CD: COMPILER DESIGN
CD: UNIT-2 09/2022

SEPTEMBER 2022 / IT-3rd Year, V Semester
FALL SEMESTER, YEAR (5th, 3rd)
FALL SESSION (2022-23)
(CD)
MS. SHWETA TIWARI
Published: SEPTEMBER, 2022

PREPARED FOR
Engineering Students
All Engineering College

CD: COMPILER DESIGN

INSTRUCTOR: Ms. SHWETA TIWARI
shwetatiwari08@recabn.ac.in
shwetatiwari08aug@gmail.com

Shweta Tiwari

September 2022

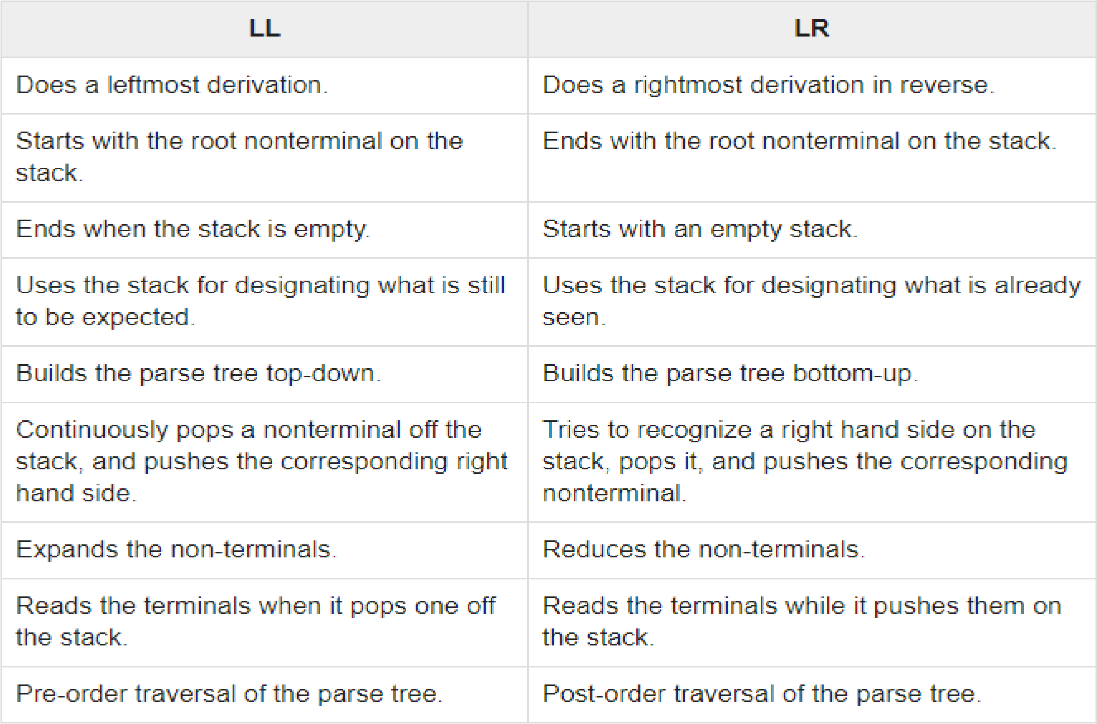
TOPIC On : LR() PARSER

LR() PARSERS

LR PARSERS:

An efficient bottom-up syntax analysis technique that can be used to parse a large class of CFG is called LR(k) parsing. The “L” is for left-to-right scanning of the input, the “R” for constructing a rightmost derivation in reverse, and the “k” for the number of input symbols of lookahead that are used in making parsing decisions.. When (k) is omitted, it is assumed to be 1. Table 2.2 shows the comparison between LL and LR parsers.

Table 1 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.
Ends when the stack is empty.	Starts with an empty stack.
Uses the stack for designating what is still to be expected.	Uses the stack for designating 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.
Reads the terminals when it pops one off the stack.	Reads the terminals while it pushes them on the stack.
Pre-order traversal of the parse tree.	Post-order traversal of the parse tree.

Types of LR parsing method:

1. SLR- Simple LR
    - Easiest to implement, least powerful.
2. CLR- Canonical LR
    - Most powerful, most expensive.
3. LALR- Look -Ahead LR
    - Intermediate in size and cost between the other two methods

The LR Parsing Algorithm:

The schematic form of an LR parser is shown in Fig 1. It consists of an input, an output, a stack, a driver program, and a parsing table that has two parts (*action* and *goto*).The driver program is the same for all LR parser. The parsing table alone changes from one parser to another. The parsing program reads characters from an input buffer one at a time. The program uses a stack to store a string of the form $s_0X_1s_1X_2s_2$...... X_ms_m , where s_m is on top. Each X_i is a grammar symbol and each s_i is a symbol called astate.

