UNIT III & IV

Bottom up parsing

5.0 Introduction

Given a grammar and a sentence belonging to that grammar, if we have to show that the given sentence belongs to the given grammar, there are two methods.

1. Leftmost – Derivation
2. Rightmost – Derivation

In left most derivation we start from the start symbol of the grammar and by choosing the production judiciously we try to derive the given sentence. The parsing methods based on this are known as top-down methods and we have already discussed this in previous chapter.

In rightmost derivation we start from the given sentence and using various productions, we try to reach the start symbol. The parsing method based on this concept are known as bottom-up method and are generally used in compilers generated using various tools like LEX and YACC. The bottom up parsing methods is more general and efficient. They can find syntactic error as soon as they occur.

We will be studying the following methods of parsing in this chapter

1. LR(0) Parsing
2. SLR (1) Parsing
3. LALR (1) Parsing Methods

Where LR stands for Left to Right scan and Rightmost derivation in reverse. SLR Stands for simple LR and LALR for Look Ahead LR and 1 in brackets indicate the number of look aheads.

LR (0) stands for no look ahead and LR (1) for one look ahead symbol. In general there can be LR (K) parsers with 'K' lookahead symbols.

5.1 Bottom up parsing

Bottom-up parsers in general use explicit stack. They are also known as shift reduce parsers.

Example: Consider a grammar S → a S b | c and a sentence a a c b b. The parser put the sentence to be recognized in the input buffer appended with end of input symbol $ and bottom of stack has also $.

Step	Start	Input buffer	Action
1	$	a a c b b $	Shift
2	$ a	a a b b $	Shift
3	$ a a	c b b $	Shift
4	$ a a c	b b $	Reduce using S → c
5	$ a a S	b b $	Shift
6	$ a a S b	b $	Reduce using S → a S b
7	$ a S	b $	Shift
8	$ a S b	$	Reduce using S → a S b
9	$ S	$	Accept.

The parser consults a table indexed by two parameters to be discussed later. The parameters what is on the top of stack and input character pointed by input buffer pointer. Assume for the time being that the table tells the parser to do one of the following activities.

1.	Shift	–	Shift the symbol to stack
2.	Reduce	–	Pop some symbols in the stack and replace by a non-terminal
3.	Accept	–	Input is syntactically correct, therefore accept
4.	Error	–	Input is syntactically not correct.

The table formation will be discussed later. From the figure above, after 3 shifts the table tells the parser to reduce in step-4 that is pop c and replace by (i.e., push) 'S'. In the 5th Step again shift is executed. In 6th step the parser is told to pop 3 symbols and replace it by S, in 7th step again parser is told to shift. In 8th step the parser pops 3 symbols & replaces it by S i.e., reduce action takes place. When top of stack is start symbol i.e., S in this case and input buffer is empty i.e., input buffer is pointing to $ (this indicates there is no more input available). The parser is told to carryout accept action. The parser is able to recognize that aacbb is indeed a valid sentence of the given grammar. We shall see later on how shift reduce and accept actions are indicated.

Example:

Consider another example of parsing using the grammar

S → a A c B e
A → A b | b
B → d

Recognize abbcde

Start	Input buffer	Action
$	a b b c d e $	Shift
$ a	b b c d e $	Shift
$ a b	b c d e $	Reduce A → b
$ a A	b c d e $	Shift
$ a A b	c d e $	Reduce A → A b
$ a A	c d e $	Shift
$ a A c	d e $	Shift
$ a A c d	e $	Reduce B → d
$ a A c B	e $	Shift
$ a A c B e	$	Reduce S → A c B e
$ S	$	Accept

Rightmost Derivation

S ⇒ a A c B e
⇒ a A c d e replace B → d
⇒ a A b c d e replace A → A b
⇒ a b b c d e replace A → b

Reduction done in the shift reduce parses is exactly the reverse order of rightmost derivation replace statements.

5.1.1 LR (0) Items: An LR (0) Items, is any production with a dot on the right hand side of a production.

Example 1:

S → a S b has the following LR (0) Items

S → • a S b
S → a • S b
S → a S • b
S → a S b •

As we can see that there are 3 symbols on the right hand side of the production. Therefore there are four positions where in a dot can be put, in general if there are n characters on RHS of a production there will be n + 1 LR (0) Items from that production.

Example 2: Find the LR (0) Items

E → E + T

Solution: LR (0) Items are

E → • E + T
E → E • + T
E → E + • T
E → E + T •

5.1.2 Augmentation of grammar

A grammar must be augmented before an LR parser could be constructed. Augmentation is nothing but adding a new production, which is not already present in the grammar that derives the start symbol.

