Bottom up parsing

1. BOTTOM UP PARSING:

Bottom-up parser builds a derivation by working from the input sentence back towards the start symbol S. Right most derivation in reverse order is done in bottom-up parsing.

(The point of parsing is to construct a derivation. A derivation consists of a series of rewrite steps)

$S \Rightarrow r_0 \Rightarrow r_1 \Rightarrow r_2 \Rightarrow - - - \Rightarrow r_{n-1} \Rightarrow r_n \Rightarrow \text{sentence}$

$\longleftarrow$

Bottom-up

Assuming the production $A \rightarrow \beta$, to reduce r_i r_{i-1} match some RHS β against r_i then replace β with its corresponding LHS, A.

In terms of the parse tree, this is working from leaves to root.

Example – 1:

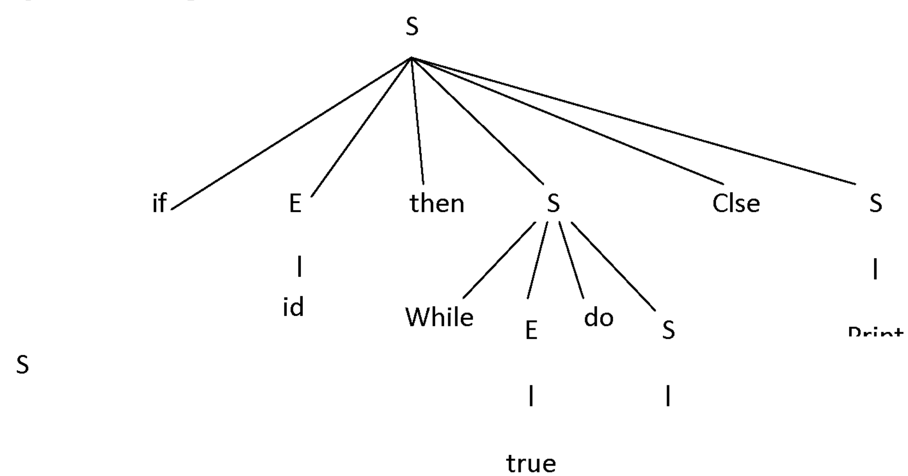
S→if E then S else S/while E do S/ print

E→ true/ False/id

Input: if id then while true do print else print.

Parse tree:

Basic idea: Given input string a, "reduce" it to the goal (start) symbol, by looking for substring that match production RHS.



$\underset{lm}{\Rightarrow}$ if E then S else S

$\underset{lm}{\Rightarrow}$ if id then S else S

$\underset{lm}{\Rightarrow}$ if id then while E do S else S

$\underset{lm}{\Rightarrow}$ if id then while true do S else S

$\underset{lm}{\Rightarrow}$ if id then while true do print else S

$\underset{lm}{\Rightarrow}$ if id then while true do print else print

$\underset{rm}{\Leftarrow}$ if E then while true do print else print

$\underset{rm}{\Leftarrow}$ if E then while E do print else print

$\underset{rm}{\Leftarrow}$ if E then while E do S else print

$\underset{rm}{\Leftarrow}$ if E then S else print

$\underset{rm}{\Leftarrow}$ if E then S else S

$\underset{rm}{\Leftarrow}$ S

1.1 Topdown Vs Bottom-up parsing:

Top-down	Bottom-up
1. Construct tree from root to leaves 2. "Guers" which RHS to substitute for nonterminal 3. Produces left-most derivation 4. Recursive descent, LL parsers 5. Recursive descent, LL parsers 6. Easy for humans	1. Construct tree from leaves to root 2. "Guers" which rule to "reduce" terminals 3. Produces reverse right-most derivation. 4. Shift-reduce, LR, LALR, etc. 5. "Harder" for humans.

→ Bottom-up can parse a larger set of languages than topdown.

→ Both work for most (but not all) features of most computer languages.

Example – 2:

		Right-most derivation
S→aAcBe	Ilp: abbcde/	S→aAcBe
A→Ab/b		→ aAcde
B→d		→aAbcde
		→ abbcde

Bottom-up approach

"Right sentential form"	Reduction
abbcde	
aAbcde	A→b
Aacde	A→Ab
AacBe	B→d
S	S→aAcBe

Steps correspond to a right-most derivation in reverse.

(must choose RHS wisely)

Example – 3:

S→aABe

A→Abc/b

B→d

1/p: abbcde

Right most derivation:

S	→	aABe	
	→	aAde	Since ( ) B→d
	→	aAbcde	Since ( ) A→Abc
	→	abbcde	Since ( ) A→b

Parsing using Bottom-up approach:

Input	Production used
abbcde	
aAbcde	$A \rightarrow b$
AAde	$A \rightarrow Abc$
AABe	$B \rightarrow d$

S parsing is completed as we got a start symbol

Hence the 1/p string is acceptable.

Example – 4

$E \rightarrow E+E$

$E \rightarrow E*E$

$E \rightarrow (E)$

$E \rightarrow id$

1/p: $id_1+id_2+id_3$

Right most derivation

$E \rightarrow E+E$

$\rightarrow E+E*E$

$\rightarrow E+E*id_3$

$\rightarrow E+id_2*id_3$

$\rightarrow id_1+id_2*id_3$

Parsing using Bottom-up approach:

Go from left to right

$id_1+id_2*id_3$	
$E+id_2*id_3$	$E \rightarrow id$
$E+E*id_3$	$E \rightarrow id$
$E*id_3$	$E \rightarrow E+E$
$E*E$	$E \rightarrow id$
E	

= start symbol, Hence acceptable.

