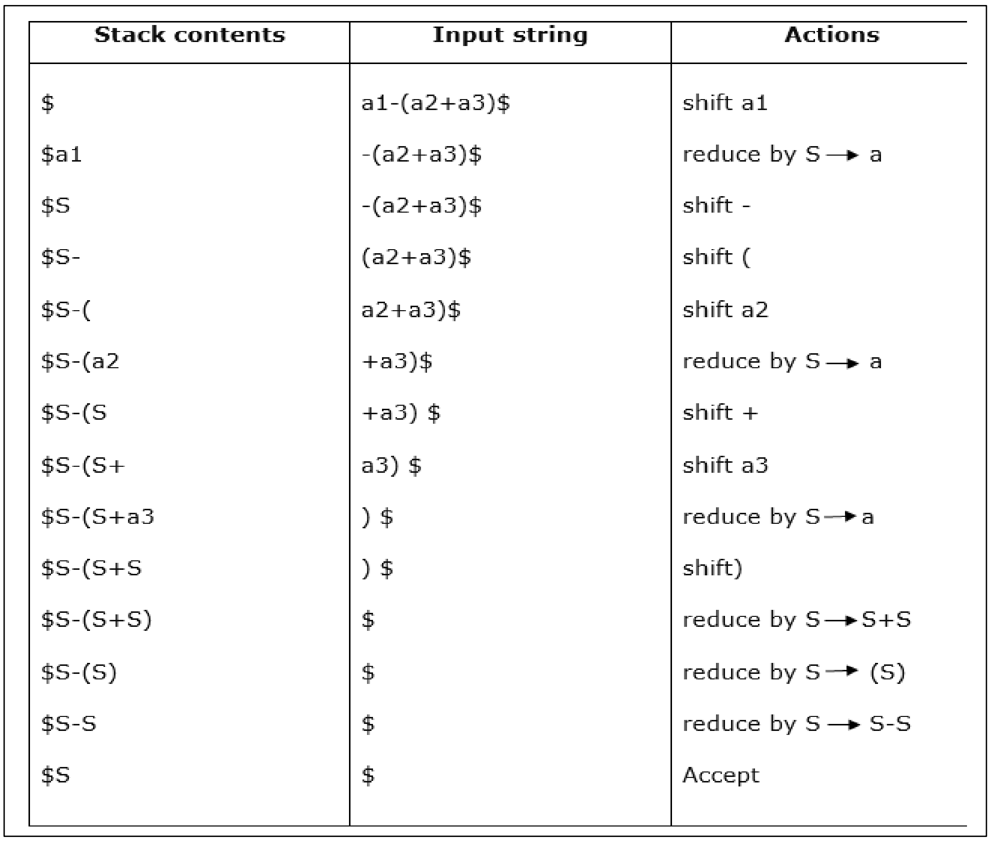
Grammar:

1. S → S+S
2. S → S-S
3. S → (S)
4. S → a

Input string:

1. a1-(a2+a3)

Parsing table:



Stack contents	Input string	Actions
$	a1-(a2+a3)$	shift a1
$a1	-(a2+a3)$	reduce by S → a
$S	-(a2+a3)$	shift -
$S-	(a2+a3)$	shift (
$S-(	a2+a3)$	shift a2
$S-(a2	+a3)$	reduce by S → a
$S-(S	+a3) $	shift +
$S-(S+	a3) $	shift a3
$S-(S+a3	) $	reduce by S→a
$S-(S+S	) $	shift)
$S-(S+S)	$	reduce by S→S+S
$S-(S)	$	reduce by S→ (S)
$S-S	$	reduce by S → S-S
$S	$	Accept

There are two main categories of shift reduce parsing as follows:

1. **Operator-Precedence Parsing**
2. **LR-Parser**

Operator precedence parsing

Operator precedence grammar is kinds of shift reduce parsing method. It is applied to a small class of operator grammars.

A grammar is said to be operator precedence grammar if it has two properties:

- No R.H.S. of any production has a∈.
- No two non-terminals are adjacent.

Operator precedence can only established between the terminals of the grammar. It ignores the non-terminal.

There are the three operator precedence relations:

a ⋗ b means that terminal "a" has the higher precedence than terminal "b".

a ⋖ b means that terminal "a" has the lower precedence than terminal "b".

a ≐ b means that the terminal "a" and "b" both have same precedence.