Example:

Given grammar S → a S b | c

Augmented grammar

$S^1 \rightarrow S$
S → a S b | c

Where S^1 is a new start symbol which derives S i.e., the start symbol of the grammar. Now S^1 becomes the new start symbol. This is required to uniquely identify the accept state.

There is a relation between LR (0) Items & Finite Automata. It is possible to construct a DFA to recognize all viable prefixes of a given grammar. Viable prefixes are right-sentential forms that do not contain any symbols to the right of the handle. Handle is any RHS of a production.

For Example:

S → a S b | c

Handles are aSb and c

Viable prefixes are
- a a c — because c is handle
- a a S b — because a c b is a handle
- a a a S b — because a c b is a handle

To start the construction of DFA we need to augment the grammar as follows

$S^1 \rightarrow S$
S → a S b | c

Then, list all the Items

$S^1 \rightarrow$ • S
S → • a S b
S → a • S b
S → a S • b
S → a S b •
S → • c
S → c •

There are seven LR (0) Items in the above grammar. These LR (0) Items form the state S of the DFA. We need to put $S^1 \rightarrow \bullet S$ always is the start state as follows. This state is called as state 0

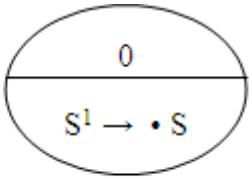

Fig 5.1 LR (0) of Augmented Production

Here we need to take closure of LR (0) Items. That is whenever a dot appears before a non-terminal we need to add to this set the first LR (0) Item derived by S i.e., $S \rightarrow \bullet a\ S\ b$ and $S \rightarrow \bullet c$. This is recursive definition. Now there are dots before only terminal a and c no closure is needed. Therefore state 0 of DFA has 3 Items.

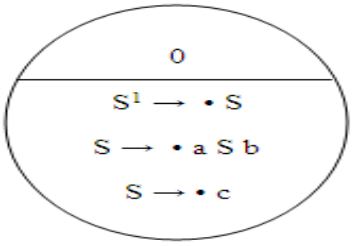

Fig 5.2 LR(0) of S^1 & S

Now we have to advance each of LR (0) Items in state 0. This is possible by assuming input is S, a and c (Note S cannot appear as input, only terminals can appear as input). The resulting states are

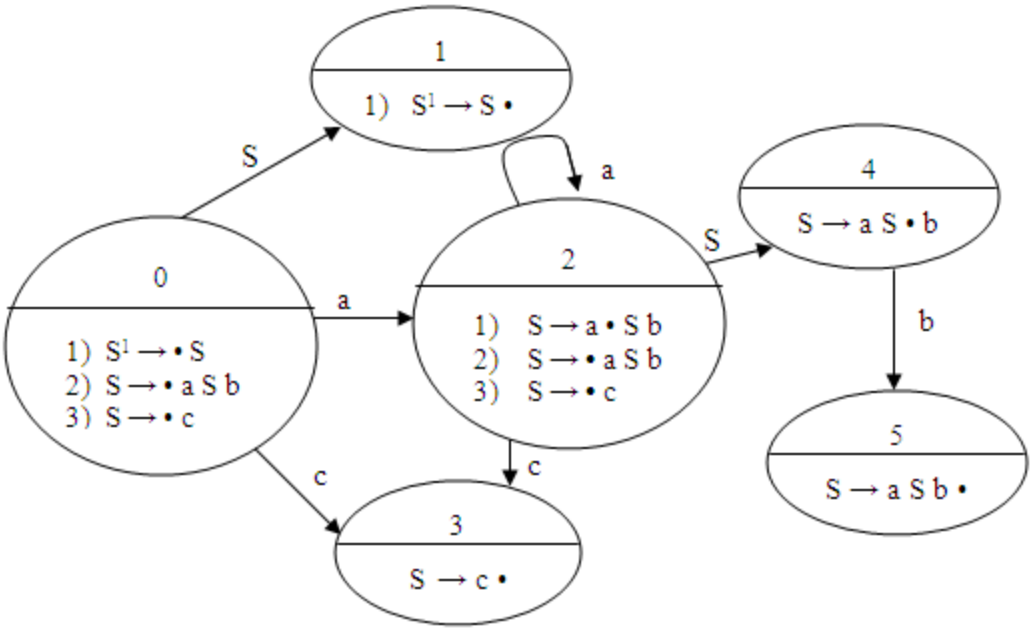

Fig 5.3 DFA of LR(0) Items

The resulting states are 1, 2 & 3. When dot moves to the end of RHS of a production and there are no other LR (0) Items, such states are called reduce states. Now we have 1, 3 & 5 are reduced states, because dot has moved to the rightmost position and no other LR (0) Items in those states. Whereas 0 and 2 state has three LR (0) Items, we say that, if 'a' occurs in state 0 then action to is shift and resulting state is 2. Similarly if 'c' occurs in state 0 then action is shift and goto state 3.