2. HANDLES:

Always making progress by replacing a substring with LHS of a matching production will not lead to the goal/start symbol.

For example:

abbcde

aAbcde $\quad A \rightarrow b$

aAAcde $\quad A \rightarrow b$

struck

Informally, A Handle of a string is a substring that matches the right side of a production, and whose reduction to the non-terminal on the left side of the production represents one step along the reverse of a right most derivation.

If the grammar is unambiguous, every right sentential form has exactly one handle.

More formally, A handle is a production $A \rightarrow \beta$ and a position in the current right-sentential form $\alpha\beta\omega$ such that:

$S \Rightarrow \alpha A \omega \Rightarrow \alpha/\beta\omega$

For example grammar, if current right-sentential form is

a/Abcde

Then the handle is $A \rightarrow Ab$ at the marked position. ‘a’ never contains non-terminals.

2.1HANDLE PRUNING:

Keep removing handles, replacing them with corresponding LHS of production, until we reach S.

Example:

$E \rightarrow E+E/E*E/(E)/id$

Right-sentential form	Handle	Reducing production
a+b*c	a	$E \rightarrow id$
E+b*c	b	$E \rightarrow id$

E+E*C	C	E→id
E+E*E	E*E	E→E*E
E+E	E+E	E→E+E
E		

The grammar is ambiguous, so there are actually two handles at next-to-last step.

We can use parser-generators that compute the handles for us.

3. SHIFT- REDUCE PARSING:

Shift Reduce Parsing uses a stuck to hold grammar symbols and input buffer to hold string to be parsed, because handles always appear at the top of the stack i.e., there's no need to look deeper into the state.

A shift-reduce parser has just four actions:

1. Shift-next word is shifted onto the stack (input symbols) until a handle is formed.
2. Reduce – right end of handle is at top of stack, locate left end of handle within the stack. Pop handle off stack and push appropriate LHS.
3. Accept – stop parsing on successful completion of parse and report success.
4. Error – call an error reporting/recovery routine.

3.1 Possible Conflicts:

Ambiguous grammars lead to parsing conflicts.

1. **Shift-reduce:** Both a shift action and a reduce action are possible in the same state (should we shift or reduce)

Example: dangling-else problem

2. **Reduce-reduce:** Two or more distinct reduce actions are possible in the same state. (Which production should we reduce with 2).

Example:

Stmt $\rightarrow$id (param)　　(a(i) is procedure call)

Param$\rightarrow$ id

Expr $\rightarrow$ id (expr) /id　(a(i) is array subscript)

Stack　　　　　　　　input buffer　　　　　action

\$…aa (i　　　) ….\$　　　　Reduce by ?

Should we reduce to param or to expr? Need to know the type of a: is it an array or a function. This information must flow from declaration of a to this use, typically via a symbol table.

3.2 Shift – reduce parsing example: (Stack implementation)

Grammar:　　E$\rightarrow$E+E/E*E/(E)/id

Input:　$id_1+id_2+id_3$

One Scheme to implement a handle-pruning, bottom-up parser is called a shift-reduce parser. Shift reduce parsers use stack and an input buffer.

The sequence of steps is as follows:

1. initialize stack with \$.
2. Repeat until the top of the stack is the goal symbol and the input token is "end of life".

   a. Find the handle

If we don't have a handle on top of stack, shift an input symbol onto the stack.

   b. Prune the handle

   if we have a handle (A$\rightarrow\beta$) on the stack, reduce

   (i) pop /β/ symbols off the stack (ii)push A onto the stack.

Stack	input	Action
\$	$id_1+id_2*id_3$\$	Shift
\$ id_1	$+id_2*id_3$\$	Reduce by E$\rightarrow$id
\$E	$+id_2*id_3$\$	Shift
\$E+	id_2*id_3\$	Shift
\$E+ id_2	$*id_3$\$	Reduce by E$\rightarrow$id

$E+E	*$id_3$$	Shift
$E+E*	$id_3$$	Shift
$E+E* id_3	$	Reduce by E→id
$E+E*E	$	Reduce by E→E*E
$E+E	$	Reduce by E→E+E
$E	$	Accept

Example 2:

Goal → Expr

Expr → Expr + term | Expr – Term | Term

Term → Tem & Factor | Term | factor | Factor

Factor → number | id | (Expr)

The expression grammar : x – z * y

Stack	Input	Action
$	Id - num * id	Shift
$ id	- num * id	Reduce factor → id
$ Factor	- num * id	Reduce Term → Factor
$ Term	- num * id	Reduce Expr → Term
$ Expr	- num * id	Shift
$ Expr -	num * id	Shift
$ Expr – num	* id	Reduce Factor → num
$ Expr – Factor	* id	Reduce Term → Factor
$ Expr – Term	* id	Shift
$ Expr – Term *	id	Shift

$ Expr – Term * id	-	Reduce Factor → id
$ Expr – Term & Factor	-	Reduce Term → Term * Factor
$ Expr – Term	-	Reduce Expr → Expr – Term
$ Expr	-	Reduce Goal → Expr
$ Goal	-	Accept

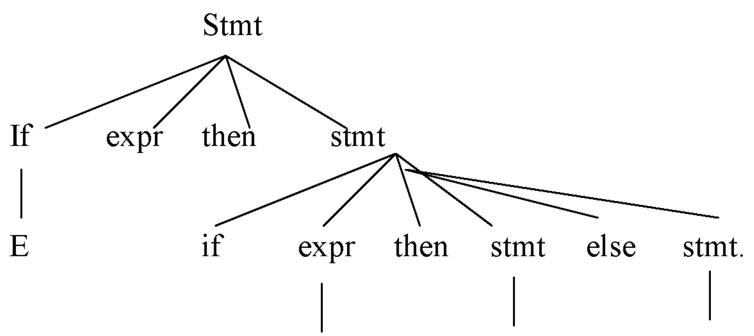
1. shift until the top of the stack is the right end of a handle
2. Find the left end of the handle & reduce.

