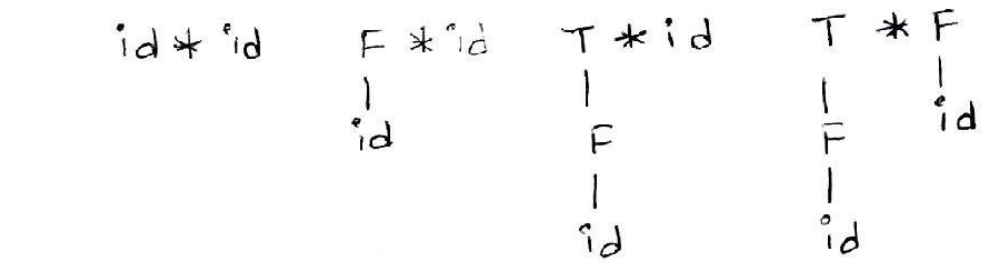
Bottom-up parsing :-

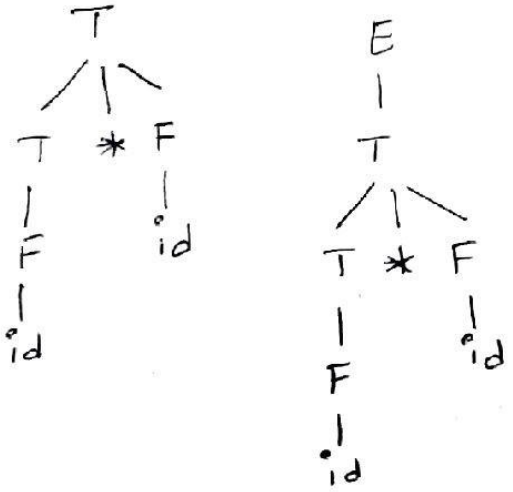
In Bottom up parsing the parse tree is constructed from bottom to up that is from leaves to root. In this process, the input symbols are placed at the leaf nodes after successful parsing. The bottom-up parse tree is created starting from leaves, the leaf nodes together are reduced further to internal nodes, these nodes are reduced and eventually a root node is obtained.

Example

$E \rightarrow E+T \mid T$
$T \rightarrow T*F \mid F$
$F \rightarrow (E) \mid id$

Input : id * id





Bottom - up parse for id * id

Reductions

In Bottom-up parsing as the process of reducing a string w to the start symbol of the grammar At each reduction step, a specific substring matching the body of a production is replaced by the non-terminal

Handle pruning

The crucial task in bottom-up parsing is to find the substring that could be reduced by appropriate non-terminal Such a substring is called handle.

In other words handle is a string of substring that matches the right side of production and we can reduce such string by a non-terminal on left hand side production

