Parul® University

COMPILER DESIGN
SUBJECT CODE: 203105351

Prof. Kapil Dev Raghuwanshi, Assistant Professor
Computer Science & Engineering

CHAPTER-3

Top-down parsing

What is Parsing

Parsing: is that phase of compiler which takes token string as input and with the help of existing grammar, converts it into the corresponding parse tree.

-> Parser is also known as Syntax Analyzer

- Parsing technique is divided in two Types.

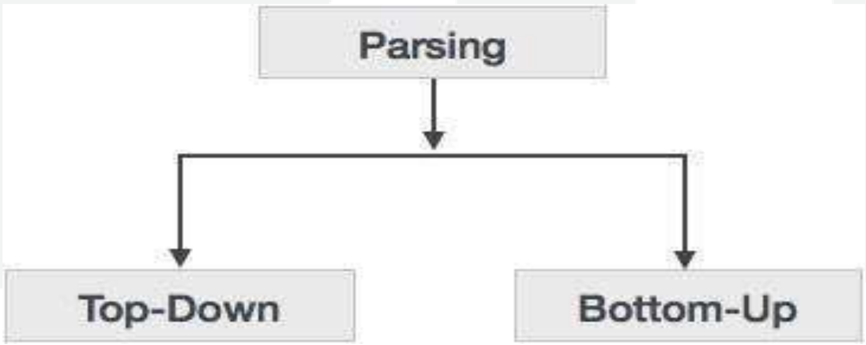
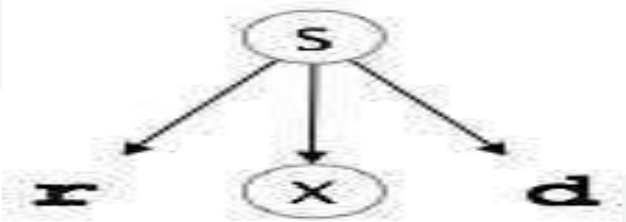


Image source : Google

Top - Down Parsing:

Top-down parser: is the parser which generates parse for the given input string with the help of grammar productions by expanding the non-terminals i.e. it starts from the start symbol and ends on the terminals.

- It uses left most derivation



Bottom-Up/Shift Reduce parser : Bottom-up Parser is the parser which generates the parse tree for the given input string with the help of grammar productions by compressing the non-terminals i.e. it starts from non-terminals and ends on the stat symbol

- It uses reverse of the right most derivation.