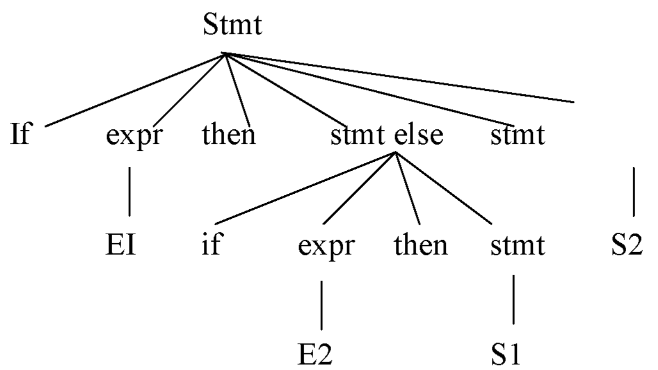
Procedure:

1. Shift until top of stack is the right end of a handle.
2. Find the left end of the handle and reduce.

* Dangling-else problem:

stmt→if expr then stmt/if expr then stmt/other then example string is: if E_1 then if E_2 then S_1 else S_2 has two parse trees (ambiguity) and so this grammar is not of LR(k) type.





3. OPERATOR – PRECEDENCE PARSING:

Precedence/ Operator grammar: The grammars having the property:

1. **No production right side is should contain ∈.**
2. **No production sight side should contain two adjacent non-terminals.**

Is called an **operator grammar.**

Operator – precedence parsing has three disjoint precedence relations, <.,=and .> between certain pairs of terminals. These precedence relations guide the selection of handles and have the following meanings:

RELATION	MEANING
a<.b	'a' yields precedence to 'b'.
a=b	'a' has the same precedence 'b'
a.>b	'a' takes precedence over 'b'.

Operator precedence parsing has a number of disadvantages:

1. It is hard to handle tokens like the minus sign, which has two different precedences.
2. Only a small class of grammars can be parsed.
3. The relationship between a grammar for the language being parsed and the operator-precedence parser itself is tenuous, one cannot always be sure the parser accepts exactly the desired language.

Disadvantages:

1. **L(G) ≠L(parser)**
2. **error detection**
3. **usage is limited**
4. **They are easy to analyse manually Example:**

Grammar: $E \rightarrow EAE|(E)|-E/id$

$A \rightarrow +|-|*|/|\uparrow$

Input string: id+id*id

The operator – precedence relations are:

	Id	+	*	$
Id		.>	.>	.>
+	<.	.>	<.	.>
*	<.	.>	.>	.>
$	<.	<.	<.	

Solution: This is not operator grammar, so first reduce it to operator grammar form, by eliminating adjacent non-terminals.

Operator grammar is:

E→E+E|E-E|E*E|E/E|E↑E|(E)|-E|id

The input string with precedence relations interested is:

$<.id.> + <.id.> * <.id.> $

Scan the string the from left end until first .> is encounted.

$<.id.>+<.id.>*<.id.<$

This occurs between the first id and +.

Scan backwards (to the left) over any ='s until a <. Is encounted. We scan backwards to $.

$<.id.>+<.id.>*<.id.>$

↑ ↑

Everything to the left of the first .> and to the right of <. Is called handle. Here, the handle is the first id.

Then reduce id to E. At this point we have:

E+id*id

By repeating the process and proceding in the same way:

$+<.id.>*<.id.>$

substitute E→id,

After reducing the other id to E by the same process, we obtain the right-sentential form

E+E*E

Now, the 1/p string afte detecting the non-terminals sis:

⇒ $+*$

Inserting the precedence relations, we get:

$<.+<.*.>$

↑ ↑

The left end of the handle lies between + and * and the right end between * and $. It indicates that, in the right sentential form E+E*E, the handle is E*E.

Reducing by E→E*E, we get:

E+E

Now the input string is:

$<.+$

Again inserting the precedence relations, we get:

⇒$<.+.>$

↑ ↑

reducing by E→E+E, we get,

$ $

and finally we are left with:

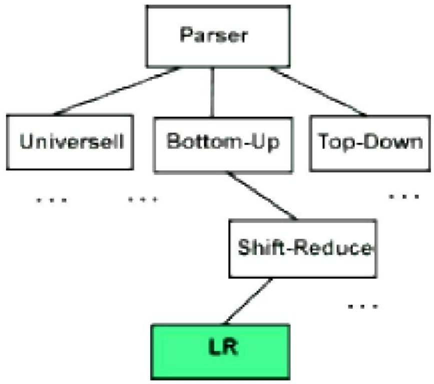
E

Hence accepted.

Input string	Precedence relations inserted	Action
id+id*id	$<.id.>+<.id.>*<.id.>$	
E+id*id	$+<.id.>*<.id.>$	E→id
E+E*id	$+*<.id.>$	E→id
E+E*E	$+*$	
E+E*E	$<.+<.*.>$	E→E*E
E+E	$<.+$	
E+E	$<.+.>$	E→E+E
E	$$	Accepted

5. LR PARSING INTRODUCTION:

The "L" is for left-to-right scanning of the input and the "R" is for constructing a rightmost derivation in reverse.



