UNIT-I CD (Compiler Design)

→ Compilers principles, Techniques and Tools - Textbooks (Ravisethi, Ullmann) - Authors

Topics:-

→ Phases of compiler - 10%
→ Lexical Analysis - 5%.
→ Parsing: i) Top down parses ii) Bottom up parses - 80%.
→ Syntax Directed Translation
→ Intermediate code generation } - 5%.
→ Runtime Environment

compiler:-

* A compiler is a software which converts High Level Language into a Low Level Language.

High Level Language → (compiler) → Low Level Language

* The output produced by the compiler is called Intermediate code.
* Intermediate Language:- Not purely High Level Language and not a Machine Level Language.
* Low Level Language can also be called as Machine Language (or) Assembly Language.

Pre-processor:-

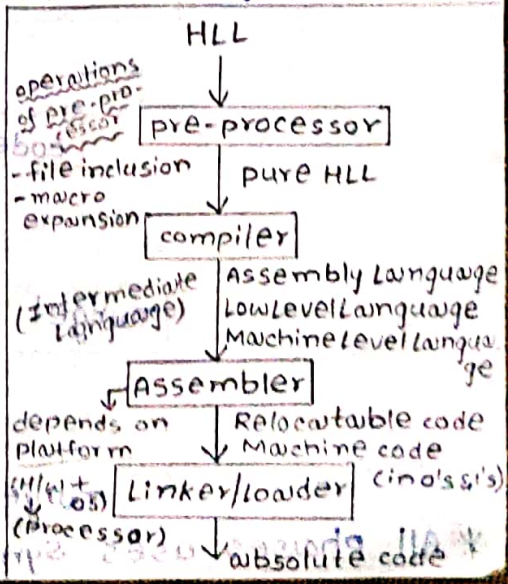
* Pre-processor replaces the shortcuts used by the user by the actual code.

Ex:-
```
# define square(x) x * x
main()
{
  int x=4, y;
  y=64/square(x);
  printf(y);
}
```

```
y=64/x*x
y=64/4*4
y=16*4
y=64
```

output:- 64

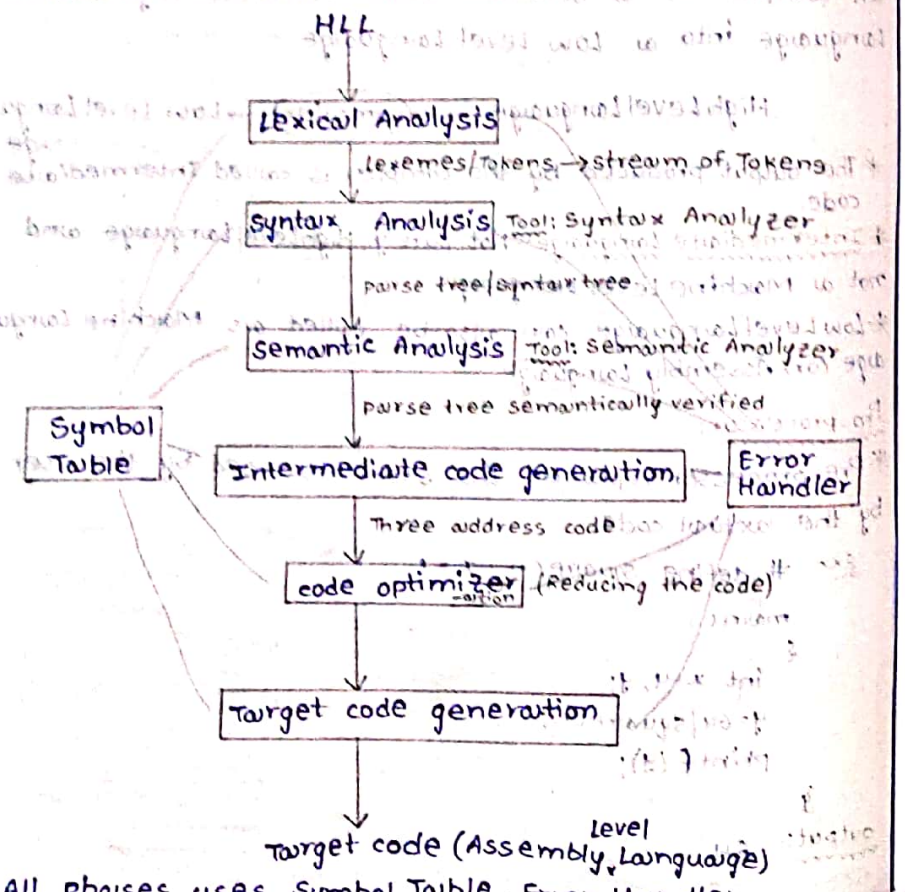
Tcc - Turbo C Compiler. It contains all the modules.