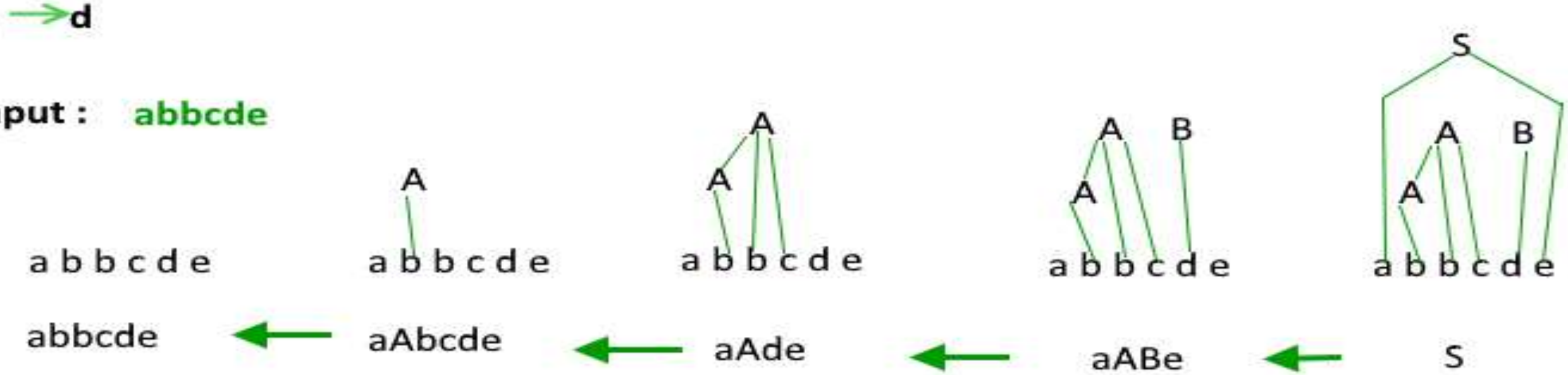
Bottom-Up parser

S → aABe

A → Abc/b

B → d

Input : abbcde



Top-Down and Bottom-Up parser are further divided in following types

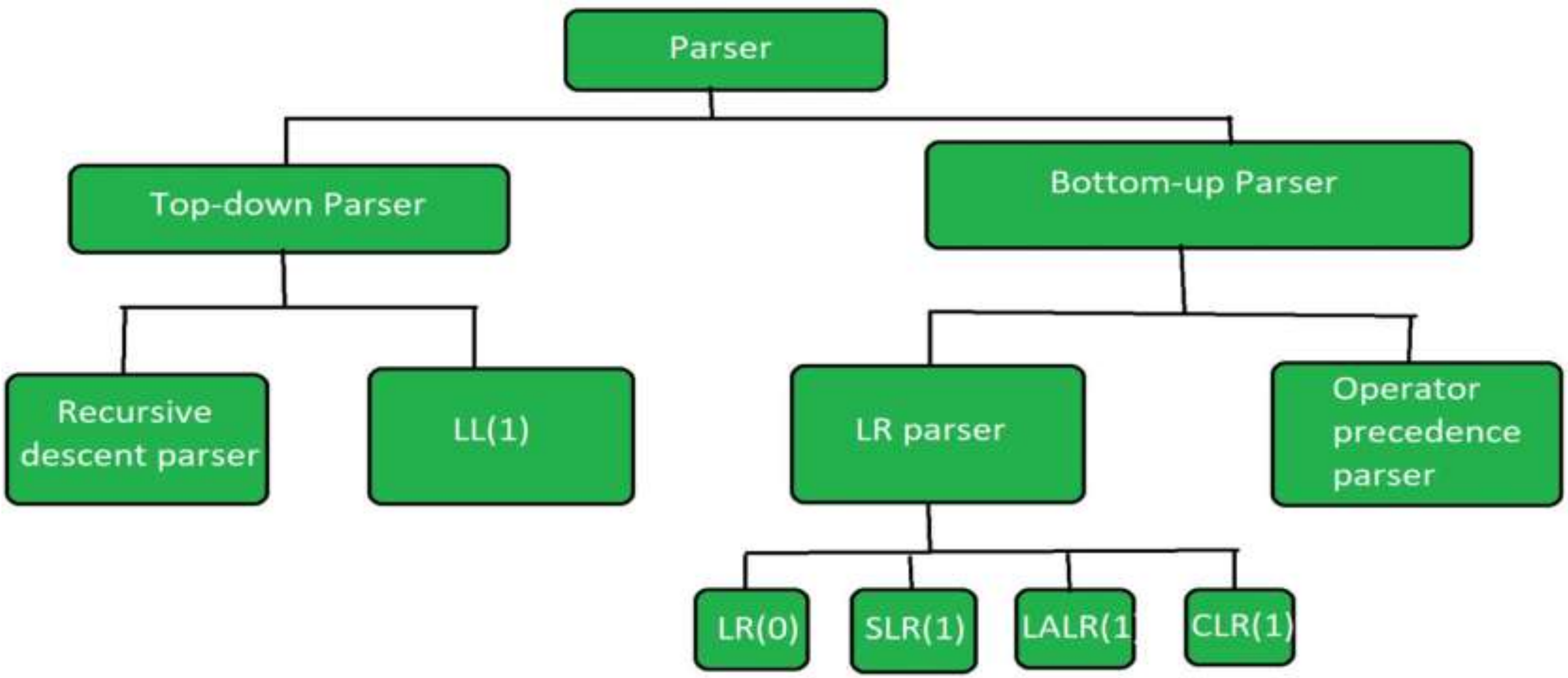


Image source : Google

Recursive Descent parser:

It is also known as Brute force parser or the with backtracking parser. It basically generates the parse tree by using brute force and backtracking.