5.2 WHY LR PARSING:

1. LR parsers can be constructed to recognize virtually all programming-language constructs for which context-free grammars can be written.
2. The LR parsing method is the most general non-backtracking shift-reduce parsing method known, yet it can be implemented as efficiently as other shift-reduce methods.
3. The class of grammars that can be parsed using LR methods is a proper subset of the class of grammars that can be parsed with predictive parsers.
4. An LR parser can detect a syntactic error as soon as it is possible to do so on a left-to-right scan of the input.

The disadvantage is that it takes too much work to constuct an LR parser by hand for a typical programming-language grammar. But there are lots of LR parser generators available to make this task easy.

5.3 LR PARSERS:

LR(k) parsers are most general non-backtracking shift-reduce parsers. Two cases of interest are k=0 and k=1. LR(1) is of practical relevance.

‘L’ stands for “Left-to-right” scan of input.

‘R’ stands for “Rightmost derivation (in reverse)”.

‘K’ stands for number of input symbols of look-a-head that are used in making parsing decisions. When (K) is omitted, ‘K’ is assumed to be 1.

LR(1) parsers are table-driven, shift-reduce parsers that use a limited right context (1 token) for handle recognition.

LR(1) parsers recognize languages that have an LR(1) grammar.

A grammar is LR(1) if, given a right-most derivation

$$S \Rightarrow r_0 \Rightarrow r_1 \Rightarrow r_2 - - - r_{n-1} \Rightarrow r_n \Rightarrow \text{sentence.}$$

We can isolate the handle of each right-sentential form r_i and determine the production by which to reduce, by scanning r_i from left-to-right, going atmost 1 symbol beyond the right end of the handle of r_i.

Parser accepts input when stack contains only the start symbol and no remaining input symbol are left.

LR(0) item: (no lookahead)

Grammar rule combined with a dot that indicates a position in its RHS.

Ex– 1: $S^I \rightarrow .S\$$

$S \rightarrow .x$

$S \rightarrow .(L)$

Ex-2: A→XYZ generates 4LR(0) items –

A→.XYZ

A→X.YZ

A→XY.Z

A→XYZ.

The ‘.’ Indicates how much of an item we have seen at a given state in the parse.

A→.XYZ indicates that the parser is looking for a string that can be derived from XYZ.

A→XY.Z indicates that the parser has seen a string derived from XY and is looking for one derivable from Z.

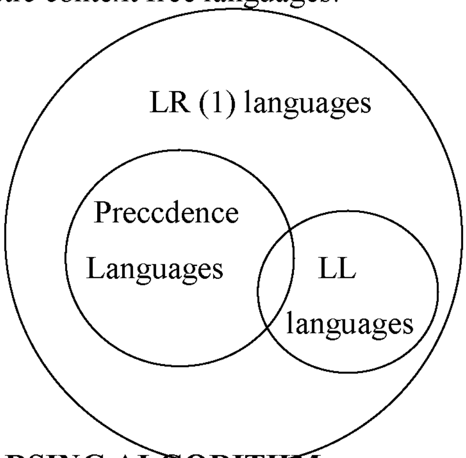
→ LR(0) items play a key role in the SLR(1) table construction algorithm.

→ LR(1) items play a key role in the LR(1) and LALR(1) table construction algorithms.

LR parsers have more information available than LL parsers when choosing a production:

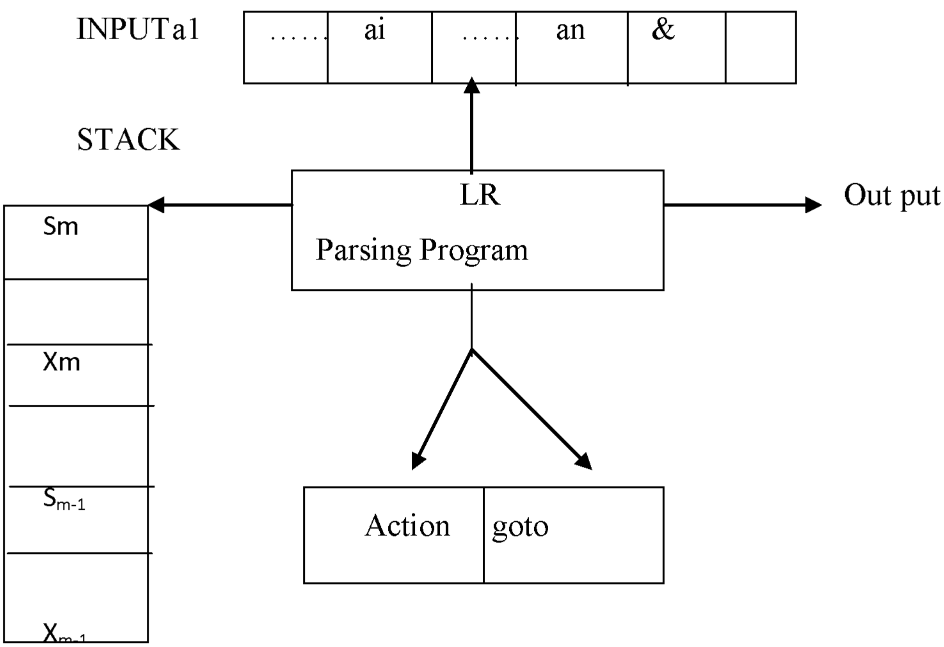
* **LR knows everything derived from RHS plus 'K' lookahead symbols.**
* **LL just knows 'K' lookahead symbols into what's derived from RHS.**

Deterministic context free languages:



5.4 LR PARSING ALGORITHM:

The schematic form of an LR parser is shown below:



It consists of an input, an output, a stack, a driver program, and a parsing table that has two parts: action and goto.