Structure of compiler:-

* Process of compiler has 2 parts
  i) Analysis (or) Front End — Tool:- LANCE
  ii) Synthesis (or) Back End (depends on platform (platform = H/W+O/S))
* Phases of compiler under Analysis
  1) Lexical Analysis
  2) Syntax Analysis
  3) Semantic Analysis and one Symbol Table, Error Handler (produce the errors)
* Phases of compiler under Synthesis
  1) Intermediate code generation
  2) code optimizer
  3) Target code generation

Phases of a compiler:-



* All phases uses Symbol Table, Error Handler

Ex:- x = a + b * c ; (HLL)

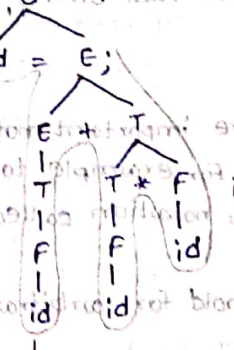
↓ Lexical Analyser → LEX

(Lexemes/Tokens) id = id + id * id ; (id = identifier)

↓ (Yacc: Yet another compiler compiler)

Syntax Analyzer (syntax Analyzer checks (or) verifies whether the Lexemes are correctly divided (or) not)

S → id = E;
E → E + T / T
T → T * F / F
F → id



id = id + id * id;

↓ (semantic Analyser verifies the parse tree generated by the Syntax Analysis is correct (or) not)

Semantic Analysis

parse tree semantically verified

↓

Intermediate code generation

t1 = b * c
t2 = a + t1
x = t2

↓

code optimizer

t1 = b * c
x = a + t1

↓

Target code generation

mul R1, R2 | a = R0
add R0, R2 | b = R1
mov R2, X | c = R2

(Target code/Assembly Level Language)

Symbol:- A symbol is an abstract entity which is not having any value. Special characters can also be called as Symbols.

Ex:- 0,1, a, b.

Alphabet:- A finite set of symbols is called alphabet. It is represented by Σ. $\Sigma = \{a, b\}$, $\Sigma = \{0, 1\}$

Ex:- $\{0, 1\}$, $\{0, \ldots, 9\}$, $\{a, \ldots, z\}$

String:- A sequence of symbols is called a string. Every string is associated with Null string, ϵ (Epsilon).

Ex:- 0010

Regular Expressions:-

* Regular Expressions are important notation for specifying Lexeme / Tokens, patterns. For example to describe a set of valid 'C' identifiers a notation called Regular Expressions is used.
* Algebraic laws that hold for arbitrary Regular Expressions r, s, T

	Description
$r+s = s+r$	+ is commutative
$r+(s+t) = (r+s)+t$	+ is associative
$r(st) = (rs)t$	concatenation is associative
$r(s+t) = rs+rt$	concatenation is distributed over +
$\epsilon r = r\epsilon = r$	ϵ is the identity for concatenation
$r^* = (r+\epsilon)^*$	ϵ is guaranteed in a closure
$(r^*)^* = r^*$	* is idempotent

1. Construct a Regular Expression for $\Sigma = \{a, b\}$ that generates all strings of length exactly 2 (if the language is finite apply union operation)

