Parsing the Input using parsing table –

Now it's the time to parse the actual string using above parsing table. Consider the parsing algorithm.

Input : The input string w that is to be parsed and parsing table.

Output : Parse w if $w \in L(G)$ using bottom-up. If $w \notin L(G)$ then report syntactical error.

Algorithm :

1. Initially push 0 as initial state onto the stack and place the input string with $ as end marker on the input tape.
2. If S is the state on the top of the stack and a is the symbol from input buffer pointed by a lookahead pointer then

a) If action[S, a] = shift j then push a, then push j onto the stack. Advance the input lookahead pointer.

b) If action[S, a] = reduce $A \rightarrow \beta$ then pop $2*|\beta|$ symbols. If i is on the top of the stack then push A, then push goto[i, A] on the top of the stack.

c) If action[S, a] = accept then halt the parsing process. It indicates the successful parsing.

Parsing the Input using LR(1) parsing table

String is aadd

Stack	Input buffer	Action table	Goto table	Parsing Action
$0	aadd$	action[0,a] = s3		shift
$0a3	add$	action[3,a] = s3		shift
$0a3a3	dd$	action[3,d] = s4		shift
$0a3a3d4	d$	action[4,d] = r4	[3,C] = 8	Reduce by C→d
$0a3a3C8	d$	action[8,d] = r3	[3,C] = 8	Reduce by C→aC
$0a3C8	d$	action[8,d] = r3	[0,C] = 2	Reduce by C→aC
$0C2	d$	action[2,d] = s7		Shift
$0C2d7	$	action[7,$] = r4	[2,C] = 5	Reduce by C→d
$0C2C5	$	action[5,$] = r2	[0,S] = 1	Reduce by S→CC
$0S1	$	action[1,$] = acc		Accept

LALR Parser.

- In this type of parser the lookahead symbol is generated for each set of item. The table obtained by this method are smaller in size than LR(K) parser. Most of the programming languages use LALR parsers.
- Steps of LALR Parser.

1) Construction of canonical set of items along with the lookahead
2) Building LALR Parsing table.
3) Parsing the input string using canonical LR Parsing table.

Construction of set of LR(1) items along with the lookahead

- The construction of LR(1) items is same as that for LR(1) parser. but the only difference is that: in construction of LR(1) items for LR parser, we have differed the two states if the second component is different but in this case we will merge the two states by merging of first and second components from both the states.
- For eg:-

I_3 : $C \to a \cdot C, a/d$
$C \to \cdot aC, a/d$
$C \to \cdot d, a/d$

I_6 : $C \to a \cdot C, \$$
$C \to \cdot aC, \$$
$C \to \cdot d, \$$

- For LALR parser we will consider these two states as same by merging these states i.e.

$I_3 + I_6 = I_{36}$

Hence I_{36} : goto (I_0, a)

$C \to a \cdot C, a/d/\$$
$C \to \cdot aC, a/d/\$$
$C \to \cdot d, a/d/\$$

Construction of LALR parsing table :-

Step 1 : Construct the LR(1) set of items.

Step 2 : Merge the two states I_i and I_j if the first component (ie the production rules with dots) are matching and create a new state replacing one of the older state such as $I_{ij} = I_i \cup I_j$

Step 3 : The parsing actions are based on each item I_i. The actions are given below.

(a) If $[A \to \alpha \cdot a\beta, b]$ is in I_i and goto $(I_i, a) = I_j$ then create an entry in the action table as
action $[i, a]$ = shift j.

(b) If there is a production $[A \to \alpha \cdot, a]$ in I_i, then in the action table action $[I_i, a]$ = reduce by $A \to \alpha$. Here A should not be S'

c) If there is a production $S' \to S \cdot, \$$ in I_i then set action $[i, \$]$ = accept.

Step 4 : The goto part of the LR table can be filled as : The goto transitions for state i is considered for non-terminals only.
If goto $(I_i, A) = I_j$ then set goto $(I_i, A) = j$

Step 5 : If the parsing action conflict then the algorithm fails to produce LALR parser and grammar is not LALR(1). All the entries not defined by rule 3 and 4 are considered to be "error".

Eg: $S \to CC$
$C \to aC$
$C \to d$

Construct the parsing table for LALR(1) parser.

Soln: First we will construct set of LR(1) items.

I_0: $S' \to \cdot S, \$$
$S \to \cdot CC, \$$
$C \to \cdot aC, a/d$
$C \to \cdot d, a/d$
— I_0

I_1: goto (I_0, S)
$S' \to S \cdot, \$$ — I_1

I_2: goto (I_0, C)
$S \to C \cdot C, \$$
$C \to \cdot aC, \$$
$C \to \cdot d, \$$
— I_2

I_3: goto (I_0, a)
$C \to a \cdot C, a/d$
$C \to \cdot aC, a/d$
$C \to \cdot d, a/d$
— I_3

I_4: goto (I_0, d)
$C \to d \cdot, a/d$ — I_4

I_5: goto (I_2, C)
$S \to CC \cdot, \$$ — I_5

I_6: goto (I_2, a)
$C \to a \cdot C, \$$
$C \to \cdot aC, \$$
$C \to \cdot d, \$$
— I_6

I_7: goto (I_2, d)
$C \to d \cdot, \$$ — I_7

I_8 : goto (I_3, C)
$C \to aC\cdot, a|d$ — I_8

I_9 : goto (I_6, C)
$C \to aC\cdot, \$$ — I_9

Now we will merge states 3, 6 then 4, 7 and 8, 9.