The LR parser program determines Sm, the current state on the top of the stack, and a_i, the current input symbol. It then consults action [Sm, a_i], 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**

The function goes to takes a state and grammar symbol as arguments and produces a state. The goto function of a parsing table constructed from a grammar G using the SLR, canonical LR or LALR method is the transition function of DFA that recognizes the viable prefixes of G. (Viable prefixes of G are those prefixes of right-sentential forms that can appear on the stack of a shift-reduce parser, because they do not extend past the right-most handle)

5.6 AUGMENTED GRAMMAR:

If G is a grammar with start symbol S, then G^I, the augmented grammar for G with a new start symbol S^I and production $S^I \rightarrow S$.

The purpose of this new start stating production is to indicate to the parser when it should stop parsing and announce acceptance of the input i.e., acceptance occurs when and only when the parser is about to reduce by $S^I \rightarrow S$.

CONSTRUCTION OF SLR PARSING TABLE:

Example:

The given grammar is:

1. **E→E+T**
2. **E→ T**
3. **T →T*F**
4. **T→F**
5. **F→(E)**
6. **F→id Step I: The Augmented grammar is:**

$E^I \rightarrow E$

$E \rightarrow E+T$

$E \rightarrow T$

$T \rightarrow T*F$

$T \rightarrow F$

$F \rightarrow (E)$

$F \rightarrow id$

Step II: The collection of LR (0) items are:

I_0: $E^I \rightarrow .E$

$E \rightarrow .E+T$

$E \rightarrow .T$

$T \rightarrow .T*F$

$T \rightarrow .F$

$F \rightarrow .(E)$

$F \rightarrow .id$

Start with start symbol after since ( ) there is E, start writing all productions of E.

Start writing 'T' productions

Start writing F productions

Goto (I_0,E): States have successor states formed by advancing the marker over the symbol it preceeds. For state 1 there are successor states reached by advancing the masks over the symbols E,T,F,C or id. Consider, first, the

I_1: $E^I \rightarrow E.$ - reduced Item (RI)

$E \rightarrow E.+T$

Goto (I_0,T):

I_2: $E \rightarrow T.$ - reduced Item (RI)

$T \to T.*F$

Goto (I_0,F):

I_2: $E \to T.$ - reduced item (RI)

$T \to T.*F$

Goto (I_0,C):

I_4: $F \to (.E)$

$E \to .E+T$

$E \to .T$

$T \to .T*F$

$T \to .F$

$F \to .(E)$

$F \to .id$

If '.' Precedes non-terminal start writing its corresponding production. Here first E then T after that F.

Start writing F productions.

Goto (I_0,id):

I_5: $F \to id.$ - reduced item.

E successor (I, state), it contains two items derived from state 1 and the closure operation adds no more (since neither marker precedes a non-terminal). The state I_2 is thus:

Goto (I_1,+):

I_6: $E \to E+.T$ start writing T productions

$T \to .T*F$

$T \to .F$ start writing F productions

$F \to .(E)$

$F \to .id$

Goto (I_2,*):

I_7: $T \rightarrow T^*.F$ start writing F productions

F→.(E)

F→.id

Goto (I_4,E):

I_8: F→(E.)

E→E.+T

Goto (I_4,T):

I_2: $E \rightarrow T.$ these are same as I_2.

T→T.*F

Goto (I_4,C):

I_4: F→(.E)

E→.E+T

E→.T

T→.T*F

T→.F

F→.(E)

F→.id

goto (I_4,id):

I_5: F→id. - reduced item

Goto (I_6,T):

I_9: E→E+T. - reduced item

T→T.*F

Goto (I_6,F):

I_3: T→F. - reduced item

Goto (I_6,C):

I_4: F→(.E)

E→.E+T

E→.T

T→.T*F

T→.F

F→.(E)

F→.id

Goto (I_6,id):

I_5: F→id. - reduced item.

Goto (I_7,F):

I_{10}: T→T*F - reduced item

Goto (I_7,C):

I_4: F→(.E)

E→.E+T

E→.T

T→.T*F

T→.F

F→.(E)

F→.id

Goto (I_7,id):

I_5: F→id. - reduced item

Goto (I_8,)):

I_{11}: $F \rightarrow (E).$ - reduced item

Goto (I_8,+):

I_{11}: $F \rightarrow (E).$ - reduced item

Goto (I_8,+):

I_6: $E \rightarrow E+.T$

$T \rightarrow .T*F$

$T \rightarrow .F$

$F \rightarrow .(E)$

$F \rightarrow .id$

Goto (I_9,+):

I_7: $T \rightarrow T*.f$

$F \rightarrow .(E)$

$F \rightarrow .id$

Step IV: Construction of Parse table:

Construction must proceed according to the algorithm 4.8

S→shift items

R→reduce items

Initially $E^I \rightarrow E.$ is in I_1 so, I = 1.

Set action [I, $] to accept i.e., action [1, $] to Acc

Action									Goto
State	Id	+	*	(	)	$	E	T	F
I_0	S_5			S_4			1	2	3
1		S_6				Accept			
2		r_2	S_7		R_2	R_2			

3		R_4	R_4		R_4	R_4			
4	S_5			S_4			8	2	3
5		R_6	R_6		R_6	R_6			
6	S_5			S_4				9	3
7	S_5			S_4					10
8		S_6			S_{11}				
9		R_1	S_7		r_1	r_1			
10		R_3	R_3		R_3	R_3			
11		R_5	R_5		R_5	R_5			

As there are no multiply defined entries, the grammar is SLR®.

