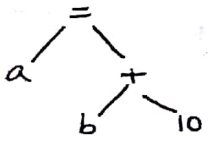
UNIT-II Syntax Analysis - parsing

- Definition - role of parsers - Top down parsing - Bottom up parsing
- Left recursion - left factoring - Handle pruning - Shift reduce parsing
- LEADING - TRAILING - Operator precendence parsing
- FIRST - FOLLOW - Predictive parsing - Recursive descent parsing
- LR Parsing - LR(0) items - SLR parsing
- Canonical LR parsing - LALR parsing

Definition of parser :-

A parsing or syntax analysis is a process which takes the input string w and produces either a parse tree (syntactic structure) or generates the syntatic errors.

For example: a = b+10;

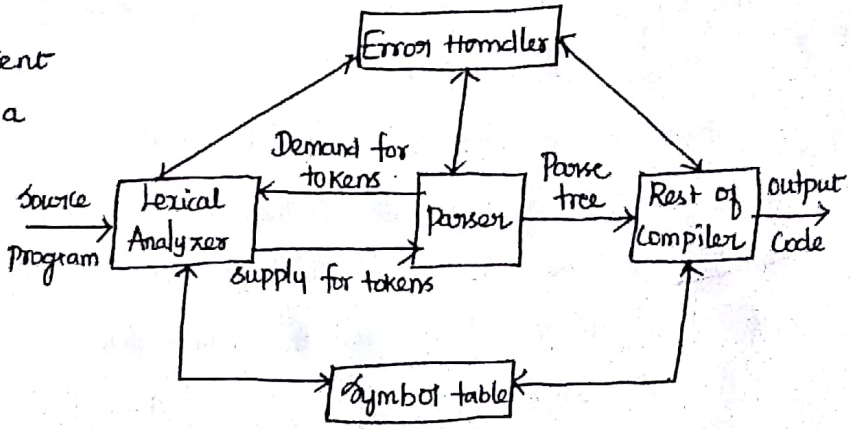


Role of parser :-

⇒ The parser collects sufficient number of tokens and builds a parse tree.

⇒ Parser smartly finds the syntactical errors.

⇒ parser should recover from commonly occuring errors.



Separating out lexical and syntax analyzer has two advantages,

1. It accelerates the process of compilation.
2. Errors in the source input can be identified precisely.

Parsing technique: Principle

1. The parser scans the input string from left to right and identifies that the derivation is leftmost or rightmost.
2. Parser makes use of production rules for choosing appropriate derivation. Finally a parse tree is constructed.

Top-down and Bottom-up parsing.

Top-down parser	Bottom-up parser.
1. Parse tree can be built from root to leaves.	Parse tree is built from leaves to root.
2. This is simple to implement	this is complex to complement.
3. This is less efficient, various problems that occur during this is 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.

Left recursion:

The left recursive grammar is a grammar of the form,

$$A \Rightarrow A\alpha$$

Where A - non-terminal, α - input string

⇒ Here, expansion of A causes further expansion of A only and due to generation of $A, A\alpha, A\alpha\alpha, A\alpha\alpha\alpha, \ldots$ the input pointer will not be advanced.

⇒ To eliminate this, we need to modify the grammar with the production rule,

$A \to A\alpha$
$A \to \beta$. Rewriting this, $\boxed{\begin{array}{l} A \to \beta A' \\ A' \to \alpha A' \mid \epsilon \end{array}}$

Example:

$$E \to E+T \mid T$$

Here $A = E$, $\alpha = +T$, $\beta = T$

Then the rule becomes, $E \to TE'$

$$E' \to +TE' \mid \epsilon$$

Left_factoring ⇒ Two alternatives is used to expand the non-terminal.

In general if $A \to \alpha\beta_1 | \alpha\beta_2$ then it can be left factored as,

$$A \to \alpha A'$$
$$A' \to \beta_1 | \beta_2$$

Example: consider the following grammar,

$S \to iEtS | iEtSeS | a$

$E \to b$

The left factored grammar becomes,

$S \to iEtSS' | a$

$S' \to eS | \epsilon$

$E \to b$

Problem 1: Eliminate left recursion for the grammar,

$E \to E+T | T$, $T \to T*F | F$, $F \to (E) | id$.