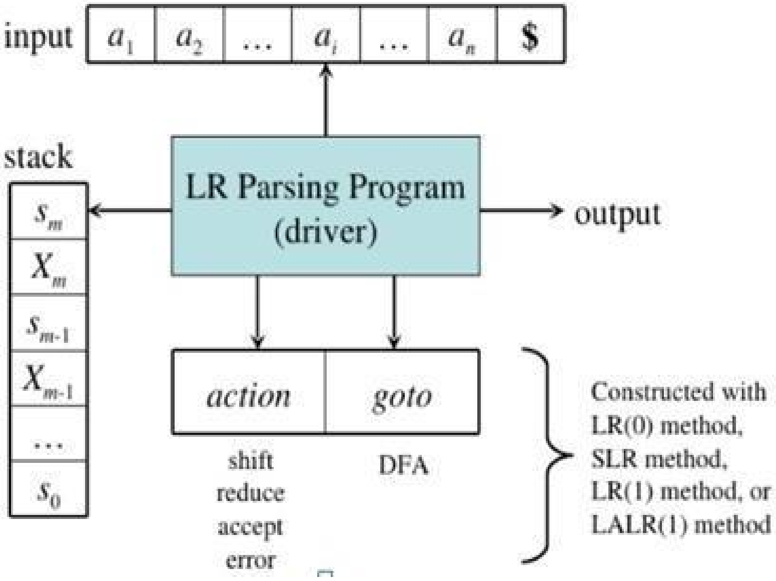


Fig 1 Model of an LR Parser The parsing table consists of two parts : *action* and *goto* functions.

Action : The parsing program determines sm, the state currently on top of stack, and ai, the current input symbol. It then consults *action*[s_m,ai] in the action table which can have one of four values :

1. shift s, where s is a state,
2. reduce by a grammar production A → β,
3. accept, and
4. error.

Goto : The function goto takes a state and grammar symbol as arguments and produces a state.

CONSTRUCTING SLR PARSING TABLE:

To perform SLR parsing, take grammar as input and do the following:

1. Find LR(0) items.
2. Completing the closure.
3. Compute *goto*(I,X), where, I is a set of items and X is a grammar symbol.

LR(0) items:

An *LR(0) item* of a grammar G is a production of G with a dot at some position on the right side. For example, production A → XYZ yields the four items :

A →•XYZ

A→ X•YZ

A→ XY•Z

A → XYZ•

Closure

operation:

If I is a set of items for a grammar G, then closure(I) is the set of items constructed from I by the two rules:

1. Initially, every item in I is added to closure(I).
2. If A → α . Bβ is in closure(I) and B → γ is a production, then add the item B → . γ to I , if it is not already there. We apply this rule until no more new items can be added to closure(I).

Goto operation:

Goto(I, X) is defined to be the closure of the set of all items [A→ αX•β] such that [A→ α•Xβ] is in I.Steps to construct SLR parsing table for grammar G are:

1. Augment G and produce G`
2. Construct the canonical collection of set of items C for G‟
3. Construct the parsing action function *action* and *goto* using the following algorithm that requires FOLLOW(A) for each non-terminal of grammar.

Algorithm for construction of SLR parsing table:

Input : An augmented grammar G‟

Output : The SLR parsing table functions *action* and *goto* for G'

Method :

1. Construct C ={I_0, I_1, …. I_n}, the collection of sets of LR(0) items for G'.
2. State *i* is constructed from I_i. The parsing functions for state *i* are determined as follows:

   (a) If [A→α•*a*β] is in I_i and goto(I_i,*a*) = I_j, then set *action*[*i*,*a*] to "shift j". Here *a* must be terminal.

   (b)If[A→α•] is in I_i , then set *action*[*i*,*a*] to "reduce A→α" for all *a* in FOLLOW(A).

   (c)If [S‟→S•] is in I_i, then set *action*[*i*,$] to "accept".

If any conflicting actions are generated by the above rules, we say grammar is not SLR(1). 3.The *goto* transitions for state *i* are constructed for all non-terminals A using the rule: If *goto*(I_i,A)= I_j, then *goto*[i,A] = *j*.

4. All entries not defined by rules (2) and (3) are made "error"
5. The initial state of the parser is the one constructed from the set of items containing [S'→•S].

SLR Parsing algorithm:

Input: An input string *w* and an LR parsing table with functions *action* and *goto* for grammar G.

Output: If *w* is in L(G), a bottom-up-parse for *w*; otherwise, an error indication.