STEP – III Finding FOLLOW ( ) set for all non-terminals.

Relevant production

FOLLOW (E) = {$} U FIRST (+T) U FIRST ( )) $\quad$ $E \rightarrow {}^{E}/_{B} + {}^{T}/_{B}$

= {+,), $} $\quad$ $F \rightarrow (E)$

$B\beta$

FOLLOW (T) = FOLLOW (E) U $\quad$ $E \rightarrow T$

FIRST (*F) U $\quad$ $T \rightarrow T*F$

FOLLOW (E) $\quad$ $E \rightarrow E+T$

B

= {+,*,),$}

FOLLOW (F) = FOLLOW (T)

= {*,*,),$}

Step – V:

1. Consider I_0:

1. The item $F \rightarrow .(E)$ gives rise to goto $(I_0,C) = I_4$, then action [0,C] = shift 4
2. The item $F \rightarrow .id$ gies rise goto $(I_0,id) = I_4$, then action [0,id] = shift 5

the other items in I_0 yield no actions.

Goto $(I_0,E) = I_1$ then goto [0,E] = 1

Goto (I_0,T) = I_2 then goto [0,T] = 2

Goto (I_0,F) = I_3 then goto [0,F] = 3

2. Consider I_1:

   1. The item $E^I \rightarrow E.$ is the reduced item, so I = 1

      This gives rise to action [1,$] to accept.

   2. The item $E \rightarrow E.+T$ gives rise to

      goto (I_1,+)=I_6, then action [1,+] = shift 6.

   3. Consider I_2:

      1. The item $E \rightarrow T.$ is the reduced item, so take FOLLOW (E),

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

The first item +, makes action [Z,+] = reduce $E \rightarrow T$.

$E \rightarrow T$ is production rule no.2. So action [Z,+] = reduce 2.

The second item, makes action [Z,)] = reduce 2

The third item $, makes action [Z,$] = reduce 2

2. The item $T \rightarrow T.*F$ gives rise to

goto [I_2,*]=I_7, then action [Z,*] = shift 7.

   4. Consider I_3:

      1. $T \rightarrow F.$ is the reduced item, so take FOLLOW (T).

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

So, make action [3,+] = reduce 4

Action [3,*] = reduce 4

Action [3,)] = reduce 4

Action [3,$] = reduce 4

In forming item sets a closure operation must be performed to ensure that whenever the marker in an item of a set precedes a non-terminal, say E, then initial items must be included in the set for all productions with E on the left hand side.

The first item set is formed by taking initial item for the start state and then performing the closure operation, giving the item set;

We construct the action and goto as follows:

1. **If there is a transition from state I to state J under the terminal symbol K, then set action [I,k] to S_J.**

2. **If there is a transition under a non-terminal symbol a, say from state 'i' to state 'J', set goto [I,A] to S_J.**

3. **If state I contains a transition under $ set action [I,$] to accept.**

4. **If there is a reduce transition #p from state I, set action [I,k] to reduce #p for all terminals k belonging to FOLLOW (A) where A is the subject to production #P.**

If any entry is multiply defined then the grammar is not SLR(1). Blank entries are represented by dash (-).

5. **Consider I_4 items:**

The item F→id gives rise to goto [I_4,id] = I_5 so,

Action (4,id) → shift 5

The item F→.E action (4,c)→ shift 4

The item goto (I_4,F) → I_3, so goto [4,F] = 3

The item goto (I_4,T) → I_2, so goto [4,F] = 2

The item goto (I_4,E) → I_8, so goto [4,F] = 8

6. **Consider I_5 items:**

F→id. Is the reduced item, so take FOLLOW (F).

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

F→id is rule no.6 so reduce 6

Action (5,+) = reduce 6

Action (5,*) = reduce 6

Action (5,)) = reduce 6

Action (5,)) = reduce 6

Action (5,$) = reduce 6

7. Consider I_6 items:

goto (I_6,T) = I_9, then goto [6,T] = 9

goto (I_6,F) = I_3, then goto [6,F] = 3

goto (I_6,C) = I_4, then goto [6,C] = 4

goto (I_6,id) = I_5, then goto [6,id] = 5

8. Consider I_7 items:

1. goto (I_7,F) = I_{10}, then goto [7,F] = 10
2. goto (I_7,C) = I_4, then action [7,C] = shift 4
3. goto (I_7,id) = I_5, then goto [7,id] = shift 5

9. Consider I_8 items:

1. goto (I_8,)) = I_{11}, then action [8,)] = shift 11
2. goto (I_8,+) = I_6, then action [8,+] = shift 6

10. Consider I_9 items:

1. E→E+T. is the reduced item, so take FOLLOW (E).

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

E→E+T is the production no.1., so

Action [9,+] = reduce 1

Action [9,)] = reduce 1

Action [9,$] = reduce 1

2. goto [I_5,*] = I_7, then acgtion [9,*] = shift 7.