Solution:

Rule to eliminate left recursion is $A \to A\alpha | \beta \Rightarrow A \to \beta A'$, $A' \to \alpha A' | \epsilon$.

(i) $E \to E+T | T \Rightarrow E \to TE'$, $E' \to +TE' | \epsilon$

(ii) $T \to T*F | F \Rightarrow T \to FT'$, $T' \to *FT' | \epsilon$

After eliminating left recursion, the grammar becomes,

$E \to TE'$

$E' \to +TE' | \epsilon$

$T \to FT'$

$T' \to *FT' | \epsilon$

$F \to (E) | id$.

Do left factoring in the following grammar,

$A \rightarrow aAB \mid aA \mid a$

$B \rightarrow bB \mid b$

Solution:

left factoring rule is $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \cdots \Rightarrow$ $A \rightarrow \alpha A'$, $A' \rightarrow \beta_1 \mid \beta_2$

(i) $A \rightarrow aAB \mid aA \mid a$ (α = a, β_1 = AB, β_2 = A, β_3 = a)

$\Rightarrow$ $A \rightarrow aA'$, $A' \rightarrow AB \mid A \mid \epsilon$

(ii) $B \rightarrow bB \mid b$ (α = b, β_1 = B, β_2 = b)

$\Rightarrow$ $B \rightarrow bB'$, $B' \rightarrow B \mid \epsilon$

After left factoring, the grammar becomes,

$A \rightarrow aA'$

$A' \rightarrow AB \mid A \mid \epsilon$

$B \rightarrow bB'$

$B' \rightarrow B \mid \epsilon$

Bottom-up parsing:-

Basic steps in bottom-up parsing are,

1) Reduction of input string to start symbol.
2) The sentential forms that are produced in the reduction process should trace out rightmost derivation in reverse.

Handle pruning:

Handle is a string of substring that matches the right side of production and reduce such string by a non-terminal on left hand side production.

Definition: Handle of right sentential form γ is a production $A \rightarrow \beta$ and a position of γ where the string β may be found and replaced by A to produce the previous right sentential form in rightmost derivation of γ.

For example,

consider the grammar, $E \rightarrow E+E$

$E \rightarrow id$

Consider the string id+id+id and the rightmost derivation is

$$E \underset{rm}{\Rightarrow} E+E$$
$$E \underset{rm}{\Rightarrow} E+E+E$$
$$E \underset{rm}{\Rightarrow} E+E+id$$
$$E \underset{rm}{\Rightarrow} E+id+id$$
$$E \underset{rm}{\Rightarrow} id+id+id$$

Right Sentential form	Handle	production
id+id+id	id	$E \rightarrow id$
E+id+id	id	$E \rightarrow id$
E+E+id	id	$E \rightarrow id$
E+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.

Shift - Reduce parser:

A shift-reduce parser requires following data structure

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

The parser performs following basic operations.

1. Shift => Moving of the symbols from input buffer onto the stack.
2. Reduce => R.H.S of rule is popped off and L.H.S is pushed in.
3. Accept => Stack contains start symbol only and input buffer is empty at the same time that action is called accept.
4. Error => A situation in which parser cannot either shift or reduce the symbols, it cannot even perform the accept action is called as error.

Problem: Consider the following grammar,

$S \rightarrow (L) | a$

$L \rightarrow L,S | S$

Parse the input string $(a,(a,a))$ using shift-reduce parser.

Solution:

Stack	Input buffer	Parsing action.
$	(a,(a,a))$	Shift
$(	a,(a,a))$	Shift
$(a	,(a,a))$	Reduce S→a
$(S	,(a,a))$	Reduce L→S
$(L	,(a,a))$	Shift
$(L,	(a,a))$	Shift
$(L,(	a,a))$	Shift
$(L,(a	,a))$	Reduce S→a
$(L,(S	,a))$	Reduce L→S
$(L,(L	,a))$	Shift
$(L,(L,	a))$	Shift
$(L,(L,a	))$	Reduce S→a
$(L,(L,S	))$	Reduce L→L,S
$(L,(L	))$	Shift
$(L,(L)	)$	Reduce S→(L)
$(L,S	)$	Reduce L→L,S
$(L	)$	Shift
$(L)	$	Reduce S→(L)
$S	$	Accept.