$L = \{aa, ab, ba, bb\} = aa + ab + ba + bb$
$= a(a+b) + b(a+b)$
$= (a+b)(a+b)$

2. construct a Regular Expression for $\Sigma = \{a, b\}$ that generates all strings whose length is atleast 2 $(2, 3, 4, 5, \ldots)$

$L = \{aa, ab, ba, bb, aaa, \ldots\}$

$(a+b)(a+b)(a+b)^*$

3. construct a Regular Expression that generates all strings over $\Sigma = \{a, b\}$ where the length of the string is atmost 2 $(0, 1, 2)$

$L = \{\epsilon, a, b, aa, ab, ba, bb\} = \epsilon + a + b + aa + ab + ba + bb$
$= \epsilon + a(\epsilon + a + b + b) + b(\epsilon + b + a + a)$
$= (a+b+\epsilon)(a+b+\epsilon)$

4. Even length strings over $\Sigma = \{a, b\}$

$L = \{\epsilon, aa, ab, ba, bb, aaaa, \ldots\}$

$[(a+b)(a+b)]^*$

5. Odd length strings over $\Sigma = \{a, b\}$

$L = \{\epsilon, aaa, aab, aba, abb, \ldots\}$

$[(a+b)(a+b)]^*(a+b)$

6. Length of the string should be divisible by 3.

$L = \{\epsilon, aaa, aab, aba, \ldots\}$

$[(a+b)(a+b)(a+b)]^*$

7. No. of a's in the strings formed by over $\Sigma = \{a, b\}$ must contain exactly two a's.

$L = \{aa, aabb, baba, bbaa, abba, \ldots\}$

$b^*ab^*ab^*$

8. No. of a's are atleast 2 in the string formed by over $\Sigma = \{a, b\}$

$L = \{aa, aaa, aab, aba, \ldots\}$

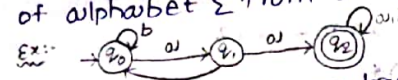
$(a+b)^*(b^*ab^*ab^*)(a+b)^*$

Finite Automata

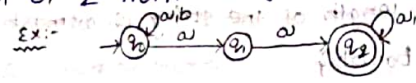
- Recognizers
- Transducers

* Recognizers produce Binary output (0 & 1 : yes / No)
* Transducers produce output.
* Recognizers – DFA, NFA, NFA-ϵ
* Transducers – Moore, Mealy

DFA:- There should be exactly one transition on each input Symbol of alphabet Σ from every state.

Ex:- 

NFA:- There can be 0,1 (or) more transitions from each input Symbol of Σ from every state.

Ex:- 

Lexical Analysis:- The tool used for Lexical Analysis is called Lexical Analyser/Scanner/Tokenizer

Role of Lexical Analysis:-

1) Lexical Analyzer functions
2) Separation of Lexical Analyser from Syntax Analysis.
3) Lexemes, Tokens and patterns
4) Lexical Errors

1) Lexical Analyzer functions:-

*LA (Lexical Analyzer) Reads the program character by character and produces a sequence (stream) of tokens.

*It removes comments in the source program.

*It removes white space characters, tab space and new line characters.

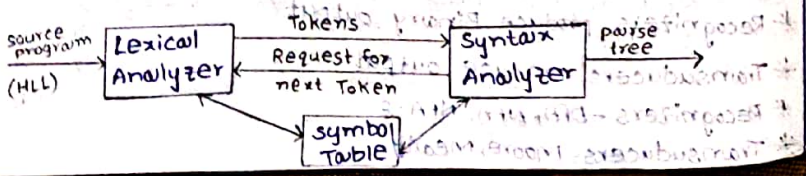
*It counts line numbers of source program.

*Generates a symbol table which stores information about identifiers.

*It provides error messages with corresponding line numbers and column numbers.

*Interaction between Lexical Analyzer and Syntax Analyzer.

