UNIT – III

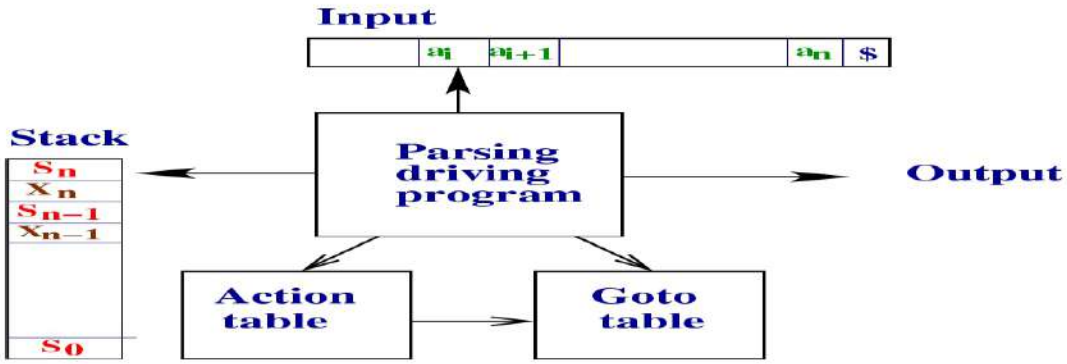
LR Grammar

***LR* Parsers**

KEY IDEAS.

- The LR parsing method is the most general non-backtracking shift-reduce parsing method known.
- LR parsers can parse a larger class of grammars than (top-down) predictive parsers.
- LR parsers usually recognize all programming language constructs specified by context-free grammar.
- LR parsers detect errors fast.
- Drawback: it is too much work to construct an LR parser by hand.
- Fortunately, we can use an LR parser generator such as YACC.

THE *LR* PARSING PRINCIPLE



An LR-Parser uses

- states to *memorize* information during the parsing process,
- an *action* table to make decisions (such as *shift* or *reduce*) and to compute states
- a *goto* table to compute states

These states are embedded with grammar symbols in a stack. More precisely, after each iteration, the stack stores a word of the form s0X1s1X2 ... sm-1Xmsm where s0 is the end-of-stack symbol, and so is the state on top of the stack.

For a state *s* and a terminal *a*, the entry *action*[*s*, *a*] has one of the four following forms

- *shift s'* where *s'* is a state,
- *reduce A* -> b,
- *accept*,
- *error*.

Algorithm

Input: A LR-Parser for an unambiguous context-free grammar G over an alphabet Σ and a word $w \in \Sigma^*$.

Output: An error if $w \notin L(G)$ or a rightmost derivation for w otherwise.

```
Set the cursor to the rightmost symbol of w$
push the initial state s0 on top of the empty stack
repeat
  let s be the state on top of the stack
  let a be the current pointed symbol in w$
  if action[s, a] = shift s' then
    push a on top of the stack
    push s' on top of the stack
    advance the cursor to the next symbol on the right in w$
  else if action[s, a] = reduce A ⟼ β then
    pop 2 | β | symbols of the stack
    let s' be the state on top of the stack
    push A on top of the stack
    push goto[s', A] on top of the stack
    output A ⟼ β
  else if action[s, a] = accept then
    return
  else
    error
```

Operator Grammar and Precedence Parsing

If the grammar satisfies the following two conditions, then we can say that type of grammar is called operator precedence grammar.

- If ε is on its RHS, no production rule exists.
- If two non-terminals are adjacent on its RHS, no production rule exists.

Operator Grammars have the property that no production right side is empty or has two adjacent non-terminals.

Example

E-> E A E | id

A-> + | *

The above grammar is not an operator grammar, but we can convert that grammar into operator grammar like –

E-> E + E | E * E | id

There are three precedence relations, which are given below –

Relation	Meaning
$a \lessdot b$	a yields precedence to b
$a = \cdot b$	a has the same precedence as b

Relation	Meaning
$a \gtrdot b$	*takes* precedence over *b*

Precedence Table

	id	+	*	$
id		⋗	⋗	⋗
+	⋖	⋗	⋖	⋗
*	⋖	⋗	⋗	⋗
$	⋖	⋖	⋖	⋗

Precedence Table

Example

The input string is as follows −

id1 + id2 * id3

After inserting precedence relations is−

<·id1·>+<·id2·>∗<·id3

Basic Principle

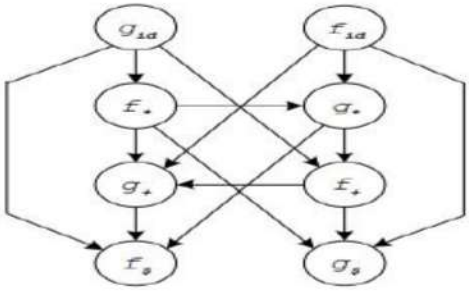
- Scan the string from the left until seeing ·> and put a pointer.
- Scan backward the string from right to left until seeing <·
- Everything between the two relations <· and ·> forms the handle.
- Replace the handle with the head of the production.
- Operator Precedence Parsing Algorithm
- The algorithm is as follows −
- Initialize: Set P to point to the first symbol of the input string w$
- Repeat − Let b be the top stack symbol and a be the input symbol pointed to by P.
- if (a is and b
- return
- else
- if a ·> b or a =· b then
- push a onto the stack
- advance P to the next input symbol
- else if a <· b then
- repeat
- c -> pop the stack
- until (c .> stack-top)