$S' \to \cdot S, \$$
$S \to \cdot CC, \$$
$C \to \cdot aC, a|d$
$C \to \cdot d, a|d$ } I_0

$S' \to S\cdot \$$ — I_1

$S \to C\cdot C, \$$
$C \to \cdot aC, \$$
$C \to \cdot d, \$$ } I_2

I_{36} : goto (I_0, a)
$C \to a\cdot C, a|d|\$$
$C \to \cdot aC, a|d|\$$
$C \to \cdot d, a|d|\$$ } I_{36}

I_{47} : goto (I_0, d)
$C \to d\cdot, a|d|\$$ — I_{47}

I_5 : $S \to CC\cdot, \$$ — I_5

I_{89} : goto (I_3, C)
$C \to aC\cdot\ a|d|\$$ — I_{89}

We have merged two states I_3 and I_6 and made the second component as a ord or $\$$.

LALR Parsing table

States	Action			Goto	
	a	d	$	S	C
0	S36	S47		1	2
1			Accept		
2	S36	S47			5
36	S36	S47			89
47	r3	r3	r3		
5			r1		
89	r2	r2	r2		

Parsing the Input string using LALR

String aadd

Stack	Input buffer	Action table	Goto table	Parsing Acti
$0	aadd$	action[0,a] = S36		shift
$0a36	add$	action[36,a] = S36		shift
$0a36a36	dd$	action[36,d] = S47		shift
$0a36a36d47	d$	action[47,d] = r36	[36,C] = 89	Reduce by C→d
$0a36a36C89	d$	action[89,d] = r2	[36,C] = 89	Reduce by C→aC
$0a36C89	d$	action[89,d] = r2	[0,C] = 2	Reduce by C→aC
$0C2	d$	action[2,d] = S47		shift
$0C2d47	$	action[47,$] = r36	[2,C] = 5	Reduce by C→d
$0C2C5	$	action[5,$] = r1	[0,S] = 1	Reduce by S→CC
$0S1	$	action[1,$] = accept		Accept.

Eg

Show that the following grammar

$S \rightarrow Aa \mid bAc \mid Bc \mid bBa$

$A \rightarrow d$

$B \rightarrow d$

is LR(1) but not LALR(1)

Recursive Descent Parser:

- A parser that uses collection of recursive procedures for parsing the given input string is called Recursive Descent Parser (RD).
- In this type of parser the CFG is used to build the recursive routines.
- The R.H.S of the production rule is directly converted to a program.
- For each non-terminal a separate procedure is written and body of the procedure (code) is R.H.S of the corresponding NT.

Basic steps for construction of RD parser

- The RHS of the rule is directly converted into program code symbol by symbol.

1) If the input symbol is NT then a call to the procedure corresponding to the NT is made.
2) If the input symbol is terminal then it is matched with the lookahead from input. The lookahead pointer has to be advanced on matching of the input symbol.
3) If the production rule has many alternates then all these alternates has to be combined into a single body of procedure.
4) The parser should be activated by a procedure corresponding to the start symbol.

(Q8) Consider the grammar having start symbol E

$E \rightarrow num\ T$

$T \rightarrow * num\ T \mid \epsilon$

Pseudo code for recursive descent parser

```
procedure E
{ if lookahead = num then
    {
      match(num);    /* call to procedure T */
      T;
    }
  else
    error;
  if lookahead = $
    {
      declare success;   /* Return on success */
    }
  else
    error;
} /* end of procedure E */

procedure T
{ if lookahead = '*'
    { match('*');
      if lookahead = 'num'
        {
          match(num);
          T;
        }
      else
        error
    }                    /* outer if closed */
  else NULL              /* indicates ε Here the other alter
                            is combined into same procedure T*
} /* end of procedure T */
```

```
procedure match (token t)
{ if lookahead = t
  lookahead = next_token   /* lookahead ptr is advanced */
  else
    error
}                          /* end of procedure match */

procedure error
{ print ("Error!!!");
}                          /* end of procedure error */
```

- We can see from the above code that the procedures for the NT are written and the procedure body is simply mapping of RHS of the corresponding rule.
- Consider that the input string is (3*4) we will parse this string using above given procedures.

3	*	4	$	E → num T	The parser will be activated by calling procedure E. Since the first input character 3 is matching with num the procedure match (num) will be invoked and then the lookahead will point to next token. And a call to procedure T is given.

3	*	4	$	T → * num T	A match with '*' is found hence ~~again the procedure for match (num) is fulfilled~~ lookahead = next_token.

3	*	4	$	T → * num T	Now '4' is matching
		↑			

with num hence again the procedure for match (num) is fulfilled. Then procedure for T is invoked. And T is replaced by ε.

3	*	4	$	Declare success!	As lookahead pointer
			↑		

points to $ we quit by reporting success. Thus the input string can be parsed using recursive descent parser.

Disadvantages of RD parser :-

- Construction of RD parser is easy. However the programming language that we choose for RD parser should support recursion.
- The internal details are not accessible in this type of parser.
- For instance, in this parser we cannot access the current leftmost sentential form. Secondly we cannot access the stack containing recursive calls.
- Writing procedures for left recursive grammar ($A \rightarrow \alpha\beta_1 | \alpha\beta_2$) is crucial. And to make such procedure simpler first we have to left factor the grammar.
- We cannot write RD parsers for all types of CFG.