Non Recursive Descent Parser OR LL (1):

It is also known as LL(1) parser or predictive parser or without backtracking parser or dynamic parser. It uses parsing table to generate the parse tree instead of backtracking.

LR parser:

LR parser is the bottom-up parser which generates the parse tree for the given string by using unambiguous grammar

Operator Precedence Parser:

It generates the parse tree form given grammar and string but the only condition is two consecutive non-terminals and epsilon never appears in the right-hand side of any production.

LR-Parser:

- A general shift reduce parsing is LR parsing.
- The L stands for scanning the input from left to right and R stands for constructing a rightmost derivation in reverse

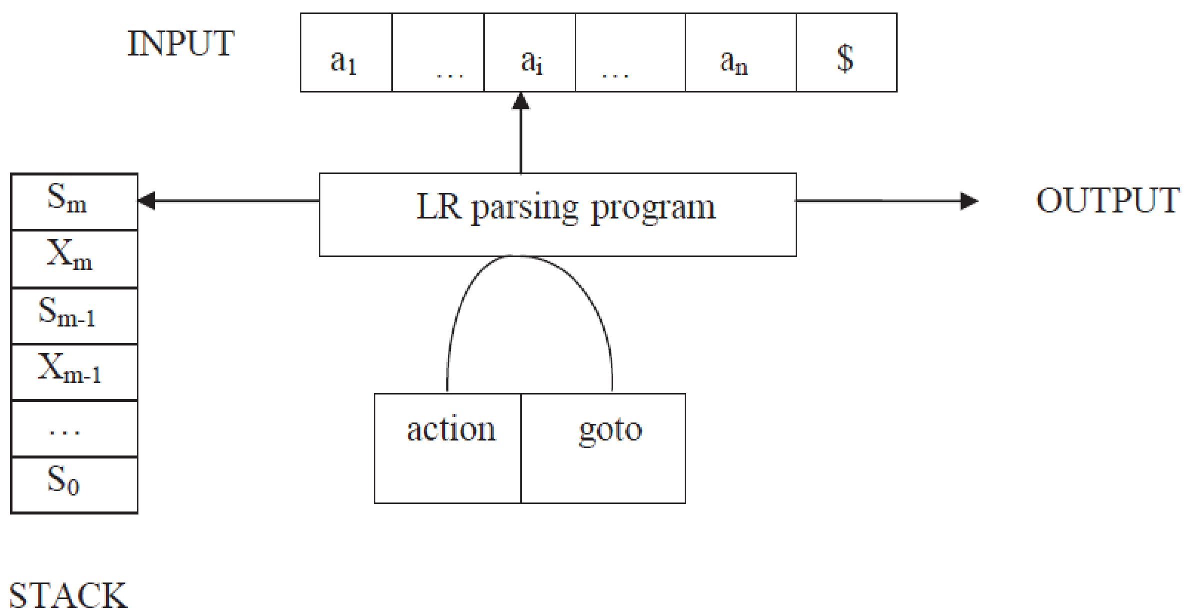
LR –parser are of following types:

(a). LR(0)
(b). SLR(1)
(c). LALR(1)
(d). CLR(1)

LR-Parser:

The LR parsing algorithm:

The schematic form of an LR parser is as follows



LR-Parser:

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 program reads characters from an input buffer one at a time.
The program uses a stack to store a string of the form $s_0X_1s_1X_2s_2\ldots X_ms_m$, where s_m is on top. Each X_i is a grammar symbol and each s_i is a state.
The parsing table consists of two parts :*action* and *goto* functions.