Operator precedence parser:

A Grammar G is said to be operator precedence if it posses following properties.

1. No production on the right side is ε.
2. There should not be any production rule possessing two adjacent Non-terminals at right hand side.

Consider the grammar.

$$\left.\begin{array}{l} E \to EAE \mid (E) \mid -E \mid id \\ A \to + \mid - \mid * \mid / \mid \wedge \end{array}\right\} \Rightarrow \begin{array}{l} E \to E+E \mid E-E \mid E*E \mid E/E \mid E \wedge E \\ E \to (E) \mid -E \mid id \end{array}$$

Precedence relations

$P \lessdot \cdot q \Rightarrow$ P gives more precedence than q

$P \doteq q \Rightarrow$ P has same precedence as q

$P \cdot\!> q \Rightarrow$ P takes precedence over q.

Consider the string id + id * id

precedence relation table for id+id*id $\quad \$ \lessdot \cdot id \cdot\!> + \lessdot \cdot id \cdot\!> * \lessdot \cdot id \cdot\!> \$$

	id	+	*	$
id		$\cdot\!>$	$\cdot\!>$	$\cdot\!>$
+	$\lessdot \cdot$	$\cdot\!>$	$\lessdot \cdot$	$\cdot\!>$
*	$\lessdot \cdot$	$\cdot\!>$	$\cdot\!>$	$\cdot\!>$
$	$\lessdot \cdot$	$\lessdot \cdot$	$\lessdot \cdot$	

Steps:

1) Scan the input from left to right until first $\cdot\!>$ is encountered.
2) Scan backwards over $\doteq$ until $\lessdot \cdot$ is encountered
3) The handle is a string between $\lessdot \cdot$ and $\cdot\!>$.

$\$ \lessdot id \cdot\!> + \lessdot \cdot id \cdot\!> * \lessdot \cdot id \cdot\!> \$$

$E + \lessdot \cdot id \cdot\!> * \lessdot \cdot id \cdot\!> \$$

$E + E * \lessdot \cdot id \cdot\!> \$$

$E + E * E$

$+ *$

$\$ \lessdot \cdot + \lessdot \cdot * \cdot\!> \$$

$\$ \lessdot \cdot + \cdot\!> \$$

$\$\$$

Advantage => Simple to implement

Disadvantage => Operator like minus has two different precedence (unary and binary). It is hard to handle.

=> Applicable only small class of grammars.

Prbm: Construct operator precedence parser and then parse the string:
$S \to (L) | a$, $L \to L,S | S$ string (a, (a, a))

Solution:

Operator precedence table.

	a	(	)	,	$
a		$\gtrdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$
(	$\lessdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$
)	$\lessdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$
,	$\lessdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$
$	$\lessdot$	$\lessdot$	$\lessdot$	$\lessdot$	

Step1: Insert $ symbol

$ (a, (a, a)) $

Step2: Process from left to right

$ (a, (a, a)) $
$ $\lessdot$ (a, (a, a)) $
$ $\lessdot$ ($\lessdot$ a, (a, a)) $
$ $\lessdot$ ($\lessdot$ a $\gtrdot$, (a, a)) $
$ $\lessdot$ ($\lessdot$ a $\gtrdot$, $\lessdot$ (a, a)) $
$ $\lessdot$ ($\lessdot$ a $\gtrdot$, $\lessdot$ ($\lessdot$ a, a)) $
$ $\lessdot$ ($\lessdot$ a $\gtrdot$, $\lessdot$ ($\lessdot$ a $\gtrdot$, a)) $
$ $\lessdot$ ($\lessdot$ a $\gtrdot$, $\lessdot$ ($\lessdot$ a $\gtrdot$, $\lessdot$ a)) $
$ $\lessdot$ ($\lessdot$ a $\gtrdot$, $\lessdot$ ($\lessdot$ a $\gtrdot$, $\lessdot$ a $\gtrdot$)) $
$ $\lessdot$ ($\lessdot$ a $\gtrdot$, $\lessdot$ ($\lessdot$ a $\gtrdot$, $\lessdot$ a $\gtrdot$) $\gtrdot$) $\gtrdot$ $