Precedence table:

	+	*	(	)	id	$
+	⋗	⋖	⋖	⋗	⋖	⋗
*	⋗	⋗	⋖	⋗	⋖	⋗
(	⋖	⋖	⋖	≐	⋖	X
)	⋗	⋗	X	⋗	X	⋗
id	⋗	⋗	X	⋗	X	⋗
$	⋖	⋖	⋖	X	⋖	X

Parsing Action

- Both end of the given input string, add the $ symbol.
- Now scan the input string from left right until the ⋗ is encountered.
- Scan towards left over all the equal precedence until the first left most ⋖ is encountered.
- Everything between left most ⋖ and right most ⋗ is a handle.
- $ on $ means parsing is successful.

Example

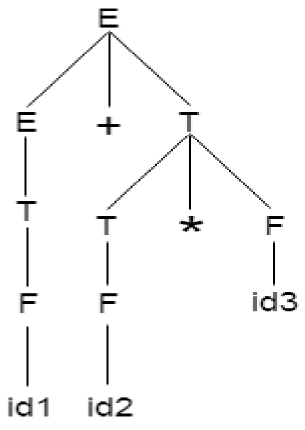
Grammar:

1. E → E+T/T
2. T → T*F/F
3. F → id

Given string:

1. w = id + id * id

Let us consider a parse tree for it as follows:



On the basis of above tree, we can design following operator precedence

	E	T	F	id	+	*	$
E	X	X	X	X	≐	X	⋗
T	X	X	X	X	⋗	≐	⋗
F	X	X	X	X	⋗	⋗	⋗
id	X	X	X	X	⋗	⋗	⋗
+	X	≐	⋖	⋖	X	X	X
*	X	X	≐	⋖	X	X	X
$	⋖	⋖	⋖	⋖	X	X	X

Now let us process the string with the help of the above precedence table:

$\$ \lessdot id1 \gtrdot + id2 * id3 \$$

$\$ \lessdot F \gtrdot + id2 * id3 \$$

$\$ \lessdot T \gtrdot + id2 * id3 \$$

$\$ \lessdot E \doteq + \lessdot id2 \gtrdot * id3 \$$

$\$ \lessdot E \doteq + \lessdot F \gtrdot * id3 \$$

$\$ \lessdot E \doteq + \lessdot T \doteq * \lessdot id3 \gtrdot \$$

$\$ \lessdot E \doteq + \lessdot T \doteq * \doteq F \gtrdot \$$

$\$ \lessdot E \doteq + \doteq T \gtrdot \$$

$\$ \lessdot E \doteq + \doteq T \gtrdot \$$

$\$ \lessdot E \gtrdot \$$

Accept.

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

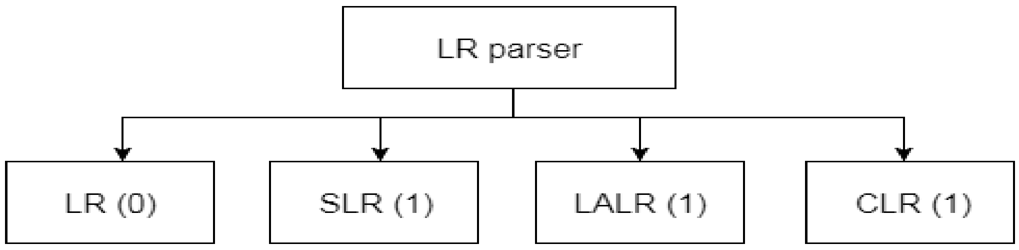


Fig: Types of LR parser

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.

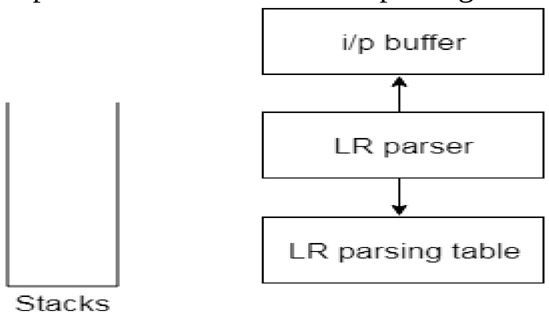


Fig: Block diagram of LR parser

Input buffer is used to indicate end of input and it contains the string to be parsed 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.