LR-Parser:

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

shift s, where s is a state,

reduce by a grammar production $A \rightarrow \beta$,

accept, and

error.

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

LR 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 s_0 on its stack, where s_0 is the initial state, and*w*$ in the input buffer. The parser then executes the following program :

Parul® University

DIGITAL LEARNING CONTENT

LR Parsing algorithm:

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

LR-Parser:

CONSTRUCTING SLR(1) 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 set of items and X is grammar symbol.

LR-Parser:

LR(O) items:

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

A →**.**XYZ

A → X**.**YZ

A → XY**.**Z

A → XYZ**.**

LR-Parser:

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:

Initially, every item in I is added to closure(I).

If A → a . Bβ is in closure(I) and B → y is a production, then add the item B → . y to I , if it is not already there.

We apply this rule until no more new items can be added to closure(I).

LR-Parser:

Goto operation:

Goto(I, X) is defined to be the closure of the set of all items [A→ aX . β] such that [A→ a . Xβ] is in I.

Goto(I, X) = 1. Add I by moving dot after X.

2. Apply closure to first step.

LR-Parser:

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

LR-Parser:

Algorithm for construction of SLR parsing table:

Input: An augmented grammar G'
Output: The SLR parsing table function s*action* and *goto*for G'
Method:
Construct C = {$I_{0,}$ $I_{1,}$ .... I_n}, the collection of sets of LR(0) items for G'.
State*i*is constructed from I $_{i.}$. The parsing functions for state *I* are determined as follows:
If [A→a·*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.
If [A→a·] is in I_{i} , then set*action*[*i*,*a*] to "reduce A→a" for al l*a* in FOLLOW(A).
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).

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

LR-Parser:

Example for SLR parsing:

Construct SLR parsing for the following grammar :

G : E → E + T | T

T → T * F | F F → (E) | id

The given grammar is :

G : E → E + T ------ (1)

E →T ------ (2)

T → T * F ------ (3)

T → F ------ (4)

F → (E) ------ (5)

F → id ------ (6)

LR-Parser:

Step 1 :Convert given grammar into augmented grammar.

Augmented grammar :

$E' \rightarrow E$

$E \rightarrow E + T$

$E \rightarrow T$

$T \rightarrow T * F$ $T \rightarrow F$

$F \rightarrow (E)$

$F \rightarrow id$

LR-Parser:

Step 2 :Find LR (0) items.

I0 : $E' \rightarrow .E$
$E \rightarrow .E + T$
$E \rightarrow .T$
$T \rightarrow .T * F$
$T \rightarrow .F$
$F \rightarrow .(E)$
$F \rightarrow .id$

LR-Parser:

GOTO (I0 , E)
I1 : E’ → E.
E → E.+ T

GOTO (I0 , T)
I2 : E → T.
T → T.* F
GOTO (I0 , F)
I3 : T → F.

GOTO (I0 , ( )
I4 : F → (.E)
E →.E + T
E →.T
T →.T * F
T →.F
F →.(E)
F →.id

GOTO (I0 , id)
I5 : F → id.

GOTO (I1 , +)
I6 : E → E +.T
T →.T * F
T →.F
F →.(E)
F →.id

LR-Parser:

GOTO (I2 , *)
I7 : T ⟶ T *.F
F ⟶.(E)
F ⟶.id

GOTO (I4 , E)
I8 : F ⟶ (E.)
E ⟶ E.+ T

GOTO (I6 , T)
I9 :
E ⟶ E + T.
T ⟶ T.* F

GOTO (I7 , F)
I10 : T ⟶ T * F.

GOTO (I8 ,))
I11 : F ⟶ (E).