Step3:

$ $\lessdot$ (S, $\lessdot$ (S, $\lessdot$ a $\gtrdot$) $\gtrdot$) $\gtrdot$ $
$ $\lessdot$ (S, $\lessdot$ (S, S) $\gtrdot$) $\gtrdot$ $
$ $\lessdot$ (S, $\lessdot$ (L, S) $\gtrdot$) $\gtrdot$ $
$ $\lessdot$ (S, $\lessdot$ (L) $\gtrdot$) $\gtrdot$ $
$ $\lessdot$ (S, S) $\gtrdot$ $
$ $\lessdot$ (L, S) $\gtrdot$ $
$ $\lessdot$ (L) $\gtrdot$ $
$ $\lessdot$ (S) $\gtrdot$ $
$ $

Recursive Descent parsing

- A parser that uses collection of recursive procedures for parsing the given input string is called Recursive Descent parser.

Prob: construct a recursive descent parser for the following grammar,

$E \to E+T \mid T$, $T \to TF \mid F$, $F \to F* \mid a \mid b$.

Solution: The given grammar is,

$$\left.\begin{array}{l} E \to E+T \mid T \\ T \to TF \mid F \\ F \to F* \mid a \mid b \end{array}\right\} \Rightarrow \text{left recursive grammar.}$$

Rule to eliminate left recursion is $A \to A\alpha \mid \beta \Rightarrow \begin{array}{l} A \to \beta A' \\ A' \to \alpha A' \mid \varepsilon \end{array}$.

(i) $E \to E+T \mid T \Rightarrow \begin{array}{l} E \to TE' \\ E' \to +TE' \mid \varepsilon \end{array}$

(ii) $T \to TF \mid F \Rightarrow \begin{array}{l} T \to FT' \\ T' \to FT' \mid \varepsilon \end{array}$

(iii) $F \to F* \mid a \mid b \Rightarrow \begin{array}{l} F \to aF' \mid bF' \\ F' \to *F' \mid \varepsilon \end{array}$

To summarise, the recursive descent parser is,

$E \to TE'$

$E' \to +TE' \mid \varepsilon$

$T \to FT'$

$T' \to FT' \mid \varepsilon$

$F \to aF' \mid bF'$

$F' \to *F' \mid \varepsilon$

Advantages of recursive descent parser

⇒ Simple to build.

⇒ It can be constructed with the help of parse tree.

Limitations

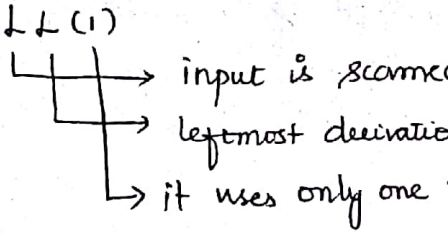
⇒ Not very efficient as compared to other parsing techniques.

⇒ There are chances that the program for RD parser may enter in an infinite loop for some input.

⇒ It can not provide good error messaging.

⇒ It is difficult to parse the string if lookahead symbol is arbitrarily long.

Predictive parsing LL(1) :-



Data structures:

(i) input buffer ⇒ to store the input tokens.

(ii) stack ⇒ used to hold left sentential form.

(iii) parsing table ⇒ Table has row for non terminal and column for terminals.

Construction of predictive LL(1) parser :-

1. Computation of FIRST and FOLLOW function.
2. Construct the predictive parsing table using FIRST and FOLLOW functions.
3. parse the i/p string with the help of predictive parsing table.

FIRST ⇒ Rules

1. If the terminal symbol a then FIRST (a) = {a}
2. If there is a rule $X \to \epsilon$ then FIRST (X) = {ε}
3. For the rule $A \to X_1 X_2 X_3 \ldots X_K$

$$FIRST(A) = \left(FIRST(X_1) \cup FIRST(X_2) \cdots \cup FIRST(X_k) \right)$$

Follow ⇒ Rules

1. For the start symbol S place $ in FOLLOW(S)
2. If there is a production $A \to \alpha B \beta$ then everything in FIRST(β) without ϵ is to be placed in follow(B).
3. If there is a production $A \to \alpha B \beta$ or $A \to \alpha B$ and FIRST(β) = ϵ then FOLLOW(A) = FOLLOW(B) or FOLLOW(B) = FOLLOW(A).