11. Consider I_{10} items:

1. T→T*F. is the reduced item, so take

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

T→T*F is production no.3., so

Action [10,+] = reduce 3

Action [10,*] = reduce 3

Action [10,)] = reduce 3

Action [10,$] = reduce 3

12. Consider I_{11} items:

1. F→(E). is the reduced item, so take

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

F→(E) is production no.5., so

Action [11,+] = reduce 5

Action [11,*] = reduce 5

Action [11,)] = reduce 5

Action [11,$] = reduce 5

VI MOVES OF LR PARSER ON id*id+id:

	STACK	INPUT	ACTION
1.	0	id*id+id$	shift by S5
2.	0id5	*id+id$	sec 5 on *
			reduce by F→id
			If A→β
			Pop 2*| β| symbols.
			=2*1=2 symbols.
			Pop 2 symbols off the stack
			State 0 is then exposed on F.

			Since goto of state 0 on F is 3, F and 3 are pushed onto the stack
3.	0F3	*id+id$	reduce by T $\rightarrow$F pop 2 symbols push T. Since goto of state 0 on T is 2, T and 2, T and 2 are pushed onto the stack.
4.	0T2	*id+id$	shift by S7
5.	0T2*7	id+id$	shift by S5
6.	0T2*7id5	+id$	reduce by r6 i.e. F $\rightarrow$id Pop 2 symbols, Append F, Secn 7 on F, it is 10
7.	0T2*7F10	+id$	reduce by r3, i.e., T $\rightarrow$T*F Pop 6 symbols, push T Sec 0 on T, it is 2 Push 2 on stack.
8.	0T2	+id$	reduce by r2, i.e., E $\rightarrow$T Pop two symbols, Push E See 0 on E. It 10 1 Push 1 on stack
9.	0E1	+id$	shift by S6.
10.	0E1+6	id$	shift by S5
11.	0E1+6id5	$	reduce by r6 i.e.,

		on F	$F \rightarrow id$ Pop 2 symbols, push F, see 6 It is 3, push 3
12.	0E1+6F3	\$	reduce by r4, i.e., $T \rightarrow F$ Pop2 symbols, Push T, see 6 on T It is 9, push 9.
13.	0E1+6T9	\$	reduce by r1, i.e., $E \rightarrow E+T$ Pop 6 symbols, push E See 0 on E, it is 1 Push 1.
14.	0E1	\$	Accept

Procedure for Step-V

The parsing algorithm used for all LR methods uses a stack that contains alternatively state numbers and symbols from the grammar and a list of input terminal symbols terminated by \$.

For example:

AAbBcCdDeEf/uvwxyz\$

Where, a. . .. f are state numbers

A . . .. E are grammar symbols (either terminal or non-terminals)

u . . . z are the terminal symbols of the text still to be parsed.

The parsing algorithm starts in state I_0 with the configuration –

0 / whole program upto $.

Repeatedly apply the following rules until either a syntactic error is found or the parse is complete.

(i) If action [f,4] = S_i then transform

aAbBcCdDeEf / uvwxyz$

to

aAbBcCdDeEfui / vwxyz$

This is called a SHIFT transition

(ii) If action [f,4] = #P and production # P is of length 3, say, then it will be of the form P → CDE where CDE exactly matches the top three symbols on the stack, and P is some non-terminal, then assuming goto [C,P] = g

aAbBcCdDEfui / vwxyz$

will transform to

aAbBcPg / vwxyz$

The symbols in the stack corresponding to the right hand side of the production have been replaced by the subject of the production and a new state chosen using the goto table. This is called a REDUCE transition.

(iii) If action [f,u] = accept. Parsing is completed

(iv) If action [f,u] = - then the text parsed is syntactically in-correct.

Canonical LR(O) collection for a grammar can be constructed by augmented grammar and two functions, closure and goto.

The closure operation:

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

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

5. CANONICAL LR PARSING*:*

Example:

$S \rightarrow CC$

$C \rightarrow CC/d.$

1. Number the grammar productions:

1. $S \rightarrow CC$
2. $C \rightarrow CC$
3. $C \rightarrow d$

2. The Augmented grammar is:

$\mathbf{S^I \rightarrow S}$

$S \rightarrow CC$

$C \rightarrow CC$

$C \rightarrow d.$

Constructing the sets of LR(1) items:

We begin with:

$S^I \rightarrow .S,\$$ begin with look-a-head (LAH) as \$.

We match the item [$S^I \rightarrow .S,\$$] with the term [$A \rightarrow \alpha.B\beta,a$]

In the procedure closure, i.e.,

$\mathbf{A = S^I}$

$\alpha = \in$

$\mathbf{B = S}$

$\beta = \in$

$a = \$$

Function closure tells us to add [$B \rightarrow .r,b$] for each production $B \rightarrow r$ and terminal b in FIRST (βa). Now $\beta \rightarrow r$ must be $S \rightarrow CC$, and since β is $\in$ and a is \$, b may only be \$. Thus,

$S \rightarrow .CC,\$$

We continue to compute the closure by adding all items [$C \rightarrow .r,b$] for b in FIRST [C$] i.e., matching [$S \rightarrow .CC,\$$] against [$A \rightarrow \alpha.B\beta,a$] we have, A=S, $\alpha=\in$, B=C and a=$. FIRST (C$) = FIRST ©

FIRST© = {c,d}

We add items:

$C \rightarrow .cC,C$

$C \rightarrow cC,d$

$C \rightarrow .d,c$

$C \rightarrow .d,d$

None of the new items have a non-terminal immediately to the right of the dot, so we have completed our first set of LR(1) items. The initial I_0 items are:

I_0 : $S^I \rightarrow .S,\$$

$S \rightarrow .CC,\$$

$C \rightarrow .CC,c/d$

$C \rightarrow .d.c/d$

Now we start computing goto (I_0,X) for various non-terminals i.e.,

Goto (I_0,S):

I_1 : $S^I \rightarrow S.,\$$ $\rightarrow$ reduced item.