Example

Consider the grammar

$E \rightarrow E+E$

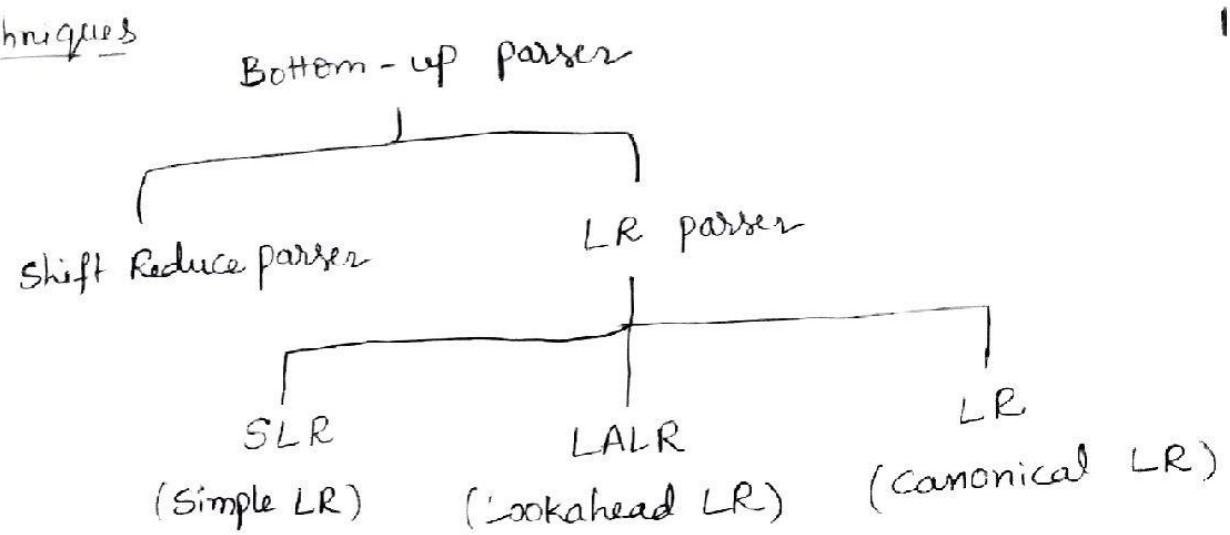
$E \rightarrow id$

the input string is $id + id + id$

Right Sentential form	Handle	Reducing production
$\underline{id} + id + id$	id	$E \rightarrow id$
$E + \underline{id} + id$	id	$E \rightarrow id$
$E + E + \underline{id}$	id	$E \rightarrow id$
$E + \underline{E + E}$	$E+E$	$E \rightarrow E+E$
$E + E$	$E+E$	$E \rightarrow E+E$
E		

Thus bottom parser is essentially a process of detecting handles and using them in reduction. This process is called Handle pruning. The rightmost derivation in reverse can be obtained by 'Handle pruning'.

Parsing techniques



Shift Reduce parser :-

Shift Reduce parser attempts to construct parse tree from leaves to root.

A shift reduce parser requires following data structures.

1 The input buffer storing the input string
2. A stack for storing and accessing the LHS and RHS of rules

The parser performs following basic operations

1. shift : Moving of the symbols from input buffer onto the stack.
2. Reduce : If the handle appears on the top of the stack then reduction of it by appropriate rule
3. Accept : If the stack contains start symbol only and input buffer is empty at the same time then that action is called Accept.
4. Error : A situation in which parser cannot either shift & reduce the symbols, it cannot even perform the accept action is called as error

Initial configuration of shif Reduce parser is

Stack	Input buffer
$S	w $

w is a input string

Example

$S \rightarrow (L)/a$

$L \rightarrow L,S/S$

input string is (a,a)

Stack	Input buffer	parsing Action
$	(a,a)$	shift (
$(	a,a)$	shift a
$(a	,a)$	Reduce $S \rightarrow a$
$(S	,a)$	Reduce $L \rightarrow S$
$(L	,a)$	shift ,
$(L,	a)$	shift a
$(L,a	)$	Reduce $S \rightarrow a$
$(L,S	)$	Reduce $L \rightarrow L,S$
$(L	)$	shift)
$(L)	$	Reduce $S \rightarrow (L)$
$S	$	Accept.

Introduction to LR parser

Why LR Parsers?

This is the most efficient method of bottom-up parsing which can be used to parse the large class of context free grammars. This method is also called LR(K) parsing.

LR(K)