Method: Initially, the parser has s0 on its stack, where s0 is the initial state, and *w*$ in the input buffer. The parser then executes the following program :

```
set ip to point to the first input symbol of w$;
repeat forever begin
        let s be the state on top of the stack and a the symbol pointed to by ip;
        if action[s, a] =shift s‟ then begin
                push a then s‟ on top of the stack;
```

```
                    advance ip to the next input symbol
               end
               else if action[s, a]=reduce A→β then begin
                    pop 2* |β |symbols off the stack;
                    let s‟ be the state now on top of the stack;
                    push A then goto[s‟, A] on top of the
                    stack; output the production A→ β
               end
               else if action[s, a]=accept
                 then return
               else error( )
          end
```

Example: Implement SLR Parser for the given

grammar: 1.E→E + T

2.E→T

3.T→T * F

4.T→F

5.F→(E)

6.F→**id**

Step 1 : Convert given grammar into augmented grammar.

Augmented grammar:

E'→E

E→E + T

E→T T→T

* F T→F

F→(E)

F→id

Step 2 : Find LR (0) items.

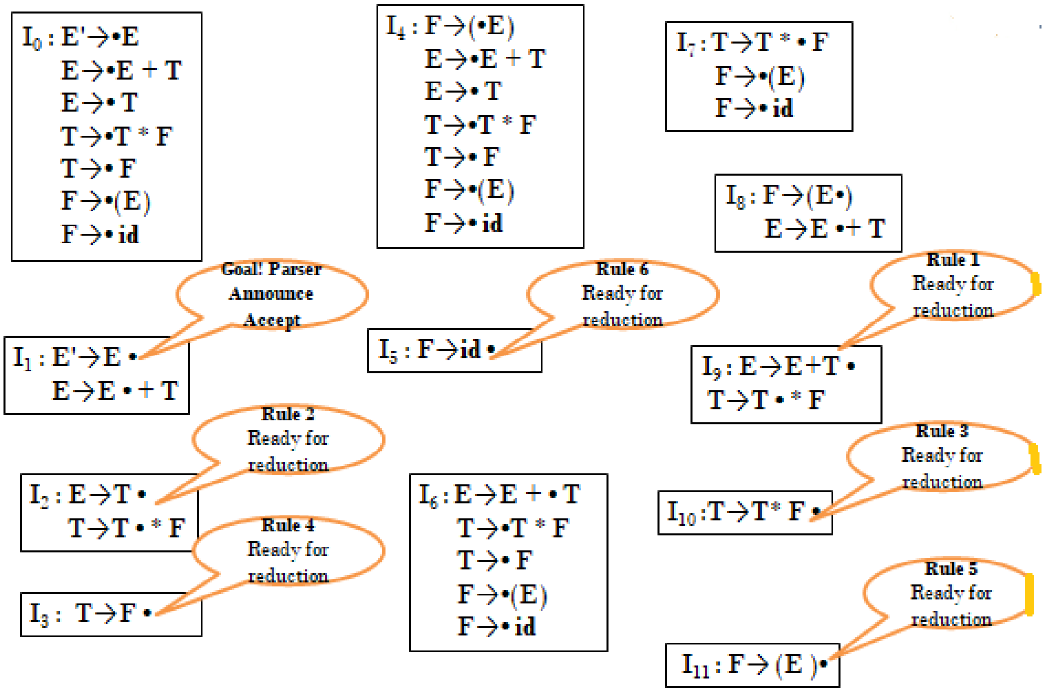


Fig 2 Canonical LR(0) collections

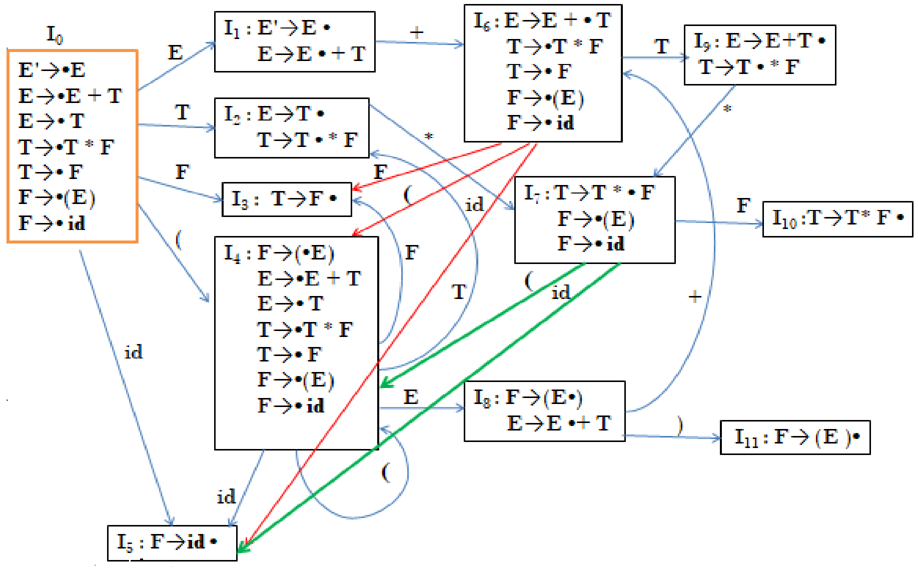


Fig 3 DFA representing the GOTO on symbols