Augment 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

Given grammar

1. S → AA
2. A → aA | b

The Augment grammar G` is represented by

1. S`→ S
2. S → AA
3. A → aA | 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:

1. S → AA
2. A → aA | b

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

1. S` → •S
2. S → •AA
3. A → •aA
4. A → •b

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

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

Here, the Production is reduced so close the State.

I1= S` → S•

I2= Go to (I0, A) = closure (S → A•A)

Add all productions starting with A in to I2 State because "•" is followed by the non-terminal. So, the I2 State becomes

I2 =S→A•A
A → •aA
A → •b

Go to (I2,a) = Closure (A → a•A) = (same as I3)

Go to (I2, b) = Closure (A → b•) = (same as I4)

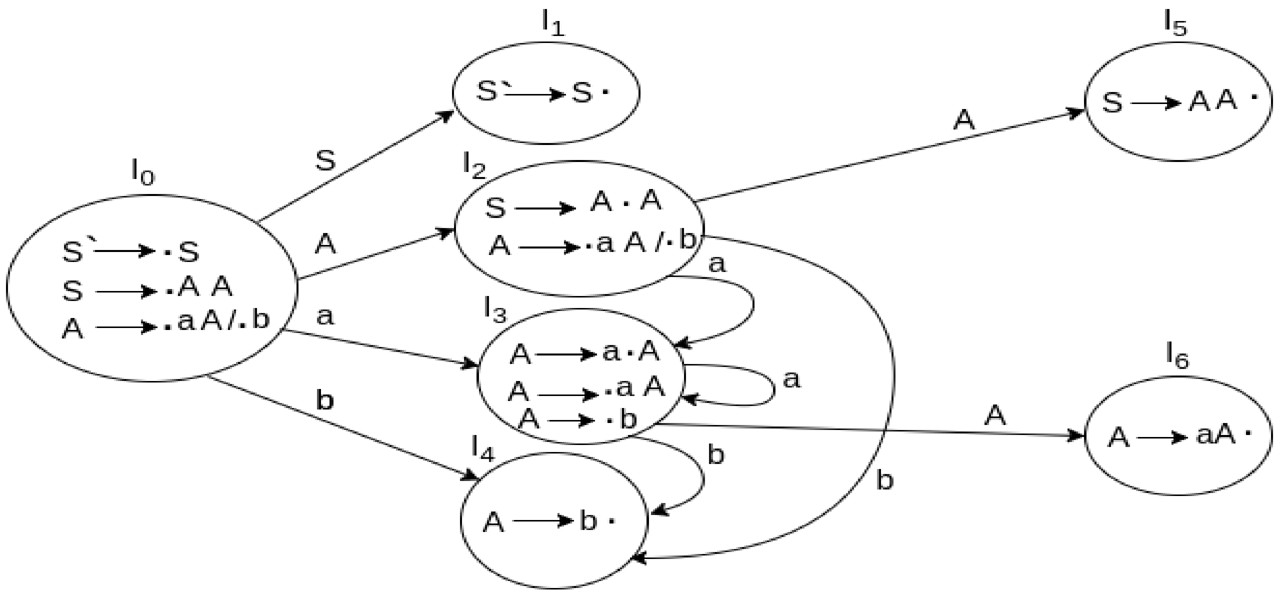
I3= Go to (I0,a) = Closure (A → a•A)

Add productions starting with A in I3.

A → a•A
A → •aA
A → •b

Go to (I3, a) = Closure (A → a•A) = (same as I3)
Go to (I3, b) = Closure (A → b•) = (same as I4)

I4= Go to (I0, b) = closure (A → b•) = A → b•
I5= Go to (I2, A) = Closure (S → AA•) = SA → A•
I6= Go to (I3, A) = Closure (A → aA•) = A → aA•
Drawing DFA:
The DFA contains the 7 states I0 to I6.



LR(0) 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 variable then it correspond to go to move.
- If a state contains the final item in the particular row then write the reduce node completely.

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		

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, I2and I3on 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 contain the final item because they contain • in the right most end. So, rate the production as production number.