Goto (I_0,C):

I_2 : $S \rightarrow C.C,\ \$$

$C \rightarrow .cC,\$$

$C \rightarrow .d,\$$

Goto (I_0,C) :

I_2 : $C \rightarrow c.C,c/d$

$C \rightarrow .cC,c/d$

$C \rightarrow .d,c/d$

Goto (I_0,d) :

I_4 : $C \rightarrow d.,\ c/d$ $\rightarrow$ reduced item.

Goto (I_2,C) : I_5

: $S \rightarrow CC.,\$$ $\rightarrow$ reduced item.

Goto (I_2,C) : I_6

$C \rightarrow c.C,\$$

$C \rightarrow .cC,\$$

$C \rightarrow .d,\$$

Goto (I_2,d) : I_7

$C \rightarrow d.,\$$ $\rightarrow$ reduced item.

Goto (I_3,C) : I_8

$C \rightarrow cC.,c/d$ $\rightarrow$ reduced item.

Goto (I_3,C) : I_3

$C \rightarrow c.C,\ c/d$

$C \rightarrow .cC,c/d$

$C \rightarrow .d,c/d$

Goto (I_3,d) : I_4

$C \rightarrow d.,c/d.$ $\rightarrow$ reduced item.

Goto (I_6,C) : I_9

$C \rightarrow cC.,\$$ $\rightarrow$ reduced item.

Goto (I_6,C) : I_6

$C \rightarrow c.C,\$$

$C \rightarrow ,cC,\$$

$C \rightarrow .d,\$$

Goto (I_6,d) : I_7

$C \rightarrow d.,\$$ $\rightarrow$ reduced item.

All are completely reduced. So now we construct the canonical LR(1) parsing table –

Here there is no neet to find FOLLOW ( ) set, as we have already taken look-a-head for each set of productions while constructing the states.

Constructing LR(1) Parsing table:

	Action			goto	
State	C	D	$	S	C
I_0	S3	S4		1	2
1			Accept		
2	S6	S7			5
3	S3	S4			8
4	R3	R3			
5			R1		
6	S6	S7			9
7			R3		
8	R2	R2			
9			R2		

1. **Consider I_0 items:**

The item S→.S.$ gives rise to goto [I_0,S] = I_1 so goto [0,s] = 1.

The item S→.CC, $ gives rise to goto [I_0,C] = I_2 so goto [0,C] = 2.

The item C→.cC, c/d gives rise to goto [I_0,C] = I_3 so goto [0,C] = shift 3

The item C→.d, c/d gives rise to goto [I_0,d] = I_4 so goto [0,d] = shift 4

2. Consider I_0 items:

The item S^I→S.,$ is in I_1, then set action [1,$] = accept

3. Consider I_2 items:

The item S→C.C,$ gives rise to goto [I_2,C] = I_5. so goto [2,C] = 5

The item C→.cC, $ gives rise to goto [I_2,C] = I_6. so action [0,C] = shift The item C→.d,$ gives rise to goto [I_2,d] = I_7. so action [2,d] = shift 7

4. Consider I_3 items:

The item C→.cC, c/d gives rise to goto [I_3,C] = I_8. so goto [3,C] = 8

The item C→.cC, c/d gives rise to goto [I_3,C] = I_3. so action [3,C] = shift 3.

The item C→.d, c/d gives rise to goto [I_3,d] = I_4. so action [3,d] = shift 4.

5. Consider I_4 items:

The item C→.d, c/d is the reduced item, it is in I_4 so set action [4,c/d] to reduce c→d. (production rule no.3)

6. Consider I_5 items:

The item S→CC.,$ is the reduced item, it is in I_5 so set action [5,$] to S→CC (production rule no.1)

7. Consider I_6 items:

The item C→c.C,$ gives rise to goto [I_6 ,C] = I_9. so goto [6,C] = 9

The item C→.cC,$ gives rise to goto [I_6 ,C] = I_6. so action [6,C] = shift 6

The item C→.d,$ gives rise to goto [I_6 ,d] = I_7. so action [6,d] = shift 7

8. Consider I_7 items:

The item C→d., $ is the reduced item, it is in I_7.

So set action [7,$] to reduce C→d (production no.3)

9. Consider I_8 items:

The item C→CC.c/d in the reduced item, It is in Is, so set action[8,c/d] to reduce C→cd (production rale no .2)

10. Consider I9 items:

The item C →cC, $ is the reduced item, It is in I9, so set action [9,$] to reduce C→cC (Production rale no.2)

If the Parsing action table has no multiply –defined entries, then the given grammar is called as LR(1) grammar

6.1 LALR PARSING:

Example:

1. Construct C={I0,I1,……….,In} The collection of sets of LR(1) items

2. For each core present among the set of LR (1) items, find all sets having that core, and replace there sets by their Union# (clus them into a single term)

I_0 →same as previous

I_1 → "

I_2 → "

I_{36} – Clubbing item I3 and I6 into one I36 item.

C →cC,c/d/\$

C→cC,c/d/\$

C→d,c/d/\$

I_5 →some as previous

I_{47} →C→d,c/d/\$

I_{89} →C→cC, c/d/\$

LALR Parsing table construction:

State	Action			Goto	
	c	d	\$	S	C
I_o	S_{36}	S_{47}		1	2
1			Accept		
2	S_{36}	S_{47}			5
36	S_{36}	S_{47}			89
47	r_3	r_3			
5			r_1		
89	r_2	r_2	r_2		