Prbm: 1) Consider the grammar,

$E \to TE'$, $E' \to +TE' | \epsilon$, $T \to FT'$, $T' \to *FT' | \epsilon$, $F \to (E) | id$

Find the FIRST and FOLLOW functions.

Solution: FIRST functions:

$E \to TE'$ FIRST(E) = FIRST(T)

$T \to FT'$ FIRST(T) = FIRST(F)

$F \to (E) | id$ FIRST(F) = {(, id}

$\therefore$ FIRST(E) = FIRST(T) = FIRST(F) = {(, id}

$E' \to +TE' | \epsilon$

FIRST(E') = {+, ϵ}

$T' \to *FT' | \epsilon$

FIRST(T') = {*, ϵ}

follow functions:

From the grammar,

$F \to (E) | id$

follow(E) = {), $}

$E \to TE'$

follow(E') = follow(E)

$\therefore$ follow(E') = {), $}

Consider the grammar,

$E \to TE'$

$E' \to +TE'$

follow(T) = FIRST(E') $\cup$ follow(E')

= {+, ϵ} $\cup$ {), $}

follow(T) = {+,), $}

Consider the grammar,

$T \to FT'$, $T' \to *FT' | \epsilon$

follow(T') = follow(T)

$\therefore$ follow(T') = {+,), $}

Consider the grammar,

$T \to FT'$

$T' \to *FT'$

follow(F) = FIRST(T') $\cup$ follow(T)

= {*, ϵ} $\cup$ {+,), $}

= {*, +,), $}

To Summarize the computation, Parse table

Symbols	FIRST	Follow
E	{(, id}	{), $}
E'	{+, ε}	{), $}
T	{(, id}	{+,), $}
T'	{*, ε}	{+,), $}
F	{(, id}	{+, *,), $}

	id	+	*	(	)	$
E	E→TE'	Error	Error	E→TE'	Error	Error
E'	Error	E→+TE'	Error	E'→ε	Error	E'→ε
T	E→FT'	Error	Error	T'→FT'	Error	Error
T'	Error	T'→ε	T'→*FT'	Error	T'→ε	T'→ε
F	F→id	Error	Error	F→(E)	Error	Error

Example: Consider the i/p string id+id*id

Stack	Input	Action
$E	id+id*id $	
$E'T	id+id*id $	E→TE'
$E'T'F	id+id*id $	E→FT'
$E'T'id	id+id*id $	F→id
$E'T'	+id*id $	
$E'	+id*id $	T'→ε
$E'T+	+id*id $	E'→+TE'
$E'T	id*id $	
$E'T'F	id*id $	T→FT'
$E'T'id	id*id $	F→id
$E'T'	*id $	
$E'T'F*	*id $	T→*FT'
$E'T'F	id $	
$E'T'id	id $	F→id
$E'T	$	
$E'	$	T'→ε
$	$	E'→ε

Pbm2: Construct the predictive parser for the following grammar

$S \rightarrow (L) | a$, $L \rightarrow L, S | S$

Construct the behavior of the parser on the sentence (a,a).

Pbm3: Construct the predictive parser for the following grammar

$S \rightarrow a | \uparrow | (T)$ $T \rightarrow T, S | S$

Show the behavior of the parser in the sentences i) (a,(a,a))

ii) ((a,a),↑,(a,),a) iii) (a,<)

Pbm4: check whether the following grammar is LL(1) grammar

$S \rightarrow iEtS | iEtSeS | a$ $E \rightarrow b$.

Also define FIRST and follow procedures.

Pbm5: Construct non-recursive predictive parsing table for the following grammar.

$E \rightarrow E$ or $E | E$ and $E |$ not $E | (E) | 0 | 1$

LR parser :- LR(k) parsing

L => left to right scanning
R => Right most derivations.
k => number of input symbols.

Properties:-

⇒ To recognize most of the programming languages written in context free grammar.
⇒ It works using non-backtracking shift reduce techniques.