LR-Parser:

FOLLOW (E) = { $,) , +)
FOLLOW (T) = { $, + ,) , * }
FOOLOW (F) = { * , + ,) , $ }

LR-Parser:

	ACTION						GOTO		
	id	+	*	(	)	$	E	T	F
I_0	s5			s4			1	2	3
I_1		s6				ACC			
I_2		r2	s7		r2	r2			
I_3		r4	r4		r4	r4			
I_4	s5			s4			8	2	3
I_5		r6	r6		r6	r6			
I_6	s5			s4				9	3
I_7	s5			s4					10
I_8		s6			s11				
I_9		r1	s7		r1	r1			
I_{10}		r3	r3		r3	r3			
I_{11}		r5	r5		r5	r5			

LR-Parser:

Stack implementation:

Check whether the input **id + id * id** is valid or not

STACK	INPUT	ACTION
0	id + id * id $	GOTO (I_0 , id) = s5 ;shift
0 id 5	+ id * id $	GOTO (I_5 , +) = r6 ;reduceby F→id
0 F 3	+ id * id $	GOTO (I_0 , F) = 3 GOTO (I_3 , +) = r4 ;reduceby T → F
0 T 2	+ id * id $	GOTO (I_0 , T) = 2 GOTO (I_2 , +) = r2 ;reduceby E → T

LR-Parser:

STACK	INPUT	ACTION
0 E 1	+ id * id $	GOTO (I_0 , E) = 1 GOTO (I_1 , +) = s6 ;shift
0 E 1 + 6	id * id $	GOTO (I_6 , id) = s5 ;shift
0 E 1 + 6 id 5	* id $	GOTO (I_5 , *) = r6 ;reduceby F $\rightarrow$ id
0 E 1 + 6 F 3	* id $	GOTO (I_6 , F) = 3 GOTO (I_3 , *) = r4 ;reduceby T $\rightarrow$ F
0 E 1 + 6 T 9	* id $	GOTO (I_6 , T) = 9 GOTO (I_9 , *) = s7 ;shift

Parul® University

DIGITAL LEARNING CONTENT

LR-Parser:

STACK	INPUT	ACTION
0 E 1 + 6 T 9 * 7	id $	GOTO (I_7 , id) = s5 ;shift
0 E 1 + 6 T 9 * 7 id 5	$	GOTO (I_5 , $) = r6 ;reduceby F → id
0 E 1 + 6 T 9 * 7 F 10	$	GOTO (I_7 , F) = 10 GOTO (I_{10} , $) = r3 ;reduceby T → T * F
0 E 1 + 6 T 9	$	GOTO (I_6 , T) = 9 GOTO (I_9 , $) = r1 ;reduceby E → E + T
0 E 1	$	GOTO (I_0 , E) = 1 GOTO (I_1 , $) =accept

Stack	Input buffer	action table	goto table	Parsing action
$0	id*id+id$	[0,id]=s5		Shift
$0id5	*id+id$	[5,*]=r6	[0,F]=3	Reduce by F → id
$0F3	*id+id$	[3,*]=r4	[0,T]=2	Reduce by T → F
$0T2	*id+id$	[2,*]=s7		Shift
$0T2*7	id+id$	[7,id]=s5		Shift
$0T2*7id5	+id$	[5,+]=r6	[7,F]=10	Reduce by F → id
$0 T2*7F10	+id$	[10,+]=r3	[0,T]=2	Reduce by T → T*F
$0T2	+id$	[2,+]=r2	[0,E]=1	Reduce by E → T
$0E1	+id$	[1,+]=s6		Shift
$0E1+6	id$	[6,id]=s5		Shift
$0E1+6id5	$	[5,$]=r6	[6,F]=3	Reduce by F → id
$0E1+6F3	$	[3,$]=r4	[6,T]=9	Reduce by T → F
$0E1+6T9	$	[9,$]=r1	[0,E]=1	E → E+T
$0E1	$	accept		Accept

DIGITAL LEARNING CONTENT

Parul® University

www.paruluniversity.ac.in