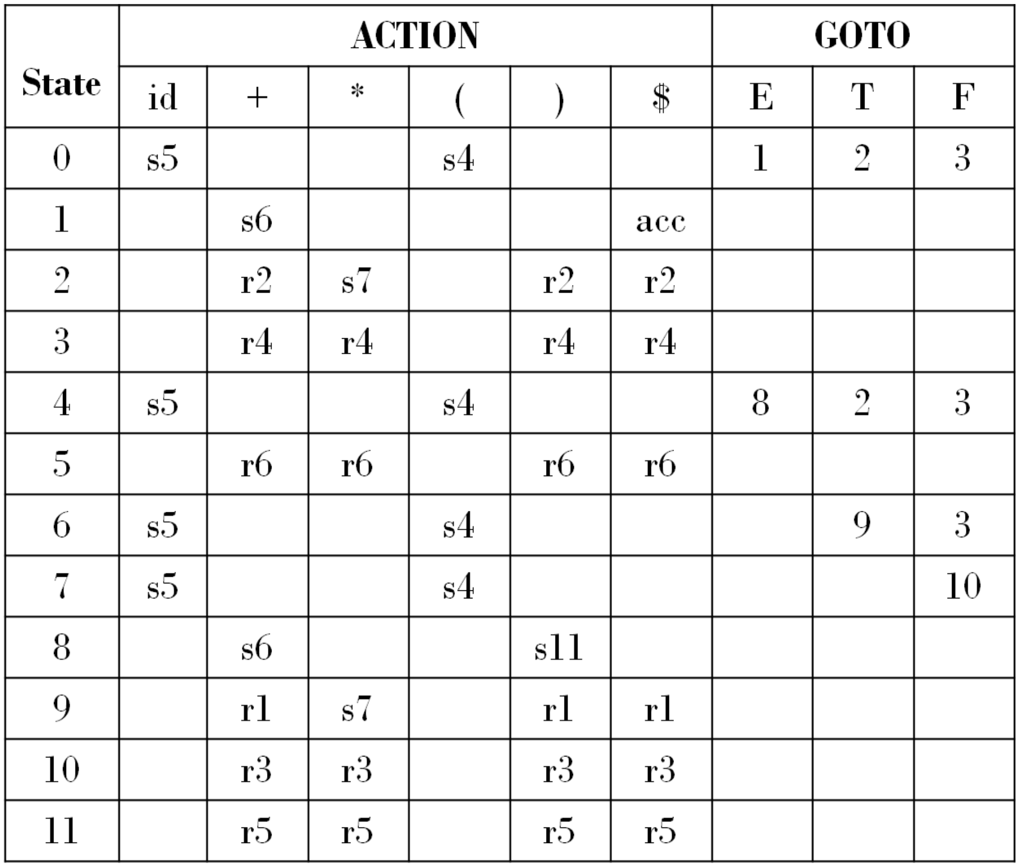
Step 3 : Construction of Parsing table.

1. Computation of FOLLOW is required to fill the reduction action in the ACTION part of the table.

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

FOLLOW(T) ={*,+,) ,$}

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



State	ACTION						GOTO		
	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

Fig 4 Parsing Table for the expression grammar

1. s_i means shift and stack state i.
2. r_j means reduce by production numbered j.
3. acc means accept.
4. Blank means error.

Step 4: Parse the given input. The Fig 5 shows the parsing the string id*id+id using stack implementation.

Stack	Input	Action
0	id*id+id$	s5 Shift 5
0 id 5	*id+id$	r6 Reduce by **F→id**
0 **F** 3	*id+id$	r4 Reduce by **T→F**
0 **T** 2	*id+id$	s7 Shift 7
0 **T** 2 * 7	id+id$	s5 Shift 5
0 **T** 2 * 7 id 5	+id$	r6 Reduce by **F→id**
0 **T** 2 * 7 **F** 10	+id$	r3 Reduce by **T→T*F**
0 **T** 2	+id$	r2 Reduce by **E→T**
0 **E** 1	+id$	s6 Shift 6
0 **E** 1+ 6	id$	s5 Shift 5
0 **E** 1+ 6 id 5	$	r6 Reduce by **F→id**
0 **E** 1+ 6 **F** 3	$	r4 Reduce by **T→F**
0 **E** 1+ 6 **T** 9	$	r1 Reduce by **E→E+T**
0 **E** 1	$	Accept

Fig 5 Moves of LR parser on **id*id+id**

***************************************THE END*****************************