Productions are numbered as follows:

1. S → AA ... (1)
2. A → aA ... (2)
3. A → b ... (3)

- I1 contains the final item which drives (S` → S•), so action {I1, $} = Accept.
- I4 contains the final item which drives A → b• 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 → AA• 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 → aA• and that production corresponds to the production number 2 so write it as r2 in the entire row.

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.

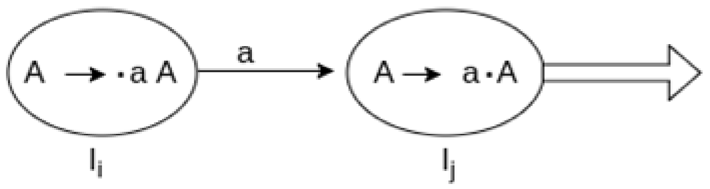
In the SLR (1) parsing, we place the reduce move only in the follow of left-hand side.
Various steps involved in the SLR (1) Parsing:

- 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

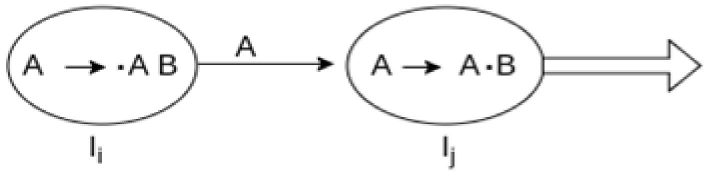
The steps which use to construct SLR (1) Table is given below:

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			

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.



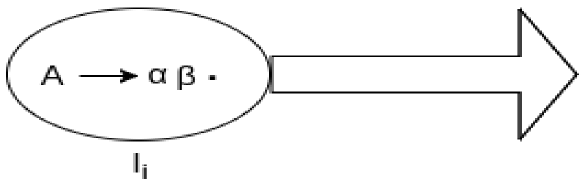
States	Action		Go to
	a	$	A
I_i			j
I_j			

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

1. S -> •Aa
2. A->αβ•

1. Follow(S) = {$}
2. Follow (A) = {a}



States	Action			Go to	
	a	b	$	S	A
I_i	r2				

SLR (1) Grammar

S → E

E → E + T | T

T → T * F | F

F → id

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

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

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

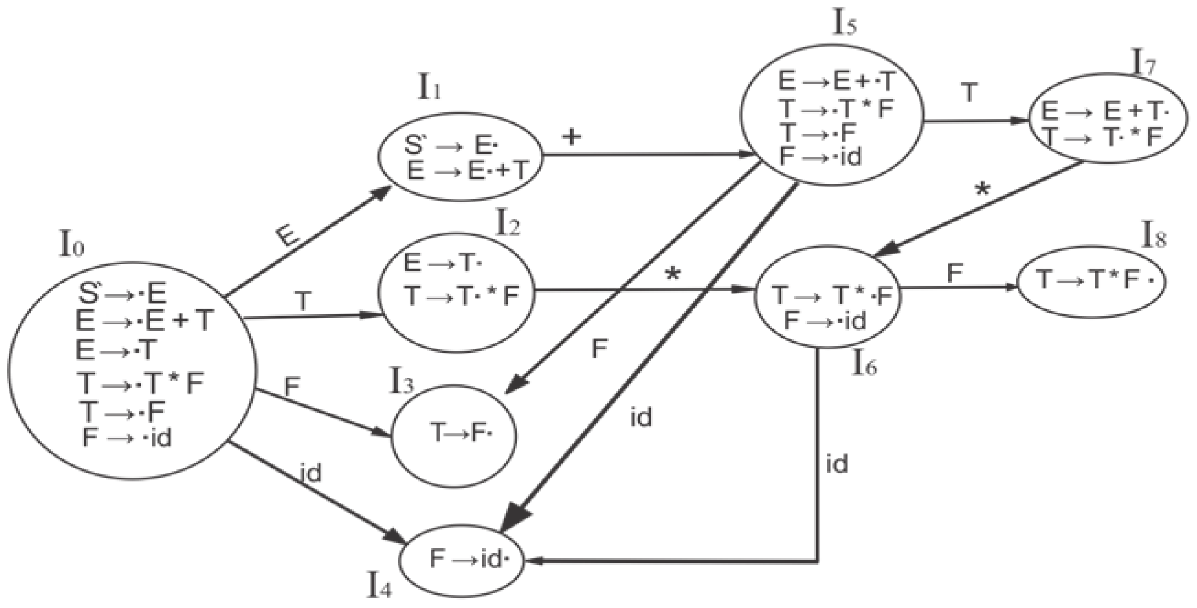
Add all productions starting with T and F in I5 State because "." is followed by the non-terminal. So, the I5 State becomes
I5 = E → E +•T
T → •T * F
T → •F
F → •id
Go to (I5, F) = Closure (T → F•) = (same as I3)
Go to (I5, id) = Closure (F → id•) = (same as I4)