L stands for left to right scanning

R stands for right most derivation in reverse

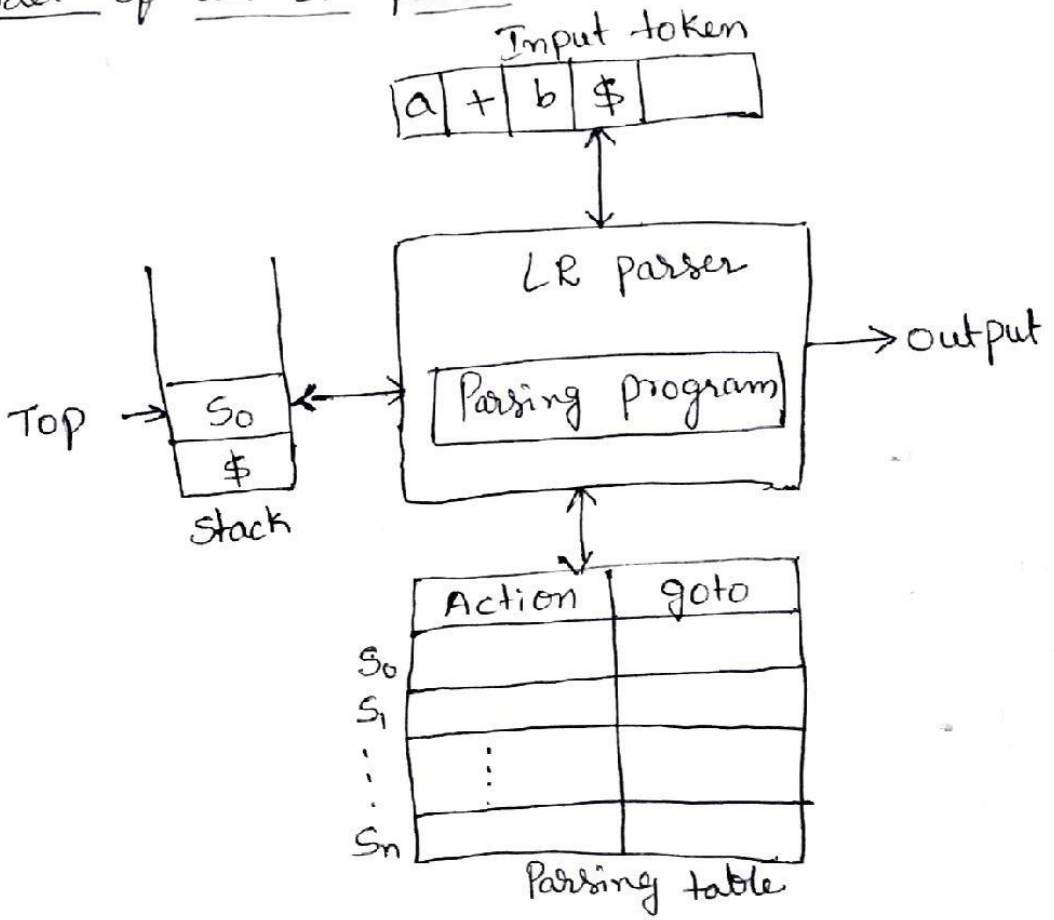
K stands for no. of input symbols. K is assumed to be 1

Properties of LR parser

(16) LR parser is used for following reasons

1. LR parsers can be constructed to recognize most of the programming languages for which context free grammar can be written
2. The class of grammar that can be parsed by LR parser is a superset of class of grammar that can be parsed using predictive parsers.
3. LR parser works using non backtracking shift reduce technique yet it is efficient one
4. LR parsers detect Syntactical errors very efficiently.

Model of an LR parser



Structure of LR parser

It consists of input buffer for storing the input string, a stack for storing the grammar symbols, output and a parsing table comprised of two parts, namely action and goto. There is one parsing program which is actually a driving program and reads the input symbol one at a time from the input buffer

The parsing program works as follows

1. It initializes the stack with ~~start~~ start symbol and invokes scanner to get next token
2. It determines S_j the state currently on the top of the stack and a_i the current input symbol
3. It consults the parsing table for the action$[S_j, a_i]$ which can have one of the four values
   (i) S_i means shift state i
   (ii) γ_j means reduce by rule j
   (iii) Accept means Successful parsing is done
   (iv) Error indicates syntactical error

Definition of LR(0) items and related terms:

1. The LR(0) items for grammar G is production rule in which symbol • is inserted at some position in R.H.S of the rule

for example

$S \rightarrow .ABC$
$S \rightarrow A.BC$
$S \rightarrow AB.C$
$S \rightarrow ABC.$

The production $S \rightarrow \epsilon$ generates only one item $S \rightarrow .$

2. Augmented grammar:

If a grammar G is having start symbol S then augmented grammar is a new grammar G' in which S' in a new start symbol such that $S' \rightarrow S$ That is when parser is about to reduce $S' \rightarrow S$ it reaches to acceptance state.