- else error
- end

Example

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

- Construct a graph using the algorithm. This graph is as follows −

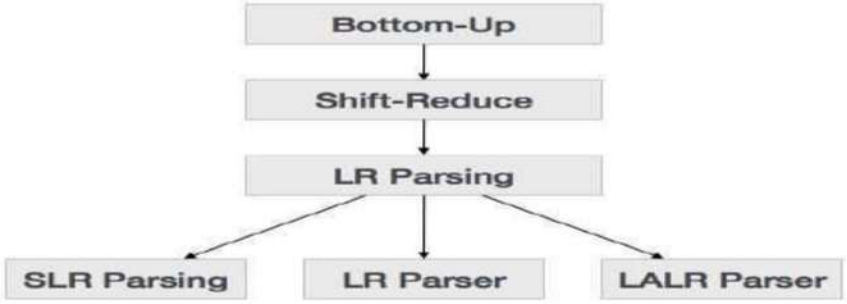


By seeing this, we can extract the precedence function like −

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

Bottom Up Parser

Bottom-up parsing starts from the leaf nodes of a tree and works upward until it reaches the root node. Here, we start from a sentence and then apply production rules in reverse order to get the start symbol. The image given below depicts the bottom-up parsers available.



Shift-Reduce Parsing

Shift-reduce parsing uses two unique steps for bottom-up parsing. These steps are known as shift-step and reduce-step.

- **Shift step**: The shift step advances the input pointer to the next input symbol, the shifted symbol. This symbol is pushed onto the stack. The shifted symbol is treated as a single node of the parse tree.
- **Reduce step**: When the parser finds a complete grammar rule (RHS) and replaces it with (LHS), it is known as the reduce step. This occurs when the top of the stack contains a handle. To reduce, a POP function is performed on the stack, which pops off the handle and replaces it with an LHS non-terminal symbol.

LR Parser

The LR parser is a non-recursive, shift-reduce, bottom-up parser. It uses a broad class of context-free grammar, making it the most efficient syntax analysis technique. LR parsers are also known as LR(k), where L stands for left-to-right scanning of the input stream, R stands for the construction of right-most derivation in reverse, and k denotes the number of lookahead symbols to make decisions.

There are three widely used algorithms available for constructing an LR parser:

- SLR(1) – Simple LR Parser:
  - Works on the smallest class of grammar
  - Few number of states, hence the tiny table
  - Simple and fast construction
- LR(1) – LR Parser:
  - Works on complete set of LR(1) Grammar
  - Generates large table and large number of states
  - Slow construction
- LALR(1) – Look-Ahead LR Parser:
  - Works on intermediate-size of grammar
  - Number of states is the same as in SLR(1)

LR Parsing Algorithm

Here, we describe a skeleton algorithm of an LR parser:

```
token = next_token()
repeat forever
   s = top of stack
   if action[s, token] = “shift si” then
      PUSH token
      PUSH si
      token = next_token()
else if action[s, token] = “reduce A::= β“ then
      POP 2 * |β| symbols
      s = top of stack
      PUSH A
      PUSH goto[s,A]
else if action[s, token] = “accept” then
      return
else
      error()
```

LL vs. LR

LL	LR
Does a leftmost derivation.	Does a rightmost derivation in reverse?
Starts with the root nonterminal on the stack.	Ends with the root nonterminal on the stack.
It ends when the stack is empty.	Starts with an empty stack.
It uses the stack to designate what is still to be expected.	It uses the stack to designate what is already seen.
Builds the parse tree top-down.	Builds the parse tree bottom-up.
Continuously pops a nonterminal off the stack and pushes the corresponding right-hand side.	Tries to recognize a right-hand side on the stack, pops it, and pushes the corresponding nonterminal.
Expands the non-terminals.	Reduces the non-terminals.
It reads the terminals when it pops one of the stacks.	Reads the terminals while it pushes them on the stack.
Pre-order traversal of the parse tree.	Post-order traversal of the parse tree.

LR(0) parsers

Problem on LR(0) parser

The LR parser is an efficient bottom-up syntax analysis technique that can be used for a large class of context-free grammar. This technique is also called LR(0) parsing.

L stands for left-to-right scanning.

R stands for rightmost derivation in reverse.

0 stands for no. of input symbols of lookahead.