Types

LR parser → SLR parser (Simple LR)
LR parser → LALR parser (Look Ahead LR)
LR parser → canonical LR parser.

SLR(1) ≤ LALR(1) ≤ LR(1)

Simple LR parsing (SLR)

⇒ Simplest form of LR parsing.

Construct for grammar
↓
construction of canonical set of items
↓
construction of SLR parsing table
↓
Parsing of input string ← input string
↓
Output

Definition of LR(0) items and related items.

1) LR(0) items for the grammar G1 is,

S → . ABC
S → A . BC
S → AB . C
S → ABC .

S → S generates S → .

2) Augmented grammar:
If the grammar G1 is having start symbol S then augmented grammar is a new grammar G1' in which S' is a new start symbol.

3) Kernel items: It is collection of items in which . are the S → . S and all the items whose dots are not the leftmost end of RHS of the rule.
Non-kernel items: The collection of all the items in which . are at the left end of RHS of the rule.

4) functions closure and goto => required to create collection of canonical set of items.

5) Viable prefix => set of prefixes in the right sentential form of production A → α. Appear on stack during shift / reduce action.

Prob: Construct the SLR(1) parsing table for,

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

Solution:

Given grammar, find canonical set of items (ie) SLR(0) items.

1) $E \to E+T$
2) $E \to T$
3) $T \to T*F$
4) $T \to F$
5) $F \to (E)$
6) $F \to id$

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

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

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

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

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

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

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

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

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

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

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

goto(I_8,))
I_{11}: $F \to (E).$

Construction of Parsing table.

State		Action					Goto		
	id	+	*	(	)	$	E	T	F
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							
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			

S ⇒ Shift Operation based on goto function.

r ⇒ reduce operation based on follow() function.

∴ follow (E) = {+,), $}

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

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

Input string : id *id +id

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
$0T2*7F10	+id$	[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

Prbm: Consider the following grammar $E \to E+T|T$, $T \to TF|F$ $F \to F*|a|b$ construct the SLR parsing table for this grammar. Also parse the input $a*b+a$.

Prbm: Check whether the following grammar is SLR(1) or not. Explain your answer with reasons.

$S \to L=R$, $S \to R$, $L \to *R$, $L \to id$, $R \to L$

Prbm: Construct LR(0) parsing table for the given grammar.

$E \to E*B$, $E \to E+B$, $E \to B$, $B \to 0$, $B \to 1$

Canonical LR parsing

⇒ The canonical set of items is the parsing technique in which a lookahead symbol is generated while constructing set of items.

⇒ Hence the collection of set of items is refered as LR(1). The value 1 in the bracket indicates that there is one lookahead symbol in the set of items.

⇒ The significance of lookahead symbols is that the parser can decide the reductions based on the k lookahead symbols.

Steps:-

i) construction of canonical set of items along with the lookahead.
ii) Building canonical LR parsing table.
iii) Parsing the input string using canonical LR parsing table.

Prbm:

Construct the LR(1) parsing table for the following grammar,

$S \to CC$

$C \to aC$

$C \to d$.

Solution:

Construct Set of LR(1) items

I_0: $S' \to .S, \$$
$S \to .CC, \$$
$C \to .aC, a/d$
$C \to .d, a/d$

I_1: goto(I_0, S)
$S' \to S., \$$

I_2: goto(I_0, C)
$S \to C.C, \$$
$C \to .aC, \$$
$C \to .d, \$$

I_3: goto(I_0, a)
$C \to a.C, a/d$
$C \to .aC, a/d$
$C \to .d, a/d$

I_4: goto(I_0, d)
$C \to d., a/d$

I_5: goto(I_2, C)
$S \to CC., \$$

I_6: goto(I_2, a)
$C \to a.C, \$$
$C \to .aC, \$$
$C \to .d, \$$

I_7: goto(I_2, d)
$C \to d., \$$

I_8: goto(I_3, C)
$C \to aC., a/d$

I_9: goto(I_6, C)
$C \to aC., \$$

Parsing table:

State	Action			Goto	
	a	d	$	S	C
0	S_3	S_4		1	2
1			Accept		
2	S_6	S_7			5
3	S_3	S_4			8
4	r_3	r_3			
5			r_1		
6	S_6	S_7			9
7			r_3		
8	r_2	r_2			
9			r_2		