I6= Go to (I2, *) = Closure (T → T * •F)
Add all productions starting with F in I6 State because "." is followed by the non-terminal. So, the I6 State becomes

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

I7= Go to (I5, T) = Closure (E → E + T•) = E → E + T•
I8= Go to (I6, F) = Closure (T → T * F•) = T → T * F•

Drawing DFA:



SLR (1) 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			

Explanation:

First (E) = First (E + T) ∪ First (T)
First (T) = First (T * F) ∪ First (F)
First (F) = {id}
First (T) = {id}
First (E) = {id}
Follow (E) = First (+T) ∪ {$} = {+, $}
Follow (T) = First (*F) ∪ First (F)
= {*, +, $}
Follow (F) = {*, +, $}

- I1 contains the final item which drives S → E• and follow (S) = {$}, so action {I1, $} = Accept
- I2 contains the final item which drives E → T• and follow (E) = {+, $}, so action {I2, +} = R2, action {I2, $} = R2
- I3 contains the final item which drives T → F• and follow (T) = {+, *, $}, so action {I3, +} = R4, action {I3, *} = R4, action {I3, $} = R4
- I4 contains the final item which drives F → id• and follow (F) = {+, *, $}, so action {I4, +} = R5, action {I4, *} = R5, action {I4, $} = R5
- I7 contains the final item which drives E → E + T• and follow (E) = {+, $}, so action {I7, +} = R1, action {I7, $} = R1
- I8 contains the final item which drives T → T * F• and follow (T) = {+, *, $}, so action {I8, +} = R3, action {I8, *} = R3, action {I8, $} = R3.

CLR (1) Parsing

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 greater number of states as compare to the SLR (1) parsing.

In the CLR (1), we place the reduce node only in the lookahead symbols.

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

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.

Example

CLR (1) Grammar

1. S → AA
2. A → aA
3. A → b

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

1. S` → •S, $
2. S → •AA, $
3. A → •aA, a/b
4. A → •b, a/b

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

I1= Go to (I0, S) = closure (S` → S•, $) = S` → S•, $
I2= Go to (I0, A) = closure (S → A•A, $)

Add all productions starting with A in I2 State because "." is followed by the non-terminal. So, the I2 State becomes

I2= S → A•A, $
A → •aA, $
A → •b, $

I3= Go to (I0, a) = Closure (A → a•A, a/b)
Add all productions starting with A in I3 State because "." is followed by the non-terminal. So, the I3 State becomes