Augmented grammar :

If G is a grammar with the starting symbol S, then G' (augmented grammar for G) is a grammar with a new starting symbol S‘ and productions S'-> .S. This new starting production aims to indicate to the parser when it should stop parsing. The ‘ . ‘ before S suggests that the left side of. ‘‘ has been read by a compiler, and the proper side of. ‘‘has yet to be read.

Steps for constructing the LR parsing table :

1. Writing augmented grammar
2. LR(0) collection of items to be found
3. Defining 2 functions: goto(list of non-terminals) and action(list of terminals) in the parsing table.

Q. Construct an LR parsing table for the given context-free grammar –

S–>AA

A–>aA|b

Solution :

STEP 1- Find augmented grammar –

The augmented grammar of the given grammar is:-

S'–>.S [0th production]

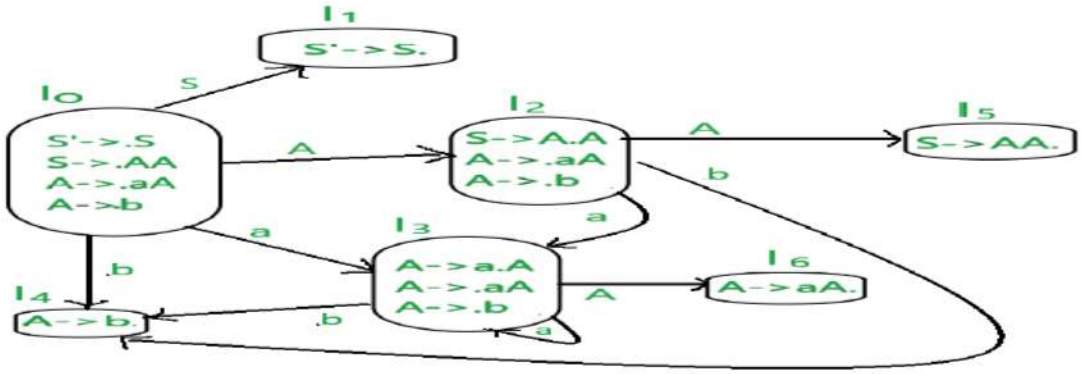
S–>.AA [1st production]

A–>.aA [2nd production]

A–>.b [3rd production]

STEP 2 – Find LR(0) collection of items

Below is the figure showing the LR(0) collection of items. We will understand everything one by one.



The terminals of this grammar are {a,b}

The non-terminals of this grammar are {S, A}

RULE – if any nonterminal has ' . ' preceding it, we must write all its production and add. '' preceding each output.

RULE – from each state to the next state, the ' . ' shifts to one place to the right.

- In the figure, I0 consists of augmented grammar.
- Io goes to I1 when ' . ' of 0th production is shifted towards the right of S(S'->S.). This state is the accepted state.
  The compiler sees s
- Io goes to I2 when ' . ' of 1st production is shifted towards the right (S->A.A) . The compiler sees a
- I0 goes to I3 when ' . ' of the 2nd production is shifted towards the right (A->a.A) . the compiler sees a.
- I0 goes to I4 when ' . ' of the 3rd production is shifted towards the right (A->b.) . the compiler sees b.
- I2 goes to I5 when ' . ' of 1st production is shifted towards the right (S->AA.) . The compiler sees a
- I2 goes to I4 when ' . ' of 3rd production is shifted towards the right (A->b.) . the compiler sees b.
- I2 goes to I3 when ' . ' of the 2nd production is shifted towards the right (A->a.A) . the compiler sees a.
- I3 goes to I4 when ' . ' of the 3rd production is shifted towards the right (A->b.) . the compiler sees b.
- I3 goes to I6 when ' . ' of 2nd production is shifted towards the right (A->aA.) . The compiler sees a
- I3 goes to I3 when ' . ' of the 2nd production is shifted towards the right (A->a.A) . the compiler sees a.

STEP3 – defining 2 functions: goto[list of non-terminals] and action[list of terminals] in the parsing table

- $ is, by default, a nonterminal that takes the accepting state.
- 0,1,2,3,4,5,6 denotes I0,I1,I2,I3,I4,I5,I6
- I0 gives A in I2, so two are added to the A column and 0 rows.
- I0 gives S in I1, adding one to the S column and 1 row.
- similarly, five is written in A column and 2nd row, six is written in A column and three rows.
- I0 gives an in I3 to .so S3(shift 3) is added to a column and 0 rows.
- I0 gives b in I4, so S4(shift 4) is added to the b column and 0 rows.
- Similarly, S3(shift 3) is added on a column and 2,3 rows, S4(shift 4) is added on b column and 2,3 rows.

- I4 is reduced state as ' . ' is at the end. I4 is the **3**rd production of grammar. So write r**3**(reduce 3) in terminals.
- I5 is reduced state as ' . ' is at the end. I5 is the **1**st production of grammar. So write r**1**(reduce 1) in terminals.
- I6 is reduced state as ' . ' is at the end. I6 is the **2**nd production of grammar. So write r**2**(reduce 2) in terminals.