*LA is also called as scanner, SA is also called as parsers



2) Separation of LA (Lexical Analyzer) from Syntax Analyzer:-

* To simplify the design of compiler design
* To increases the efficiency of a compiler.
* To increases the portability of a compiler.

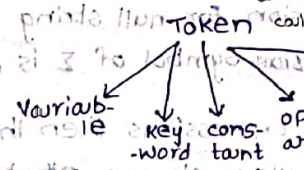
3) Lexemes, Tokens and patterns:-

Token:- It is a pair containing two components

<Token name, attribute value>

*A Token can be a variable, keyword, constant, operator, special character.

*The process of forming tokens from an input stream of characters is called Tokenization.



Lexemes:-

*It is a sequence of character

Ex:- E = M * c * * 2;

Lexemes	Token
E	<id, 1> (or) id 1
=	<=>
M	<id, 2> (or) id 2
*	<*>
c	<id, 3> (or) id 3
*	<*>
2	<2>
;	<;>

Symbol Table

1 id	- - - -
2 id	- - - -
3 id	- - - -

patterns:- It is used to define regular Expression

Ex:- RE for identifier $(a/\_)\,(a/\_/d)^*$

$\left[[a-z]\,[A-Z]\,\_\right]\left([a-z]\,[A-Z]\,\_\,[0-9]\right)^*$

Lexical Errors:- Commonly generated Lexical Errors are

1. Spelling Errors
2. Unmatched string
3. appearance of illegal characters
4. Exceeding length of an identifier

Specification of Tokens:-

* Regular Expressions are used to specify Tokens.

Regular Expressions:-

* A Regular Expression over Σ can be defined as

1. ϕ is a Regular Expression for Empty set.
2. ϵ is a Regular Expression for null string
3. 'a' is a Regular Expression Symbol of Σ (alphabet) is a Regular Expression.
4. If R and s are Regular Expressions then their union R+s, concatenation Rs, Kleen closure R^* s^* are Regular Expressions.

Meta characters/patterns of R.E:-

Metacharacter	Definition
x	x is a metacharacter matches with a character x
.	. is a metacharacter matches with any character other than new line
*	* is a metacharacter matches with 0 (or) more occurances
+	+ is a metacharacter matches with 1 (or) more occurances
^	^ is a metacharacter matches with begining of a line
$	$ is a metacharacter matches with end of a line
[abc] ([])	matches with a (or) b (or) c
|	Union
?	? is a metacharacter matches with 0 (or) 1 occurance.

Regular definition:- It is a name given to the Regular Expression and we can use that name later ($d_1 \to (a+b)a^*$, $d_2 \to r_2$; d_1, d_2 are the names given to R.E.)

Ex: 1. Regular Expression for an identifier (Variable).

$\Sigma = \{a, \ldots, z, \_, A, \ldots, Z, 0, 1, \ldots, 9\}$

letter $\to a|b|\ldots|z|\_|A|B\ldots|Z$

digit $\to 0|1|\ldots|9|$

(letter) (letter|digit)*

2. Regular Expression for numbers. numbers → int -123,456; float -123.456, -123.456E20, -123.456E+20, -123.456E-20

$\Sigma = \{$ digit $\to 0|1|2|\ldots|9|$

digits $\to$ (digit) (digit)*

number $\to$ digits (. digits)? (E[+ -]? digits)?

(or) unsigned numbers

Recognization Token:-

* Lexical Analyzer reads the source program character by character and produces the sequence of Tokens.
* Tokens are specified by a Regular Expression.
* Tokens are recognized by Transition diagram.
* A delimiter is a white space/Tab space/New line