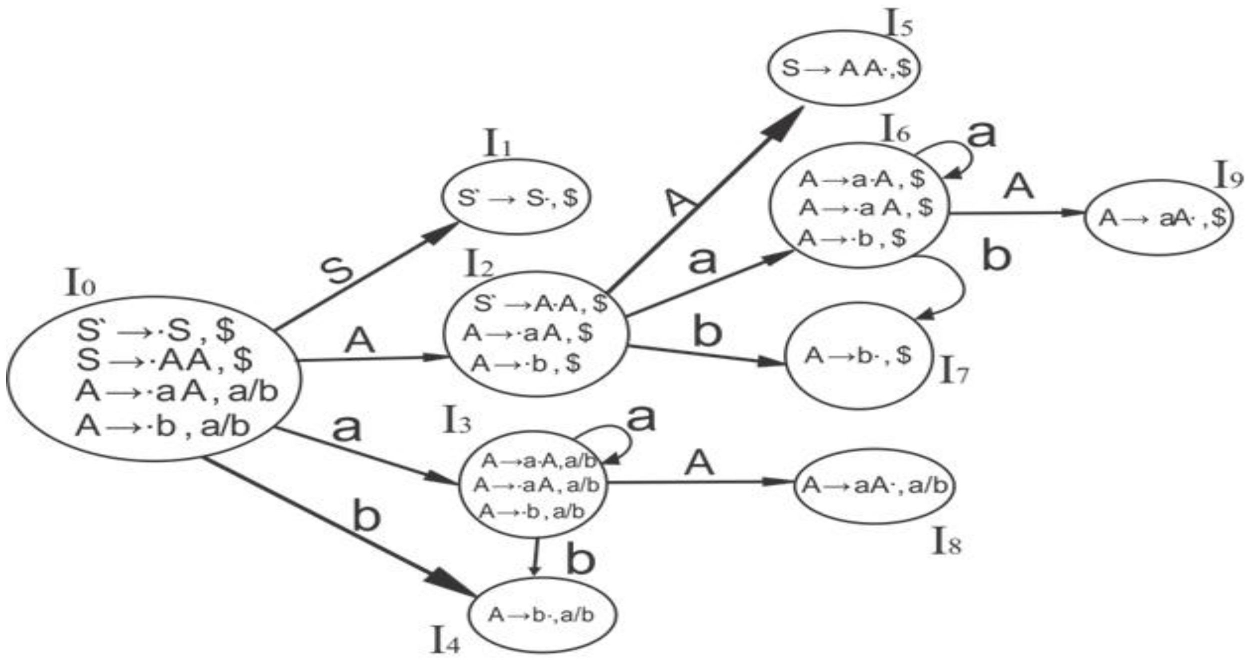
I3= A → a•A, a/b
A → •aA, a/b
A → •b, a/b
Go to (I3, a) = Closure (A → a•A, a/b) = (same as I3)
Go to (I3, b) = Closure (A → b•, a/b) = (same as I4)

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, $)

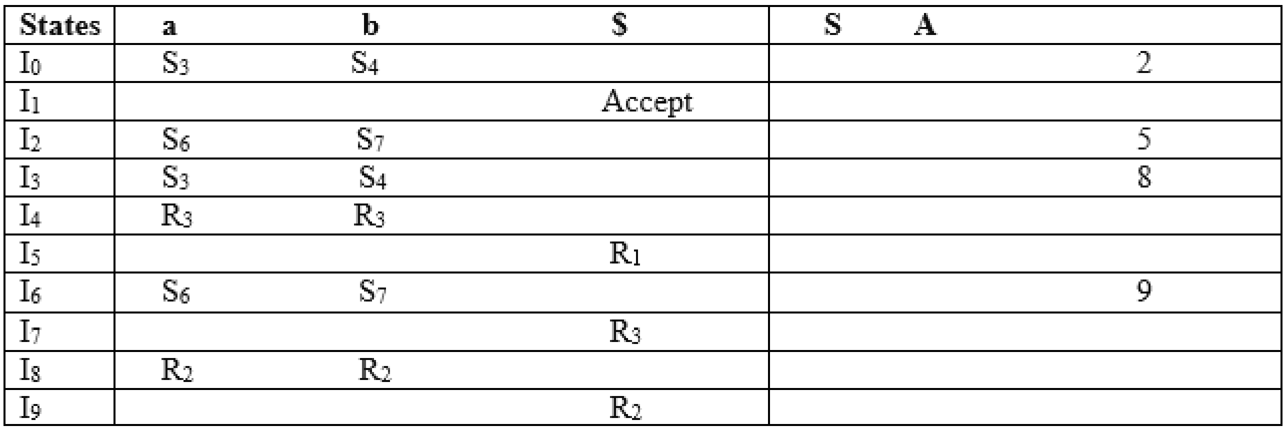
Add all productions starting with A in I6 State because "." is followed by the non-terminal. So, the I6 State becomes

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

I7= Go to (I2, b) = Closure (A → b•, $) = A → b•, $
I8= Go to (I3, A) = Closure (A → aA•, a/b) = A → aA•, a/b
I9= Go to (I6, A) = Closure (A → aA•, $) = A → aA•, $
Drawing DFA:



CLR (1) Parsing table:



States	a	b	$	S	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		

Productions are numbered as follows:

1. S → AA ... (1)
2. A → aA .... (2)
3. A → b ... (3)

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 → b•, a/b), so action {I4, a} = R3, action {I4, b} = R3.

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

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

I8 contains the final item which drives (A → aA•, a/b), so action {I8, a} = R2, action {I8, b} = R2.