	ACTION			GOTO	
	a	b	$	A	S
0	S3	S4		2	1
1			accept		
2	S3	S4		5	
3	S3	S4		6	
4	R3	R3	R3		
5	R1	R1	R1		
6	R2	R2	R2		

Each cell has only one value, so the grammar is LR(0).

Construction of SLR

SLR Parser – SLR Parsing table

In the previous module, we discussed constructing the set of items. These canonical items are used to build the SLR parsing table. In this module, we will briefly discuss the construction of the SLR parsing table and the SLR parsing procedure.

SLR Parsing Table Construction

This algorithm does not produce a uniquely defined parsing action table for all grammar. However, it results pretty well in many grammars used for programming languages. The input to this table is the canonical set of items, which is given as the Deterministic finite automata, as already discussed in the previous module. The DFA is designed to recognize viable prefixes. The SLR parsing table is constructed between the set of canonical items against the grammar symbols. The non-terminals section of the parsing table corresponds to the goto() function, and the terminals section of the table corresponds to shift, reduce, accept, and error action. The steps involved in the SLR parsing table construction are detailed in Algorithm 15.1

Algorithm 15.1 SLR Parsing Table

- Input: Augmented Grammar G'
- Output: SLR parsing table with functions, shift, reduce, and accept SLR_Parsing_Table(Augmented grammar G')

{

1. Construct the set $C=\{I0,I1,\ldots,In\}$ of *LR(0) items*
2. State 'i' is constructed from Ii. The parsing action for state 'i' is determined as follows:

a. If $[A \to a \bullet ab]$ Î Ii and $goto(Ii,a)=Ij$ then set $action[i,a]$=shift *j, where a is a terminal*

b. If $[A \to a \bullet]$ Î Ii then set $action[i,a]$=reduce A®a for all a Î FOLLOW(A) where A^1S')

c. If $[S' \to S \bullet]$ is in Ii then set $action[i,\$]$=accept

3. If $goto(Ii,A)=Ij$ then set $goto[i,A]=j$
4. All entries other than 2 3 are declared as error

a. Repeat for all the items until no more entries are added

5. The initial state i is the Ii holding item $[S' \to \bullet S]$

}

The rows of the parsing table correspond to the item number, which is referred to as states. The columns correspond to the non-terminals and terminals. The non-terminals correspond to the goto() function. The table entry between the state and the terminals will reflect one of the four actions: shift, reduce, accept, or error. On the other hand, the table entry between the state entry and the non-terminals corresponds to the goto section.

From algorithm 15.1, step 2 of the algorithm handles the actions. In 2a, it considers the canonical collection of items, and we perform a goto(I, X) computation where X is a terminal. It considers the target item number from which the goto() is initiated and the destination item number formed. Sets it as a shift action at the table entry [i, a] if goto(Ii, a)=Ij. The step 2b of the algorithm is for reduce action, where we consider the kernel canonical items that have a dot at the end, and the table entry [i, a] is set as reduce by the production A®a, where 'a' corresponds to the FOLLOW(A). Step 2c of the algorithm is straightforward: it considers the item with the augmented grammar's kernel item and marks the table entry [kernel item number, $] as accepted. The remaining entries of the action section of the parsing table are set to error.

Step 3 of the algorithm generates table entries between states and the non-terminals. The table entry [i, X] = j, if goto(Ii, X)=Ij, where X is a non-terminal. The remaining entries of the parsing table are considered errors.

Example 15.1 Let us construct the parsing table for the expression grammar. The grammar is given here for reference. Table 15.1 also provides the canonical collection of items for quick reference.

- E' àE
- E à E +T
- E à T
- T à T * F
- T à F
- F à (E)
- F à id

Since E is the start symbol of the original grammar, E gets $ in the FOLLOW() set. Using the production F à (E), E gets the ')' symbol. Using the production, E à E+T, E gets "+." Using the production E à T, FOLLOW(T) = FOLLOW(E). In addition to the FOLLOW(E), FOLLOW(T) gets "*" from the production T à T*F. Tà F yields FOLLOW(F) = FOLLOW(T), and thus, the FOLLOW(T) and FOLLOW(F) results in the same set. The computation of FOLLOW() is given below in equations 15.1 to 15.3

Goto(I3, ( ) = Goto(I6, ( ) = Goto(I10, ( ) =I3 => [3, ( ] = [6, (] = [10, ( ] = s3 Goto(I7,)) = I9 => [7,)] = s9

Similarly, for the goto () section of the parsing table, consider Goto(I0, E) = 1, and so at the table entry [0, E], we set it to 1. The other entries are given below

Goto(I0, T) = Goto(I3, T) = 2 => [0,T] = [3, T] = 2

Goto(I0, F) = Goto(I3, F) = Goto(I6, F) = 4, => [0,F] = [3, F] = [6, F] = 4

Goto(I3, E) = 7 => [3, E] = 7

Goto(I6, T) = 8 => [6,T] = 8

Goto(I10, F) = 11 => [10, F] =11

After looking at the shift and goto actions, it is now convenient to set the accept action. The accepted action is set based on the first production of augmented grammar. So, at the item number where [S'à S.] is available, we put this item number against $ as accepted.