Word Space → (delimiter) (delimiter)*

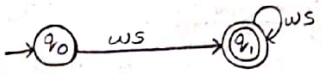
(or)

ws → White space/Tab space/Newline

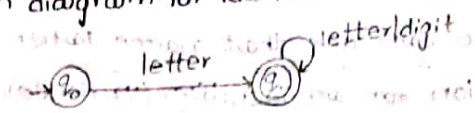
delimiter → (ws) (ws)*

Recognization:-

Transition diagram

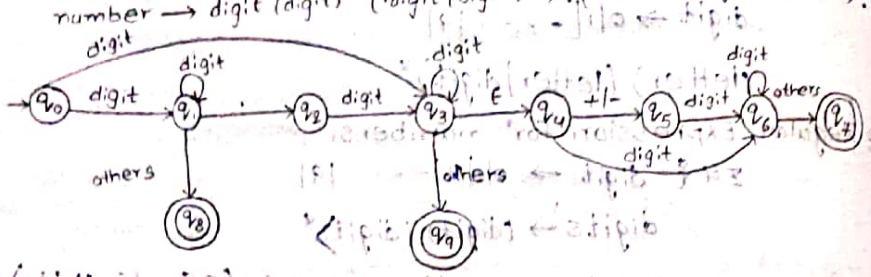


Transition diagram for identifier :-



Transition diagram for unsigned numbers:-

number → digit (digit)* (.digit (digit)*)? (E [+ -]? digit (digit)*)?



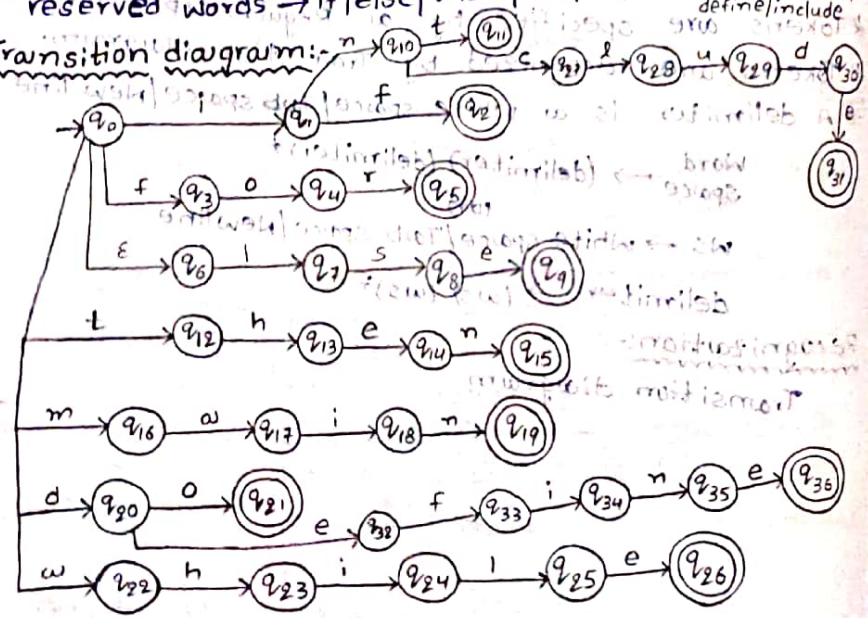
Regular Expression for reserved words (or) key words:-

If Reserved Words : if, for, else, then, ~~for~~, main, do, while, int, define, include.

Ans:

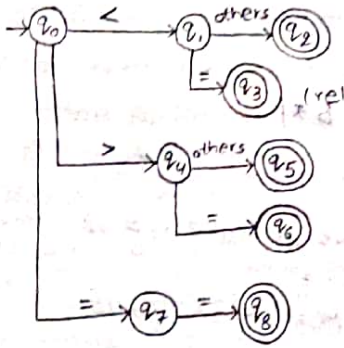
reserved words → if | else | then | for | do | main | while | int | define | include

Transition diagram:-



Regular Expression for Relational operator

relop → < | <= | > | >= | == | <>



UNIT-1

Overview of Language Processing - pre-processor, compiler, assembler, interpreter and Loader and Linker

phases of a complier - 6

i) Lexical Analysis - a) Role of LA b) LA vs Parsing c) Token d) pattern e) Lexemes f) Lexical Errors.