Consider state 2. Here there are three LR (0) Items, because of the closure of production – 1 now to progress from state of three are 3. Possibilities i.e., S, a and c. If 'c' occurs it goes to state 3 and 'a' occurs it go to itself as shown. When S occurs in state 2 it goes to state 4. By taking closure in state 4 no new Items are added. In state 4 when b comes it goes to state 5. Since the dot has reached the last position in state 5, no further states are generated.

We conclude that with 6 states (i.e., state 0 to 5) we can recognize all viable prefixes i.e.,

Viable Prefixes are

1) S
2) a a* c
3) a a* S
4) a a* S b
5) c

Since we need to recognize five viable prefixes we need to have 6 states.

5.1.3 Classification of states of DFA

States of the DFA are classified as following:

Accept State: The state which recognizes $S^1 \rightarrow S$. Note this is also a special reduce state

Reduce State: Where LR (0) Items have their dot placed on the extreme right position example 3 and 5

Shift State: Where all LR (0) Items have their do not in the extreme right position.

Shift/Reduce State: Mix of shift & reduce Items as defined above.

5.1.4 Constructing parsing table SLR (1)

The parsing table has rows indexed by state of DFA. Whereas columns are indexed by terminals (including $) and non-terminals (except S^1). The portion of table having terminal has index is known as ACTION Part and that indexed by non-terminals is known as GOTO portion as shown below.

States	ACTION				GOTO
	a	b	c	$	S
0	S2		S3		1
1				Acc	
2	S2		S3		4
3					
4					
5					

Table 5.4 Table for SLR (1) Parser

The filling of table is simple. In state 0 on 'a' it goes to state 2. Therefore it is a shift action, hence an entry S2. Similarly on 'c' it goes to state 3, again shift action, hence S3. On 'S' it goes to state '1' since 'S' is non-terminal it is filled only with state without any action.

Similarly all the other states are filed. Since $ is always follows any end of sentence, since we are putting ourselves to recognize the end of input and also when $S^1 \rightarrow S.$ comes. It is treated as accept state, as this state is unique.

Example: Consider the following grammar

$S^1 \rightarrow S$
$S \rightarrow (S)\ S$
$S \rightarrow \epsilon$

1) $S \rightarrow (S)\ S$
2) $S \rightarrow \epsilon$

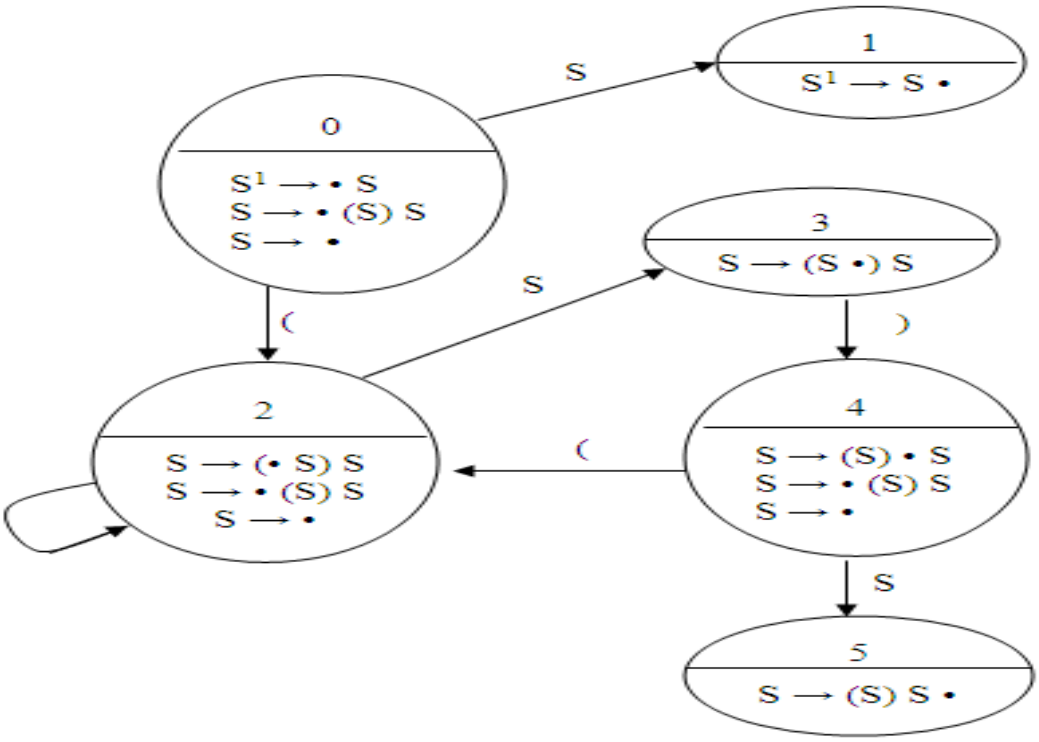

Fig 5.4 DFA for LR (0) Items

FOLLOW(S)={), $ }

State	Action			Goto
	(	)	$	S
0	S2	r2	r2	1
1			Acc	
2	S2	r2	r2	3
3		S4		
4	S2	r2	r2	5
5		r1	r1	