3. Kernel items:

It is a collection of items $S' \rightarrow .S$ and all the items whose dots are not at the leftmost end of R.H.S of the rule

4. Non kernel items:

The collection of all the items in which . are at the left end of R.H.S of the rule.

5. functions of closure and goto:

These functions are required to create collection of canonical set of items.

Clouse

consider I is a set of canonical items and initially every item I is added to closure (I)

if rule $A \rightarrow \alpha.B\beta$ is a rule in closure (I) and there is another rule for B such as $B \rightarrow \gamma$ then

closure (I): $A \rightarrow \alpha.B\beta$

$B \rightarrow .\gamma$

This rule has to be applied until no more new items can be added to clouse (I)

goto:

If there is a production $A \rightarrow \alpha.B\beta$ then goto $(A \rightarrow \alpha.B\beta, B) = A \rightarrow \alpha B.\beta$

goto ($A \to \alpha . B\beta, B$) = $A \to \alpha B . \beta$ ∴ goto(I_i, B)

that means simply shifting of • one position ahead over the grammar symbol (may be terminal or nonterminal)

SLR (Simple LR) :-

It is the weakest of the three methods but it is easiest to implement.

The parsing can be done as follows

1. Construction of canonical set of items
2. Construction of SLR parsing table
3. Input parsing

Construction of canonical set of items.

1. For the Grammar G initially add $S' \to .S$ in the set of items C.
2. For each set of items I_i in C and for each grammar Symbol X (may be terminal or non-terminal) add Closure (I_i, X). This process should be repeated by applying goto (I_i, X) for each X in I_i such that goto (I_i, X) is not empty and not in C. The set of items has to be constructed until no more set of Items can be added to C.

Example

$E \to E+T$
$E \to T$
$T \to T*F$
$T \to F$
$F \to (E)$
$F \to id$

I_0 : $E' \to .E$
$E \to .E+T$
$E \to .T$
$T \to .T*F$
$T \to .F$
$F \to .(E)$
$F \to .id$

I_1 : goto(I_0, E)
$E' \to E.$
$E \to E.+T$

I_2 : goto(I_0, T)
$E \to T.$
$T \to T.*F$

I_3 : goto(I_0, F)
$T \to F.$

I_4 : goto$(I_0, ()$
$F \to (.E)$
$E \to .E+T$
$E \to .T$
$T \to .T*F$
$T \to .F$
$F \to .(E)$
$F \to .id$

I_5 : goto(I_0, id)
$F \to id.$

I_6 : goto$(I_1, +)$
$E \to E+.T$
$T \to .T*F$
$T \to .F$
$F \to .(E)$
$F \to .id$

I_7 : goto$(I_2, *)$
$T \to T*.F$
$F \to .(E)$
$F \to .id$

I_8 : goto(I_4, E)
$F \to (E.)$
$E \to E.+T$

I_2 : goto(I_4, T)
$E \to T.$
$T \to T.*F$

I_3 : goto(I_4, F)
$T \to F.$

I_4 : goto$(I_4, ()$
$F \to (.E)$
$E \to .E+T$
$E \to .T$
$T \to .T*F$
$T \to .F$
$F \to .(E)$
$F \to .id$

I_5 : goto(I_4, id)
$F \to id.$

I_9 : goto(I_6, T)
$E \to E+T.$
$T \to T.*F$

I_3 : goto(I_6, F)
$T \to F.$

I_4 : goto$(I_6, ()$
$F \to (.E)$
$E \to .E+T$
$E \to .T$
$T \to .T*F$
$T \to .F$
$F \to .(E)$
$F \to .id$