Regular Expressions - a) Definition of RE b) Regular definition for Language constructs - strings, sequence

Transition diagrams for recognization of tokens; identifiers, reserved word, unsigned numbers with examples.

Tokenization:-

* It is the process of converting source program into the sequence of Tokens is called Tokenization.
* A Lexical Analyzer is also called as Lexer (or) Scanner (or) Reader (or) Tokenizer
* A Tokenizer performs Tokenization

* Strings enclosed within double quotes are considered as Single Token.

Ex:- 1. printf("i = %d, &i = %x", i, &i);

3

∴ No. of tokens = 10

2. int main()
{
/* find max of a & b */
int a=20, b=30;
if (a>=b)
return (a);
else
return (b);
}

∴ No. of tokens = 32

a) 32 b) 31 c) 33 d) 40

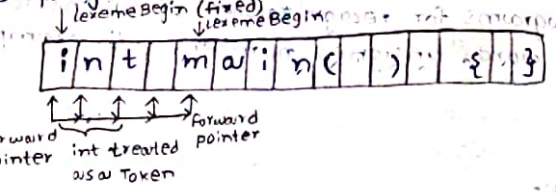
Input Buffering:-

The main role of Lexical Analyzer:-

* Lexical Analyzer reads the source program character by character and produces sequence of Tokens.
* In order to read the Tokens the lexical Analyzer uses two pointers i) Lexeme Begin pointer ii) forward pointer

Ex:- int main()
{
}

* This program is stored in secondary memory as



* For reading a single character one system call is required. so to read 100 characters then 100 system calls are required. To overcome this problem Buffering concept is used.

* Buffering means, instead of reading the single character from the hard Disk a block of characters are read into the Buffer simultaneously with the help of single system call.
* Buffering can be implemented in two ways
  1) one Buffer Scheme
  2) Two Buffer Scheme

One Buffer scheme:- It uses one Buffer

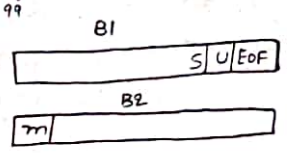
* If the size of the input string is more than the capacity of Buffer the Buffer is overwritten to store the remaining character of input string
* The solution to this problem is using two Buffer Scheme.

Two Buffer scheme:-

* It uses two Buffers.
* To determine the 1st Buffer is completly filled (or) not a special character EOF (End of File) (or) sentinal character is used and It is not part of the source program. After the 1st Buffer is completly is filled the 2nd Buffer is filled. Until the 2nd Buffer is filled the 1st Buffer is not over written

Ex:-
Ip string 500 characters
(B1, B2) Buffer size - 100 characters

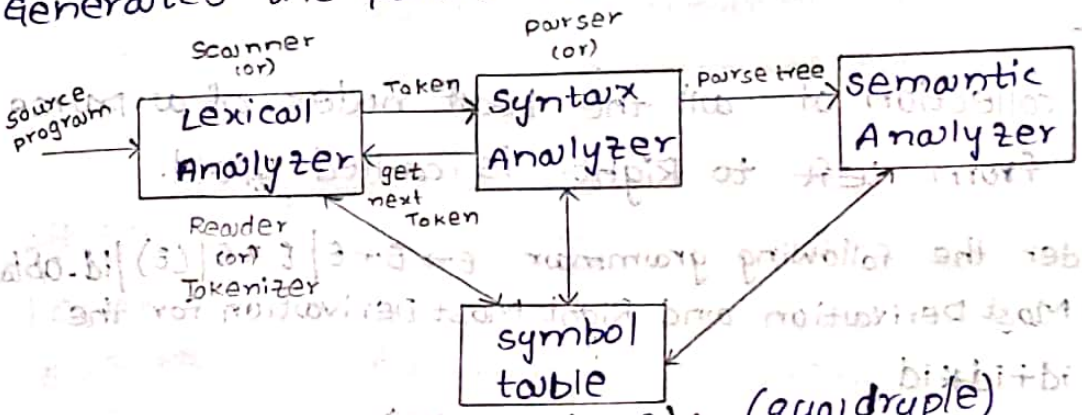
sum = sum+1;
100
99



UNIT-II SYNTAX ANALYSIS

Role of Syntax Analysis:-

* The syntax Analyzer after recieving the Tokens from Lexical Analyzer, collecting information regarding tokens from the symbol table, verifying the Syntax of tokens based on the language rules. Generates the parse tree.