The expression grammar [E' à E.] is available in Item 1. So, at entry [1, $], we mark "acc" to indicate acceptance.

Now, onto the reduced actions. The item numbers that have the kernel item qualify for this action. Table 15.3 summarizes the item numbers that have this kernel item as one of its components. These items may or

may not have other items and are not considered. As seen from Table 15.3, there are six productions in the grammar, and thus, there are six kernel items. The reduced action procedure is explained in Table 15.3

Using Table 15.3, we construct Table 15.2, which incorporates the reduced action. However, this parser is not very powerful.

After constructing the table, the SLR parsing algorithm uses this table and a stack to validate a string. The steps involved in the parsing algorithm are given below:

- If *action*[*sm*, *ai*] = shift *s*, then push *ai*, push *s*, and advance input:

(*s*0 *X*1 *s*1 *X*2 *s*2 ... *Xm sm ai s*, *ai*+1 … *an* **$**)

- If *action*[*sm* ,*ai*] = reduce A ® b and *goto*[*sm -r*,*A*] = *s* with *r*=|b| then pop 2*r* symbols, push *A*, and push *s*:

(*s*0 *X*1 *s*1 *X*2 *s*2 ... *Xm -r sm-r A s*, *ai ai*+1 … *an* **$**)

The stack configuration contains alternate state numbers corresponding to item numbers and grammar symbols, with the top of the stack being a state number. The input includes a terminal. The top of the stack state is compared with the input, and we take a push action if the action is shifted, which would pop if the action is reduced. The next module details the algorithm with more explanations and examples.

Canonical LR

CLR Parser (with Examples)

LR parsers :

LR (k) parsing is an efficient bottom-up syntax analysis technique that can be used to parse large classes of context-free grammar.

L stands for left-to-proper scanning.

R stands for rightmost derivation in reverse.

k stands for no. of input symbols of lookahead

Advantages of LR parsing :

- It recognizes virtually all programming language constructs for which CFG can be written
- It can detect syntactic errors
- It is an efficient, non-backtracking shift, shift-reducing parsing method.

Types of LR parsing methods :

1. SLR
2. CLR
3. LALR

CLR Parser :

The CLR parser stands for canonical LR parser. It is a more powerful LR parser. It makes use of lookahead symbols. This method uses a large set of items called LR(1). The main difference between LR(0) and LR(1) items is that, in LR(1) items, it is possible to carry more information in a state, which will rule out useless reduction states. This extra information is incorporated into the state by the lookahead symbol. The general syntax becomes [A->∝.B, a]

Where A->∝.B is the production, and a is a terminal or right-end marker $

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

How do I add a lookahead with the production?

CASE 1 –

A->∝.BC, a

Suppose this is the 0th production. Since ' . ' precedes B, we must also write B's productions.

B->.D [1st production]

Suppose this is B's production. The look ahead of this production is given as we look at previous productions, i.e., the 0th production. Whatever is after B, we find FIRST(of that value), which is the lookahead of 1st production. So, here in the 0th production, after B, C is there. assume FIRST(C)=d, then 1st production become

B->.D, d

CASE 2 –

Now, if the 0th production was like this,

A->∝.B, a

Here, we can see there's nothing after B. So the lookahead of 0th production will be the lookahead of 1st production. ie-

B->.D, a

CASE 3 –

Assume a production A->a|b

A->a,$ [0th production]

A->b,$ [1st production]

Here, the first production is part of the previous production so that the lookahead will be the same as the last. These are the two rules of looking ahead.

Steps for constructing CLR parsing table :

1. Writing augmented grammar
2. LR(1) collection of items to be found
3. Defining 2 functions: goto[list of terminals] and action[list of non-terminals] in the CLR parsing table

EXAMPLE

Construct a CLR parsing table for the given context-free grammar

S-->AA

A-->aA|b

Solution :

STEP 1 – Find augmented grammar

The augmented grammar of the given grammar is:-

S'-->.S ,$ [0th production]

S-->.AA ,$ [1st production]

A-->.aA ,a|b [2nd production]

A-->.b ,a|b [3rd production]