Table 5.5 SLR Parsing table

Parsing using the above table. Input ( )

	Input	Action
$ 0	( ) $	S2
$ 0 ~~C 2~~	) $	R2 : S → Є
$ 0 C 2 5 3	) $	S4
$ 0 (2 5 3) 4	$	r2 : S → Є
$ 0 ~~(2 5 3) 4 5 5~~	$	r1 : S → (S) S
$ 0 S 1	$	Accept
$ 0		

Example: Consider the following grammar.

$A \rightarrow (A) \mid a$,

Augment the grammar

0) $A^1 \rightarrow A$

1) $A \rightarrow (A)$

2) $A \rightarrow a$

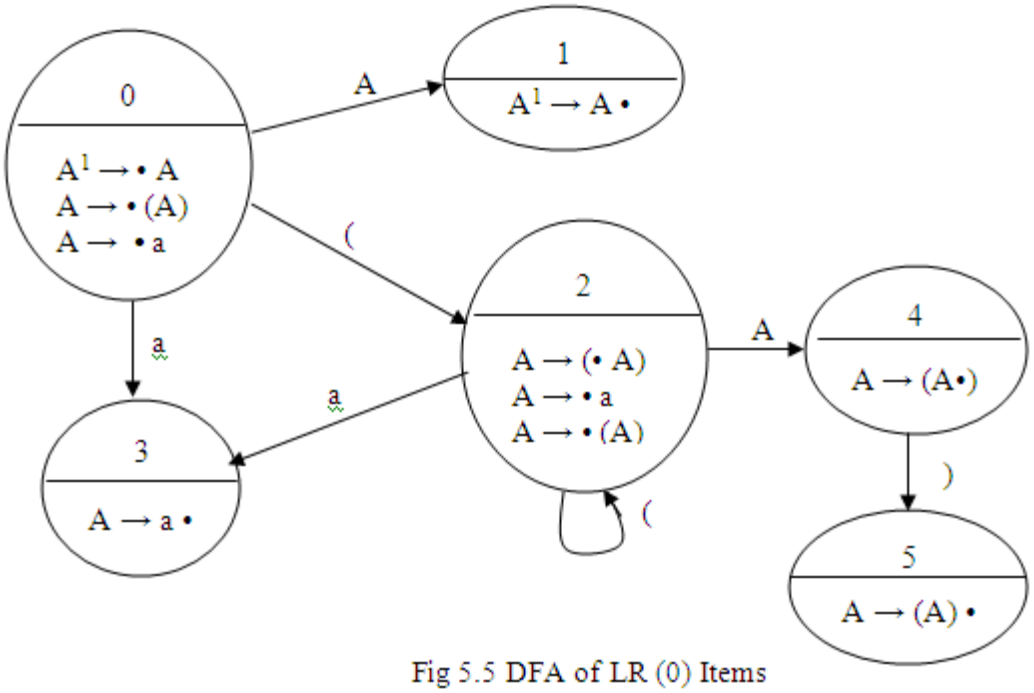

Fig 5.5 DFA of LR (0) Items

Follow (A) = {), $ }

State	Action				Goto
	a	(	)	S	A
0	S3	S2			1
1				acc	
2	S3	S2			4
3			r2	r2	
4			S5		
5			r1	r1	