context Free Grammar (CFG):- (quadruple)

*A grammar $G=(V,T,P,S)$ is a cfg where V is a set of non-terminal/variables

T is set of Terminal symbol

S is a start symbol

P is a set of productions of the form

$V \rightarrow (V \cup T)^*$

Derivation:- The process of applying a sequence of production rules to derive a string is called a Derivation.

*A Derivation can be of two types

1) Left Most Derivation (LMD)
2) Right Most Derivation (RMD)

Left Most Derivation:-

*During the derivation process, in each step the Left most production variable of RHS production is replaced first.

Right Most Derivation:-

* During the derivation process, in each step the Right most variable of RHS production is replaced first

parse tree:-

*The pictorial (or) diagramatic representation of a derivation process is called Parse tree.

Yield:-

* The collection of all the Leaf nodes of a parse tree from left to Right is called Yield.

1. consider the following grammar $E \to E+E \mid E*E \mid (E) \mid id$. Obtain Left Most Derivation and Right Most Derivation for the string id+id*id (unambiguous grammar)

Let $G=(V,T,P,S)$ be the given CFG

Where V = set of Non-terminals $V=\{E\}$

T = set of Terminals $T=\{id,+,*,(,)\}$

P = Production

P: $E \to E+E \mid E*E \mid (E) \mid id$

S = start symbol = E

Left Most Derivation (LMD):-

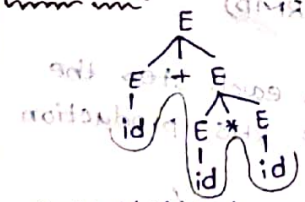
$E \Rightarrow E+E$ [$\because E \to E+E$]
$\Rightarrow id+E$ [$\because E \to id$]
$\Rightarrow id+E*E$ [$\because E \to E*E$]
$\Rightarrow id+id*E$ [$\because E \to id$]
$\Rightarrow id+id*id$ [$\because E \to id$]

Right Most Derivation (RMD):-

$E \Rightarrow E*E$ [$\because E \to E*E$]
$\Rightarrow E*id$ [$\because E \to id$]
$\Rightarrow E+E*id$ [$\because E \to E+E$]
$\Rightarrow E+id*id$ [$\because E \to id$]
$\Rightarrow id+id*id$ [$\because E \to id$]

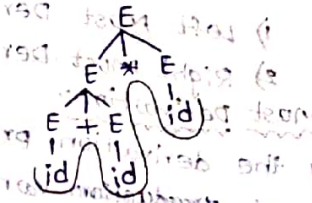
$E \to E*E \to E+E*E \to id+E*E \to id+id*E \to id+id*id$

Parse tree:-



Yield: id+id*id

Parse tree:-



Yield: id+id*id

2. consider the following grammar $S \to 0A \mid 1B \mid 0 \mid 1$
$A \to 0S \mid 1B \mid 1$
$B \to 0A \mid 1S$

construct LMD and Parse tree for the following

i) 0101 ii) 1100101

Let $G=(V,T,P,S)$ be the given CFG

Where V = set of Non-terminals $V=\{S,A,B\}$

T = set of Terminals $T=\{0,1\}$

P = production $S \to 0A \mid 1B \mid 0 \mid 1$, $A \to 0S \mid 1B \mid 1$, $B \to 0A \mid 1S$