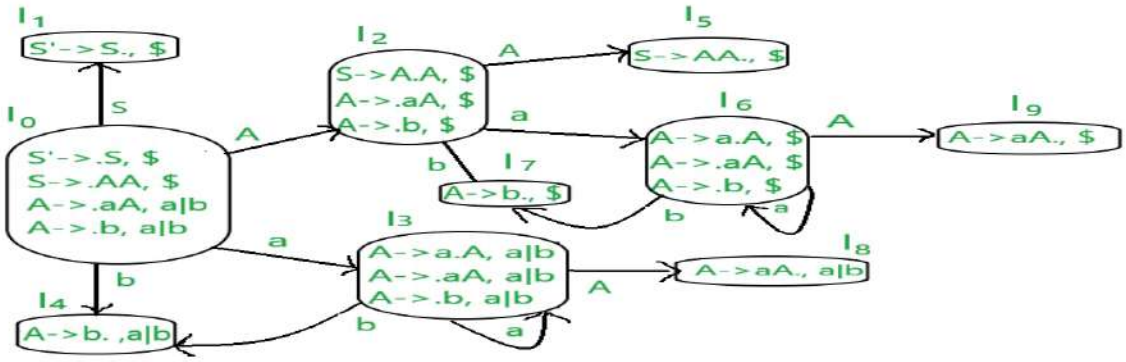
Let's apply the rule of lookahead to the above productions

- The initial look ahead is always $
- Now, the 1st production came into existence because of ' . ' Before 'S' in the 0th production. There is nothing after 'S,' so the lookahead of the 0th production will be the lookahead of the 1st production. I.e., S–>.AA,$

- Now, the 2nd production came into existence because of ' . ' Before 'A' in the 1st production. After 'A', there's 'A'. So, FIRST(A) is a,b Therefore, the look ahead for the 2nd production becomes a|b.
- Now, the 3rd production is a part of the 2nd production.So, the look-ahead will be the same.

STEP 2 – Find LR(1) collection of items

Below is the figure showing the LR(1) collection of items. We will understand everything one by one.



The terminals of this grammar are {a,b}

The non-terminals of this grammar are {S, A}

RULE-

1. If any non-terminal has ' . ' preceding it, we must write all its production and add. '' preceding each output.
2. from each state to the next state, the ' . ' shifts to one place to the right.
3. All the rules of lookahead apply here.

- In the figure, I0 consists of augmented grammar.
- Io goes to I1 when ' . ' of 0th production is shifted towards the right of S(S'->S.). This state is the accepted state. The compiler sees s. Since I1 is a part of the 0th production, the lookahead is the same, i.e. $
- Io goes to I2 when ' . ' of 1st production is shifted towards right (S->A.A) . The compiler sees a. Since I2 is a part of the 1st production, the lookahead is the same, i.e., $.
- I0 goes to I3 when ' . ' of the 2nd production is shifted towards the right (A->a.A) . The compiler sees a. Since I3 is a part of the 2nd production, the lookahead is the same, i.e., a|b.
- I0 goes to I4 when ' . ' of the 3rd production is shifted towards right (A->b.) . the compiler sees b. Since I4 is a part of the 3rd production, the lookahead is the same, i.e., a | b.
- I2 goes to I5 when ' . ' of 1st production is shifted towards right (S->AA.) . The compiler sees a. Since I5 is a part of the 1st production, the lookahead is the same, i.e., $.
- I2 goes to I6 when ' . ' of 2nd production is shifted towards the right (A->a.A) . The compiler sees a. Since I6 is a part of the 2nd production, the lookahead is the same, i.e. $.
- I2 goes to I7 when ' . ' of 3rd production is shifted towards right (A->b.) . The compiler sees a. Since I6 is a part of the 3rd production, the lookahead is the same, i.e., $.
- I3 goes to I3 when ' . ' of the 2nd production is shifted towards the right (A->a.A) . the compiler sees a. Since I3 is a part of the 2nd production, the lookahead is the same, i.e. a|b.
- I3 goes to I8 when ' . ' of 2nd production is shifted towards the right (A->aA.) . The compiler sees a. Since I8 is a part of the 2nd production, the lookahead is the same, i.e. a|b.
- I6 goes to I9 when ' . ' of 2nd production is shifted towards the right (A->aA.) . The compiler sees a. Since I9 is a part of the 2nd production, the lookahead is the same, i.e., $.
- I6 goes to I6 when ' . ' of the 2nd production is shifted towards the right (A->a.A) . The compiler sees a. Since I6 is part of the 2nd production, the lookahead is the same, i.e., $.

- I6 goes to I7 when ' . ' of the 3rd production is shifted towards right (A->b.) . the compiler sees b. Since I6 is a part of the 3rd production, the lookahead is the same, i.e., $.