Table 5.6 SLR Parsing table

Example: Consider the grammar S → (L) | a , L → L, S | S, Augment the grammar

S1 → • S
S → • (L) | a
L → • L, S | S

1) S → (L)
2) S → a
3) L → L, S
4) L → S

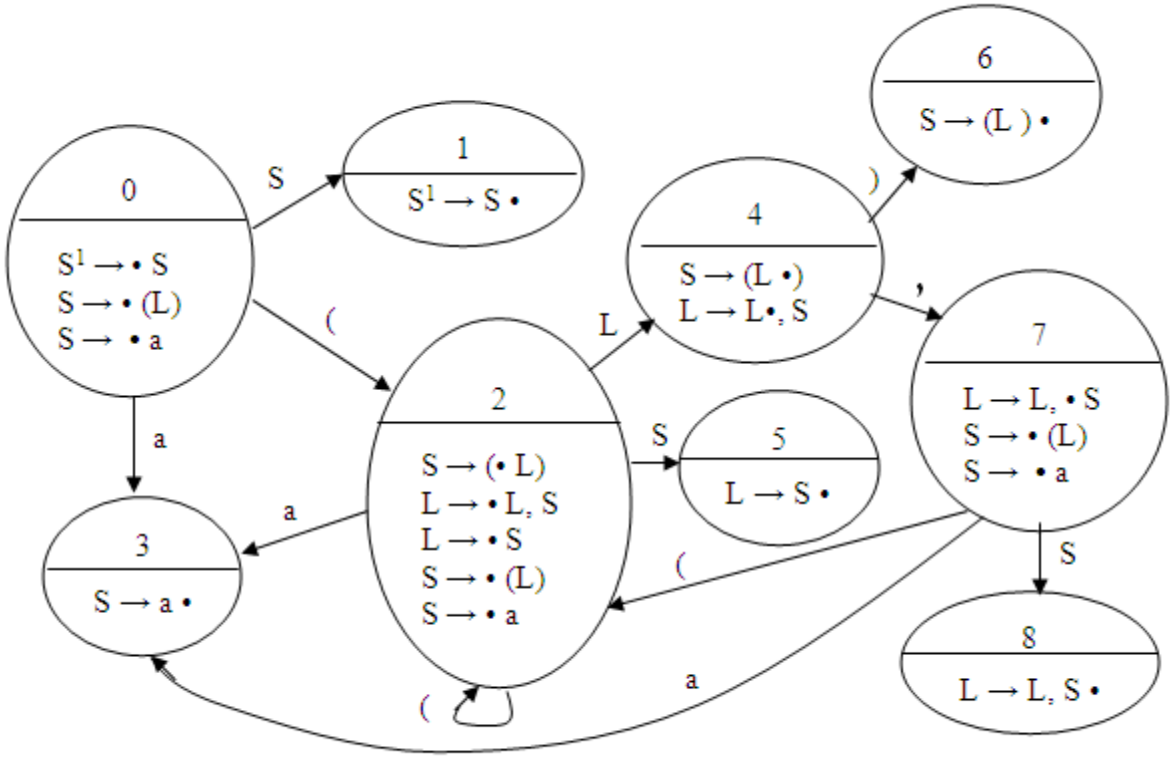

Fig 5.6 DFA of LR (0) Items

Follow (L) = {, ,) }

Follow (S) = {), $}

State	Action					Goto	
	a	(	)	,	$	S	L
0	S3	S2				1	
1					acc		
2	S3	S2				5	4
3			r2	r2	r3		
4			S6	S7			
5			r4	r4			
6			r1	r1	r1		
7	S3	S2				8	
8			r3	r3	r3		

Table 5.7 SLR Parsing table

We have seen the construction SLR (0) parser. This parser is able to parse most of the programming constructs.

5.1.5 Construction of LR(0) Parser:

LR (0) Parser is not a popular parser and it can be constructed following the procedure used for SLR (1) parser. The difference between LR(0) and SLR (1) parser is LR(0) parser actions, do not depend on the look ahead.

Consider the SLR (0) parser constructed for the simple grammar

$S \rightarrow a\ S\ b \mid c$

Procedure followed earlier is used to construct the DFA shown in fig 5.3. There are 6 states Action of each state depends whether it is a reduce state or shift state. In fig. 5.3 we see that there are

1) Reduce States 1, 3 and 5
2) Shift states 0, 2 & 4

Here the shift or reduce actions are done without looking at the input symbol. The table will have the following entries.

State	Action	Rule	Input			GOTO
			a	b	c	
0	Shift	- - - - - - -	2	-	3	1
1	Reduce	S1 → S	-	-	-	-
2	Shift	- - - - - - -	2	-	3	4
3	Reduce	S → c	-	-	-	-
4	Shift	- - - - - - -		5		
5	Reduce	S → a S b	-	-	-	-

Construction of the table is same as followed in SLR (1) parser and there is no need to calculate follow symbol for all the reduce states.

Now let use the above parser table to check the input a c b

Step	Parsing Stack	Input	Action
1	$ 0	a c b $	Shift
2	$ 0 a 2	c b $	Shift
3	$ 0 a 2 ~~c 3~~	b $	Reduce S → c
4	$ 0 a 2 S 5	b $	Shift
5	$ 0 ~~a 2 S 4 b 5~~	$	Reduce S → a S b
6	$ 0 S 1	$	Accept

To start with put starting state i.e., state 0 in parsing stack and input 'acb' appended by $ in input. Stack has stage '0', which is a shift state therefore shift a in to the stack. 0 state on a goes to state 2 as seen in the table. Therefore push state 2 on to the stack. State 2 is again shift state, therefore shift character c on to the stack. State 2 on character 'c' goes to state 3, therefore push state 3 on to the stack. This procedure continues i.e., state on the top of the stack decides the action to be gone without looking at the input symbol. State 1 is a reduce state but it is special because occurs only when we are ready to accept the input. As we see there is not much change from SLR (1) parsing.