I_5 : goto(I_6, id)
$F \to id.$

I_{10} : goto(I_7, F)
$T \to T*F.$

I_4 : goto$(I_7, ()$
$F \to (.E)$
$E \to .E+T$
$E \to .T$
$T \to .T*F$
$T \to .F$
$F \to .(E)$
$F \to .id$

I_5 : goto(I_7, id)

$F \rightarrow id.$

I_{11} : goto(I_8,))

$F \rightarrow (E).$

I_6 : goto(I_8, +)

$E \rightarrow E + .T$

$T \rightarrow .T * F$

$T \rightarrow .F$

$F \rightarrow .(E)$

$F \rightarrow .id$

I_7 : goto(I_9, *)

$T \rightarrow T * .F$

$F \rightarrow .(E)$

$F \rightarrow .id$

Construction of parsing table :

There are two parts of SLR parsing table. they are action and goto. basic parsing actions are shift, reduce accept and error. we will fill up the action table.

1. Initially construct set of items $C = \{I_0, I_1, \ldots I_n\}$ where C is a collection of set of LR(0) items for Grammar G'
2. The actions are based on each Item I_i:
   (a) If $A \rightarrow \alpha . a\beta$ is in I_i and goto(I_i, a) = I_j then set action[i, a] as "shift j". a must be a terminal symbol
   (b) If $A \rightarrow \alpha .$ is in I_i then set action[i, a] to "reduce $A \rightarrow \alpha$" for all symbols a, where $a \in$ Follow(A). A must not be an augmented grammar S'
   (c) If $S' \rightarrow S$ is in I_i then action[i, \$] = "accept"
3. The goto part of the SLR is for state i is considered for non-terminals only. If goto(I_i, A) = I_j then goto[I_i, A] = j
4. All the entries not defined by rule 2 and 3 are considered to be "error"

Now we will also prepare Follow set for all the non-terminals

$E = \{\$, +,)\}$ $T = \{*, +,), \$\}$ $F = \{+, *,), \$\}$

γ1. $E \to E+T$
γ2. $E \to T$
γ3. $T \to T*F$
γ4. $T \to F$
γ5. $F \to (E)$
γ6. $F \to id$

2nd rule

$E \to \underset{\alpha}{E} \overset{B}{+} \underset{\beta}{T}$

$= FIRST(\beta)$
$= FIRST(+T)$
$= \{+\}$

$E \to E + T$ (α B β) $\quad E \to T$ (α B β) $\quad F \to (E)$

$T \to T * F$ (α B β) $\quad F \to id$ ×

3rd rule

$E \to E + T$ (α B β)

$FIRST(\beta) \neq \epsilon$

$E \to E + T$ (α B β) ε

$FIRST(\beta) = \epsilon$

$E \to T$ (α B β) ε

$T \to T * F$ (α B β) ×

$FIRST(\beta) \neq \epsilon$

$T \to T * F$ (α B β) ε

$T \to F$ (α B β) ε

$F \to (E)$ (α B β)

$FIRST(\beta) \neq \epsilon$

$F \to id$ ×

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

Parsing table

	Action						Goto		
	id	+	*	(	)	$	E	T	F
0	S_5			S_4			1	2	3
1		S_6				Accept			
2		γ_2	S_7		γ_2	γ_2			
3		γ_4	γ_4		γ_4	γ_4			
4	S_5			S_4			8	2	3
5		γ_6	γ_6		γ_6	γ_6			
6	S_5			S_4				9	3
7	S_5			S_4					10
8		S_6			S_{11}				
9		γ_1	S_7		γ_1	γ_1			
10		γ_3	γ_3		γ_3	γ_3			
11		γ_5	γ_5		γ_5	γ_5			

Input parsing

1. Initially push 0 as initial state onto the stack and 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 Input Symbol then
   (a) if action[S, a] = shift j then push a, then push j onto the stack
   (b) If action[S, a] = reduce A → β then pop |β| symbols if i is on the top of the stack then push A, then goto[i, A] on the top of the stack.
   (c) if action[S, a] = accept then halt the parsing process it indicates Accept the String.