STEP 3- Define two functions: goto[list of terminals] and action[list of non-terminals] in the parsing table. Below is the CLR parsing table

	ACTION			GOTO	
	a	b	$	A	S
0	S3	S4		2	1
1			accept		
2	S6	S7		5	
3	S3	S4		8	
4	R3	R3			
5			R1		
6	S6	S7		9	
7			R3		
8	R2	R2			
9			R2		

- $ is, by default, a nonterminal that takes the accepting state.
- 0,1,2,3,4,5,6,7,8,9 denotes I0,I1,I2,I3,I4,I5,I6,I7,I8,I9
- I0 gives A in I2, adding two to the A column and 0 row.
- I0 gives S in I1,so 1 is added to the S column and 1st row.
- Similarly, five are written in the A column and second row, 8 are in the A column and third row, and nine are in the A column and sixth row.
- I0 gives a in I3, so S3(shift 3) is added to a column and 0 row.
- I0 gives b in I4, so S4(shift 4) is added to the b column and 0 row.
- Similarly, S6(shift 6) is added on 'a' column and 2,6 row ,S7(shift 7) is added on b column and 2,6 row,S3(shift 3) is added on 'a' column and 3 row ,S4(shift 4) is added on b column and 3 row.
- I4 is reduced as ' . ' is at the end. I4 is the 3rd production of grammar. So write r3(reduce 3) in lookahead columns. The lookahead of I4 is a and b, so write R3 in the a and b columns.
- I5 is reduced as ' . ' is at the end. I5 is the first grammar production, so write r1(reduce 1) in the lookahead columns. The lookahead of I5 is $, so write R1 in the $ column.
- Similarly, write R2 in a,b column and 8th row, write R2 in $ column and 9th row.

LALR parsing tables.

LALR Parser (with Examples)

LALR Parser :

LALR Parser is a look-ahead LR parser. It is the most potent parser that can handle large classes of grammar. The size of the CLR parsing table is quite large compared to other parsing tables. LALR reduces the size of this table.LALR works similar to CLR. The only difference is that it combines the similar states of the CLR parsing table into one single state.

The general syntax becomes [A->∝.B, a]

Where A->∝.B is the production, and a is a terminal or right-end marker $

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

How do I add a lookahead with the production?

CASE 1 –

A->∝.BC, a

Suppose this is the 0th production. Since ' . ' precedes B, we must also write B's productions.

B->.D [1st production]

Suppose this is B's production. The look ahead of this production is given as we look at the previous production, i.e., the 0th production. Whatever is after B, we find FIRST(of that value), which is the lookahead of the first production. So, here in the 0th production, after B, C is there. Assume FIRST(C)=d, then the first production becomes.

B->.D, d

CASE 2 –

Now, if the 0th production was like this,

A->∝.B, a

Here, we can see there's nothing after B. So the lookahead of 0th production will be the lookahead of 1st production. ie-

B->.D, a

CASE 3 –

Assume a production A->a|b

A->a,$ [0th production]

A->b,$ [1st production]

Here, the first production is part of the previous production, so the lookahead will be the same as the last production.

Steps for constructing the LALR parsing table :

1. Writing augmented grammar
2. LR(1) collection of items to be found
3. Defining 2 functions: goto[list of terminals] and action[list of non-terminals] in the LALR parsing table

EXAMPLE

Construct CLR parsing table for the given context-free grammar

S-->AA

A-->aA|b

Solution:

STEP1- Find augmented grammar

The augmented grammar of the given grammar is:-

S'-->.S ,$ [0th production]

S-->.AA ,$ [1st production]

A-->.aA ,a|b [2nd production]

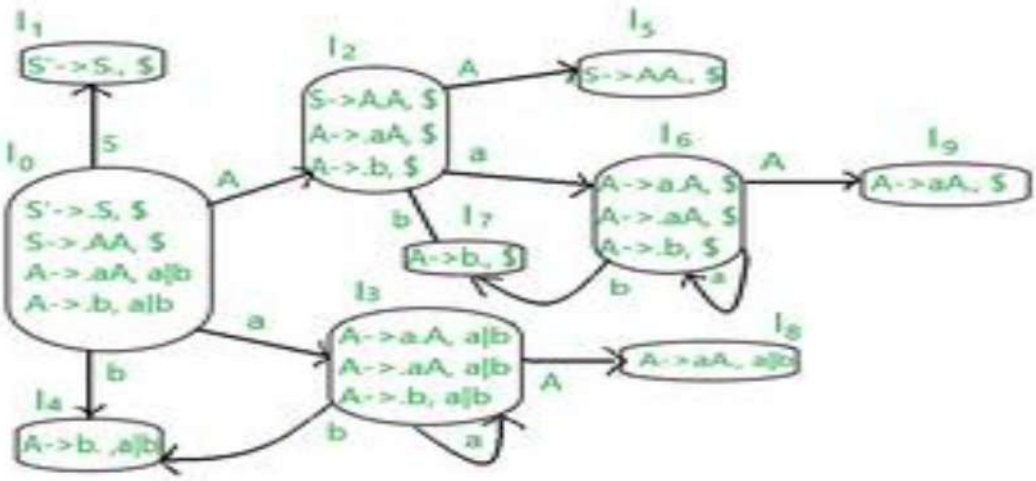
A-->.b ,a|b [3rd production]

Let's apply the lookahead rule to the above productions.