5.2 Limitations of SLR (1)

SLR(1) as we have seen in simple, yet powerful to parse almost all languages constructs. But this fail in two cases.

1. Shift/Reduce conflict: When a state has both shift and reduce Items are present and parser indicates that both the operations to be done on a particular symbol. At this point parser will not be able to resolve and if fails.

2. Reduce/Reduce conflict: When a particular state has LR(1) Items which are both reduce Items which are both reduce Items i.e., the dot in LR (0) Items has reached rightmost position. And both are to be reduced on the same symbol.

Let us consider the parser to understand shift/reduce conflict.

$S^1 \rightarrow S$
$S \rightarrow L = R$
$S \rightarrow R$
$L \rightarrow A R$
$L \rightarrow i d$
$R \rightarrow L$

Consider the SLR(1) Parser

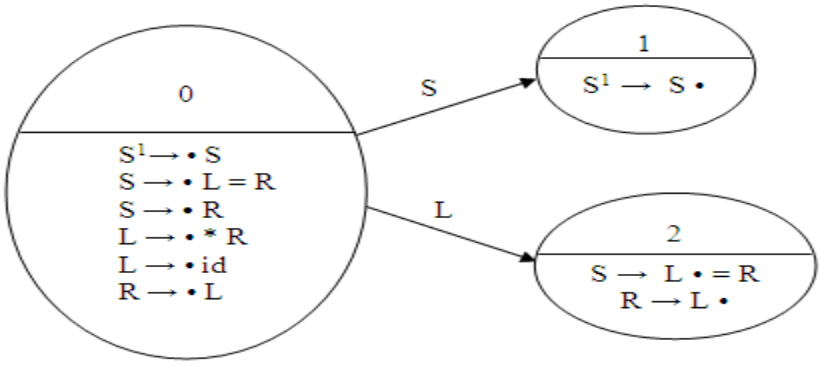

Fig 5.7 DFA for LR (0) Items

Let us analyze the state 2. There are two Items

1. S → L • = R
2. R → L •
3.

Item–1 is shift them and Item–2 is reduce Item of we calculate the follow symbols of R = { = }

There is a conflict on symbol =. We are unable to decide whether to use production R → L to reduce or shift S → L • = R to next state. This is known shift/reduce content. There is no solution and parser cannot proceed and SLR (1) fails.

5.3 Construction of LR (1) Parser:

LR (1) parsing or canonical parsing is resorted when SLR (1) fails Here we need to calculate LR (1) Items instead of LR (0) Items. i.e., the state of DFA will have LR (1) Items.

LR (1) Items have two parts

1. First part, same as LR(0) Item
2. Second part, look ahead or follow symbols that can occur at that instant.

Consider a grammar given below, to illustrate the calculation of LR (1) Items. once LR(1) Items are calculated the construction of DFA is simple and straightforward.

$S^1 \rightarrow S$
$S \rightarrow DD$
$D \rightarrow cD$
$D \rightarrow d$

For $S^1 \rightarrow \bullet S$ the follow symbols obviously is \$ as S^1 is a start symbol. Therefore LR(1) Item is $S^1 \rightarrow \bullet S,\$$

If we take closure of • S and replace S by • D D still the follow symbol remain \$. Therefore

S → • D D, $

Now take closure of D we replace D by • iD. Note that 1st D is replaced. All the First symbols of second D will become follow symbols.

D → • c D, c / d

D → • d, c / d

Note here we are calculating follow symbol in that context. If we had

S → D • D, $

When we take closure of D, it is the 2nd D . Therefore the Follow symbol will remain $. i.e.,

D → • c D, $ & D → • d, $

We complete the DFA as follows with LR (1) Items.