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

LALR (1) Parsing:

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 is combined to form a single set of items

LALR (1) parsing is same as the CLR (1) parsing, only difference in the parsing table.

Example

LALR (1) Grammar

1. S → AA
2. A → aA
3. A → b

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

1. S` → •S, $
2. S → •AA, $
3. A → •aA, a/b
4. A → •b, a/b

I0 State:

Add Augment production to the I0 State and Compute the ClosureL

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

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

I2= Go to (I0, A) = closure (S → A•A, $)

Add all productions starting with A in I2 State because "•" is followed by the non-terminal. So, the I2 State becomes

I2= S → A•A, $

A → •aA, $

A → •b, $

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

Add all productions starting with A in I3 State because "•" is followed by the non-terminal. So, the I3 State becomes

I3= A → a•A, a/b

A → •aA, a/b

A → •b, a/b

Go to (I3, a) = Closure (A → a•A, a/b) = (same as I3)

Go to (I3, b) = Closure (A → b•, a/b) = (same as I4)

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, $)

Add all productions starting with A in I6 State because "•" is followed by the non-terminal. So, the I6 State becomes

I6 = A → a•A, $

A → •aA, $

A → •b, $

Go to (I6, a) = Closure (A → a•A, $) = (same as I6)

Go to (I6, b) = Closure (A → b•, $) = (same as I7)

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

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

I9= Go to (I6, A) = Closure (A → aA•, $) A → aA•, $

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

I3 = {A → a•A, a/b

A → •aA, a/b

A → •b, a/b

}

I6= {A → a•A, \$
A → •aA, \$
A → •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 → a•A, a/b/\$
A → •aA, a/b/\$
A → •b, a/b/\$
}

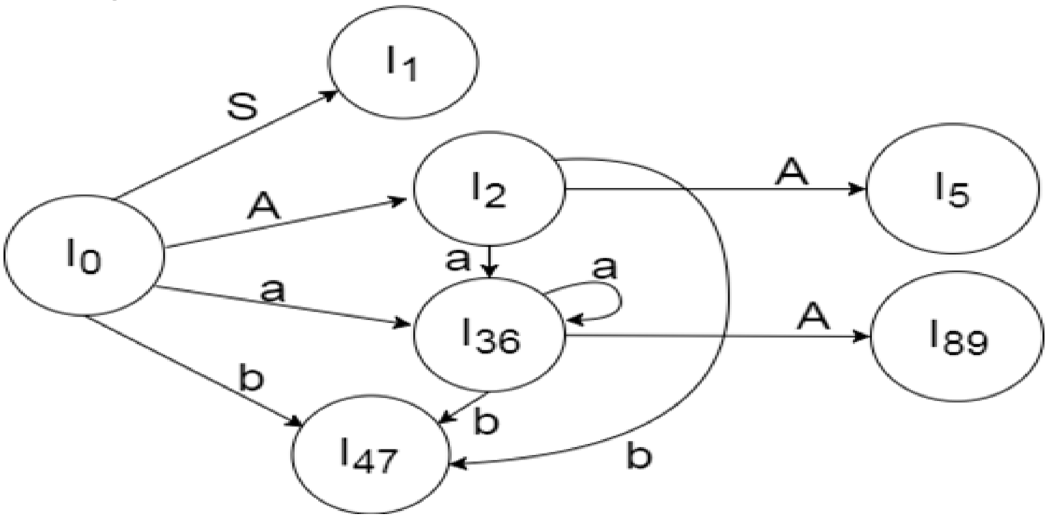
The I4 and I7 are same but they differ only in their look ahead, so we can combine them and called as I47.

I47 = {A → b•, 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 → aA•, a/b/\$}

Drawing DFA:



LALR (1) Parsing table:

States	a	b	\$	S	A
I_0	S_{36}	S_{47}		12	
I_1		accept			
I_2	S_{36}	S_{47}			5
I_{36}	$S_{36}S_{47}$				89
I_{47}	R_3R_3	R_3			
I_5			R_1		
I_{89}	R_2	R_2	R_2		

Automatic Parser Generator

YACC is an automatic tool that generates the parser program.
As we have discussed YACC in the first unit of this tutorial so you can go through the concepts again to make things more clear.

- YACC