- The initial look ahead is always $
- Now, the 1st production came into existence because of ' . ' before 'S' in the 0th production. There is nothing after 'S,' so the lookahead of the 0th production will be the lookahead of the 1st production. i.e., S–>.AA,$
- Now, the 2nd production came into existence because of ' . ' before 'A' in the 1st production. After 'A', there's 'A'. So, FIRST(A) is a,b. Therefore, the lookahead of the 2nd production becomes a|b.
- Now,the 3rd production is a part of the 2nd production.So, the look-ahead will be the same.

STEP2 – Find LR(0) collection of items

Below is the figure showing the LR(0) collection of items. We will understand everything one by one.



The terminals of this grammar are {a,b}
The non-terminals of this grammar are {S, A}

RULES –

1. If any non-terminal has ' . ' preceding it, we must write all its production and add. '' preceding each output.
2. from each state to the next state, the ' . ' shifts to one place to the right.

- In the figure, I0 consists of augmented grammar.
- Io goes to I1 when ' . ' of 0th production is shifted towards the right of S(S'->S.). This state is the accepted state. The compiler sees s. Since I1 is a part of the 0th production, the lookahead is the same, i.e. $
- Io goes to I2 when ' . ' of 1st production is shifted towards right (S->A.A) . The compiler sees a. Since I2 is a part of the 1st production, the lookahead is the same, i.e., $.
- I0 goes to I3 when ' . ' of the second production is shifted towards the right (A->a.A). The compiler sees a. Since I3 is part of the second production, the lookahead is the same, i.e., a|b.
- I0 goes to I4 when ' . ' of 3rd production is shifted towards right (A->b.) . the compiler sees b. Since I4 is part of the 3rd production, the lookahead is the same, i.e., a|b.
- I2 goes to I5 when ' . ' of 1st production is shifted towards right (S->AA.) . The compiler sees a. Since I5 is a part of the 1st production, the lookahead is the same, i.e., $.
- I2 goes to I6 when ' . ' of 2nd production is shifted towards the right (A->a.A) . The compiler sees a. Since I6 is a part of the 2nd production, the lookahead is same i.e. $.
- I2 goes to I7 when ' . ' of 3rd production is shifted towards right (A->b.) . The compiler sees a. Since I6 is a part of the 3rd production, the lookahead is same i.e. $.
- I3 goes to I3 when ' . ' of the 2nd production is shifted towards the right (A->a.A) . the compiler sees a. Since I3 is a part of the 2nd production, the lookahead is the same, i.e. a|b.
- I3 goes to I8 when ' . ' of 2nd production is shifted towards the right (A->aA.) . The compiler sees a. Since I8 is a part of the 2nd production, the lookahead is same i.e. a|b.
- I6 goes to I9 when ' . ' of 2nd production is shifted towards the right (A->aA.) . The compiler sees a. Since I9 is a part of the 2nd production, the lookahead is same i.e. $.
- I6 goes to I6 when ' . ' of the 2nd production is shifted towards the right (A->a.A) . The compiler sees a. Since I6 is part of the 2nd production, the lookahead is the same, i.e., $.
- I6 goes to I7 when ' . ' of the 3rd production is shifted towards right (A->b.) . the compiler sees b. Since I6 is a part of the 3rd production, the lookahead is same i.e. $.

STEP 3 –

The parsing table defines two functions, goto [list of terminals] and action[list of non-terminals]. Below is the CLR parsing table

	ACTION			GOTO	
	a	b	$	A	S
0	S3	S4		2	1
1			accept		
2	S6	S7		5	
3	S3	S4		8	
4	R3	R3			
5			R1		
6	S6	S7		9	
7			R3		
8	R2	R2			
9			R2		

Once we make a CLR parsing table, we can easily make a LALR parsing table from it.

In the step diagram, we can see that

- I3 and I6 are similar except for their lookaheads.
- I4 and I7 are similar except their lookaheads.
- I8 and I9 are similar except their lookaheads.

In LALR parsing table construction, we merge these similar states.

- Wherever there are 3 or 6, make it 36(combined form)
- Wherever there are 4 or 7, make it 47(combined form)
- Wherever there is 8 or 9, make it 89(combined form)

Below is the LALR parsing table.

	ACTION			GOTO	
	a	b	$	A	S
0	S36	S47		2	1
1			accept		
2	S36	S47		5	
36	S36	S47		89	
47	R3	R3			
5			R1		
36	S36	S47		89	
47			R3		
89	R2	R2			
89			R2		

Now, we have to remove the unwanted rows

- As we can see, 36 rows have the same data twice, so we delete 1 row.
- We combine two 47 rows into one by combining each value in the single 47 rows.
- We combine two 89 rows into one by combining each value in the single 89 rows.

The final LALR table looks like the one below.

	ACTION			GOTO	
	a	b	$	A	S
0	S36	S47		2	1
1			accept		
2	S36	S47		5	
36	S36	S47		89	
47	R3	R3	R3		
5			R1		
89	R2	R2	R2		

---------- X ----------