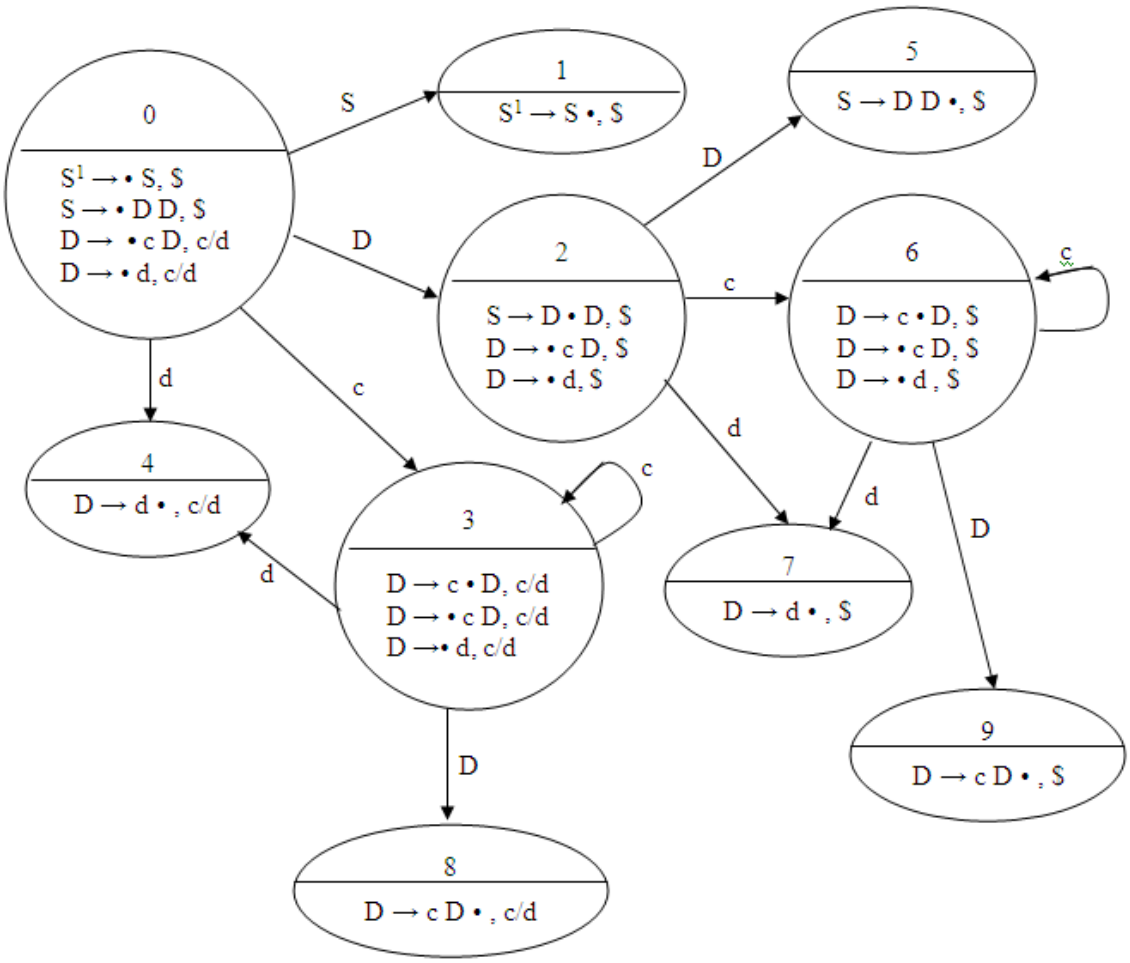

Fig 5.8 DFA of LR (1) Items

The parsing table is constructed in the same way as SLR (1) parser but there is no need to calculate follow symbols.

States	Action			Go To	
	c	d	$	S	D
0	S3	S4		1	2
1			Acc		
2	S6	S7			5
3	S3	S4			8
4	r3	r3			
5			r1		
6	S6	S7			9
7			r3		
8	r2	r2			
9			r2		

Table 5.8 LR (1) Parsing table

The r1, r2, r3 are as follows

r1: S→ D D

r2: D → c D

r3: D → d

Let us now parse a sentence derived from the above grammar. Let the given sentence be

Stack	Input buffer	Action
$ 0	c c d d $	S3
$ 0 c 3	c d d $	S3
$ 0 c 3 c 3	d d $	S4
$ 0 c 3 c 3 ~~d~~ 4	d $	r3: D → d
$ 0 c ~~3 D 8~~	d $	r2: D → c D
$ 0 ~~c 3 D 8~~	d $	r2: D → c D
$ 0 D 2	d $	S7
$ 0 D 2 ~~d 7~~	$	r3 : D → d
$ 0 ~~D 2 D 5~~	$	r1: S → D D
$ 0 S	$	Acc
$ 0		

5.3.1 Limitations of LR(1) Parser:

In the previous example (Fig 6.2) we see that state 8 and 9 have the same first component but they differ only in their look ahead symbols. Similarly states 4 & 7 and 3 & 6. LR (1) parser has more states than SLR (1) parser and for a typical programming language it may have several thousands of states. This increase the memory requirement.

5.4 Construction LALR(1) Parser:

This can easily be constructed from the table of LR (1) parser. Basic principle being the states which have the same first component (called common core) are combined, to form new states, which we may call them by their combined states. For example

States 3 & 6 are combined & a new state is called 36 state

Similarly we create 89 and 47 states.