Stack	Input buffer	Action	goto	Parsing action
				Shift
$0	id * id + id $	[0, id] = S5		
$0 id 5	* id + id $	[5, *] = r6	[0, F] = 3	Reduce by F → id
$0 F 3	* id + id $	[3, *] = r4	[0, T] = 2	Reduce by T → F
$0 T 2	* id + id $	[2, *] = S7		shift
$0 T 2 * 7	id + id $	[7, id] = S5		shift
$0 T 2 * 7 id 5	+ id $	[5, +] = r6	[7, F] = 10	Reduce by F → id
$0 T 2 * 7 F 10	+ id $	[10, +] = r3	[0, T] = 2	Reduce T → T * F
$0 T 2	+ id $	[2, +] = r2	[0, E] = 1	Reduce E → T
$0 E 1	+ id $	[1, +] = S6		shift
$0 E 1 + 6	id $	[6, id] = S5		shift
$0 E 1 + 6 id 5	$	[5, $] = r6	[6, F] = 3	Reduce F → id
$0 E 1 + 6 F 3	$	[3, $] = r4	[6, T] = 9	Reduce T → F
$0 E 1 + 6 T 9	$	[9, $] = r1	[0, E] = 1	Reduce E → E + T
$0 E 1	$	[1, $] = Accept		Accepted

$ (L, (L	)) $	Shift
$ (L, (L)	) $	Reduce S → (L)
$ (L, S	) $	Reduce L → L, S
$ (L	) $	Shift
$ (L)	$	Reduce S → (L)
$ S	$	Accept

Example 4.5.4 *Design shift reduce Parser for the following grammar :*

$S \rightarrow 0\ S\ 0 \mid 1\ S\ 1 \mid 2$

Solution : To design the shift-reduce parser we will consider the input "10201".

Stack	Input Buffer	Parsing Action
$	10201$	Shift
$1	0201$	Shift
$10	201$	Shift
$102	01$	Reduce S → 2
$10 S	01$	Shift
$10S0	1$	Reduce S → 0S0
$1S	1$	Shift
$1S1	$	Reduce S → 1S1
$S	$	Accept

4.6 Difference between LR and LL Parsers

Sr. No.	LR Parsers	LL Parsers
1.	These are bottom up parsers	These are top down parsers
2.	This is complex to implement	This is simple to implement
3.	For LR(1) the first L means the input is scanned from left to right. The second R means it uses rightmost derivation in reverse for the input string. The number 1 indicates that one lookahead symbol to predict the parsing process.	For LL(1) the first L means the input is scanned from left to right. The second L means it uses leftmost derivation for the input string. The number 1 indicates that one lookahead symbol to predict the parsing process.
4.	These are efficient parsers.	This is less efficient.
5.	It is applied to a large class of programming languages.	It is applied to small class of languages.

Q.8 *Distinguish between top-down and bottom-up parsing.*

May-05, 04, Set-1,3; Jan.-10, Set-4, Marks 8

Ans. :

Sr. No.	Top down parser	Bottom up parser
1.	Parse tree can be built from root to leaves.	Parse tree is built from leaves to toot
2.	This is simple to implement.	This complex to to implement.
3.	This is less efficient parsing techniques. Various problems that occur during top down technique are ambiguity left recursion	When the bottom up parser handles ambiguous grammar conflicts occur in parse table.
4.	It is applicable to small class of languages.	It is applicable to a broad class of languages.
5.	Various parsing techniques are 1) Recursive descent parser 2) Predictive parser	Various parsing techniques are 1) Shift reduce 2) Operator precedence 3) LR parser.