Parsing the input using LR(1) parsing table: Input string = "aadd"

Stack	Input buffer	Action table	Goto table	parsing action
$0	aadd$	action [0,a] = S_3		
$0a3	add$	action [3,a] = S_3		Shift
$0a3a3	dd$	action [3,d] = S_4		Shift
$0a3a3d4	d$	action [4,d] = r_3	[3,C] = 8	Reduce by C→d
$0a3a3C8	d$	action [8,d] = r_2	[3,C] = 8	Reduce by C→aC
$0a3C8	d$	action [8,d] = r_2	[0,C] = 2	Reduce by C→aC
$0C2	d$	action [2,d] = S_7		Shift
$0C2d7	$	action [7,$] = r_3	[2,C] = 5	Reduce by C→d
$0C2C5	$	action [5,$] = r_1	[0,S] = 1	Reduce by S→CC
$0S1	$	accept		

LALR parser:

⇒ Lookahead symbol is generated for each set of items.

Steps

1) Construction of canonical set of items along with the lookahead.
2) Building LALR parsing table
3) Parsing the input string using canonical LR parsing table.

Pbm: Consider the following grammar,

$S \rightarrow L = R$

$S \rightarrow R$

$L \rightarrow *R$

$L \rightarrow id$

$R \rightarrow L$. Discuss the LALR parsing method for this grammar. List out canonical collections and also construct a parsing table.

Solution:

Construct LR(1) items.

I_0: $S' \to .S, \$$
$S \to .L=R, \$$
$S \to .R, \$$
$L \to .*R, =/\$$
$L \to .id, =/\$$
$R \to .L, \$$

I_1: goto (I_0, S)
$S' \to S. , \$$

I_2: goto (I_0, L)
$S \to L.=R, \$$
$R \to L., \$$

I_3: goto (I_0, R)
$S \to R., \$$

I_4: goto $(I_0, *)$
$L \to *.R, =/\$$
$R \to .L, =/\$$
$L \to .*R, =/\$$
$L \to .id, =/\$$

I_5: goto (I_0, id)
$L \to id., =/\$$

I_6: goto $(I_2, =)$
$S \to L=.R, \$$
$R \to .L, \$$
$L \to .*R, \$$
$L \to .id, \$$

I_7: goto (I_4, R)
$L \to *R., =/\$$

I_8: goto (I_4, L)
$R \to L., =/\$$

I_9: goto (I_6, L)
$R \to L., \$$ $S \to L=R., \$$

I_{10}: goto (I_6, L)
$R \to L., \$$

I_{11}: goto $(I_6, *)$
$L \to .*R, \$$
$R \to .L, \$$
$L \to .*R, \$$
$L \to .id, \$$

I_{12}: goto (I_6, id)
$L \to id., \$$

I_{13}: goto (I_{11}, R)
$L \to *R., \$$

From above set of items,

$I_4 = I_{11} \Rightarrow I_{411}$
$I_5 = I_{12} \Rightarrow I_{512}$
$I_7 = I_{13} \Rightarrow I_{713}$
$I_8 = I_{10} \Rightarrow I_{810}$

∴ Set of items for LALR are,

I_0: $S' \to .S, \$$
$S \to .L=R, \$$
$S \to .R, \$$
$L \to .*R, =/\$$
$L \to .id, =/\$$
$R \to .L, \$$

I_1: $S' \to S., \$$

I_2: $S \to L.=R, \$$
$R \to L., \$$
$S \to R., \$$

I_{411}: $L \to *.R, =/\$$
$R \to L., =/\$$
$L \to .*R, =/\$$
$L \to .id, =/\$$

I_{512}: $L \to id., =/\$$

I_6: $S \to L=.R, \$$
$R \to .L, \$$

LALR Parsing table is,

State	Action				Goto		
	id	*	=	$	S	L	R
0	S_{512}	S_{411}			1	2	3
1				Accept			
2			S_6	r_5			
3				r_2			
411	S_{512}	S_{411}				810	713
512			r_4	r_4			
6	S_{512}	S_{411}				810	9
713			r_3	r_3			
810			r_5	r_5			
9				r_1			