The action and GOTO are modified and the new follows LALR (1) parsing table is as

States	Action			Go To	
	c	d	S	S	D
0	S36	S47		1	2
1			acc		
2	S36	S47			5
36	S36	S47			89
47	R3	R3	R3		
5			R1		
89	R2	R2	R2		

Table 5.9 LR (1) Parsing table

The procedure for parsing is same as in SLR (1) parser.

LALR (1) parsers are the most common and are modern parsers. The parser generator generally construct LALR (1) parsers. These parser do have states comparable to that of SLR (1) parser.

5.5 Parser Generator : YACC:

/*Yacc program for checking the validity of an arithmetic expression*/

```
%{
#include<stdio.h>
int flag;
%}
%left    '-' '+'
%left    '*' '/'
%left    '%'
%token DT, OP            /* DT → digits; OP → operands; these
                                    are defined in the lex program below*/
%%

E:S      {flag = 1; printf("\n valid expression \n");}
S:S '+' S                           /* S is being defined in this section;
 |S '-' S                          S could be defined as  S+S; or S-S;
 |S '*' S                          S*S; or S/S or (S) or digit or
 |S '/' S                          operand for checking the validity of
 |'(' S ')'                        the expression */
 |DT
 |OP
 ;
%%

int main()
 {
        char ch;
        flag = 0;                   /* initialization of flag */
        printf("\n enter expression : ");
        yyparse();

        return 0;
 }

int yyerror()
 {
        if(flag==0)
         printf("\n invalid expression\n");
        return 0;
 }
```

/*Supporting Lex program for checking the validity of an arithmetic expression*/

```
%{
#include "y.tab.h"
%}
%%
[0-9]+        {return DT;}
[a-zA-Z]+     {return OP;}
```

```
[\t]          ;
\n            {return 0;}
.             {return yytext[0];}
%%
```

Output:

1. enter expression : (A+B)
   valid expression

2. enter expression : a*b-c
   valid expression

3. enter expression : a/c+b-e
   valid expression

4. enter expression : 5+(8-4)
   valid expression

5. enter expression : 5(8-4)
   invalid expression

6. enter expression : (a*b-c
   invalid expression

5.6 Error Recovery in Bottom-up parser:

As discussed earlier error recovery is to give suitable error message and also give a correction so that parsing is successful.

Consider the grammar following grammar

$S \rightarrow a\ S\ b \mid c$

And the corresponding parsing table at 5.14. In this table blanks indicate error. Here we are to give connection to suitable error routine which gives a suitable error messages and does the appropriate correction. The table is modified as follows

States	Action				Go To
	a	b	c	$	S
0	S2	e1	S3		1
1				Acc	
2	S2	e2	S3		4
3		r2		r2	
4		S5			
5		r1		r1	

Table 5.10 LALR Parsing Table

e1: Error routing : Error – Message – rising a push a & cover is with state 2

e2: Error routine : Error Message – unbalanced b & remove input symbol.

Now let us see how parser will be able recover while parsing a syntactically wrong sentence and also give suitable messages.

Step	Stack	Input	Action
1	$ 0	b c b $	E1: Error! rising 'a' push a & cover it with state 2
2	$ 0 a 2	b c b $	E2: unbalanced b remove input b shift S3
3	$ 0 a 2	c b $	Shift S3
4	$ 0 a 2 ~~e 3~~	b $	Reduce S → c
5	$ 0 a 2 S 4	b $	Shift S 5
6	$ 0 ~~a 2 S 4 b 5~~	$	Reduce S → a S b
7	$ 0 S 1	$	Accept

Above recovery shows that with error routines e1 and e2 not only relevant error messages were given and parser was able recover from the error to accept the syntactically wrong input 'b c b'

In general error routine design requires careful consideration. The above error recovery is only to illustrate a typical case. However we have left other error places (blanks) without any action. By careful consideration all the error places we can suggest suitable recovery action.

Assignment

1. Construct LR (0) items for grammar

   E→(L)/a

   L→ L,E/E

2. Construct LR (0) items for grammar

   S →A a | b A c | B c | b B a

   A →d

   B →d

4. Consider the grammar Q no 1, construct the SLR parsing table for the same

5. Show that the grammar

$S \rightarrow A\ a\ Ab \mid B\ b\ B\ a$

$A \rightarrow Є$

$B \rightarrow Є$

is not SLR

6. Construct LR (0) items for grammar
7. Construct SLR parsing table for the grammar in Q. No.5
8. Show the parsing stack and the actions of an SLR parser for the input string ((a),a,(a,a)) considering the grammar of Q no 1
9. Construct DFA of LALR(1) items for the grammar in Q no 1
10. Construct LALR parsing table for the grammar in Q no 1
11. Show that the following grammar is not LR(1)

    $A \rightarrow aAa / \varepsilon$