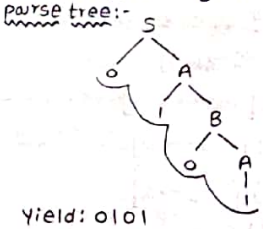
S = start symbol = S

Left Most Derivation (LMD):-

i) 0101

$S \Rightarrow 0A$ [$\because S \to 0A$]
$\Rightarrow 01B$ [$\because A \to 1B$]
$\Rightarrow 010A$ [$\because B \to 0A$]
$\Rightarrow 0101$ [$\because A \to 1$]

parse tree:-

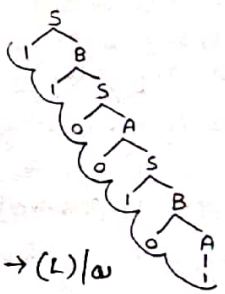


Yield: 0101

ii) 1100101

$S \Rightarrow 1B$ [$\because S \to 1B$]
$\Rightarrow 11S$ [$\because B \to 1S$]
$\Rightarrow 110A$ [$\because S \to 0A$]
$\Rightarrow 1100S$ [$\because A \to 0S$]
$\Rightarrow 11001B$ [$\because S \to 1B$]
$\Rightarrow 110010A$ [$\because B \to 0A$]
$\Rightarrow 1100101$ [$\because A \to 1$]

parse tree:-



Yield: 1100101

3. consider the following grammar $S \to (L) \mid a$
$L \to L,S \mid S$

a. What are the Terminals, Non-terminals and Start Symbol

b. Find LMD, RMD and parse trees for the following

i) (a,a) ii) (a, (a,a))

Let $G=(V,T,P,S)$ be the CFG where

a) V = set of Non-terminals $V=\{S,L\}$

T = set of Terminals $T=\{(,),a,,\}$

P = Production $S \to (L) \mid a$
$L \to L,S \mid S$

S = start symbol = S

b. i) (a, a)

LMD:-

$S \Rightarrow (L)$ [∵ $S \to (L)$]
$\Rightarrow (L,S)$ [∵ $L \to L,S$]
$\Rightarrow (S,S)$ [∵ $L \to S$]
$\Rightarrow (a,S)$ [∵ $S \to a$]
$\Rightarrow (a,a)$ [∵ $S \to a$]

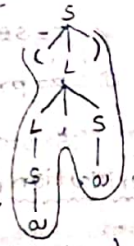
RMD:-

$S \Rightarrow (L)$ [∵ $S \to (L)$]
$\Rightarrow (L,S)$ [∵ $L \to L,S$]
$\Rightarrow (L,a)$ [∵ $S \to a$]
$\Rightarrow (S,a)$ [∵ $L \to S$]
$\Rightarrow (a,a)$ [∵ $S \to a$]

Parse tree:

Yield: (a, a)

parse tree:-



Yield: (a, a)

ii) (a, (a, a))

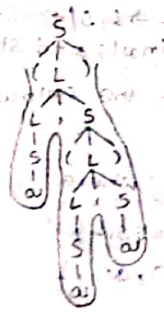
LMD:-

$S \Rightarrow (L)$ [∵ $S \to (L)$]
$\Rightarrow (L,S)$ [∵ $L \to L,S$]
$\Rightarrow (S,S)$ [∵ $L \to S$]
$\Rightarrow (a,S)$ [∵ $S \to a$]
$\Rightarrow (a,(L))$ [∵ $S \to (L)$]
$\Rightarrow (a,(L,S))$ [∵ $L \to L,S$]
$\Rightarrow (a,(S,S))$ [∵ $L \to S$]
$\Rightarrow (a,(a,S))$ [∵ $S \to a$]
$\Rightarrow (a,(a,a))$ [∵ $S \to a$]

RMD:-

$S \Rightarrow (L)$ [∵ $S \to (L)$]
$\Rightarrow (L,S)$ [∵ $L \to L,S$]
$\Rightarrow (L,(L))$ [∵ $S \to (L)$]
$\Rightarrow (L,(L,S))$ [∵ $L \to L,S$]
$\Rightarrow (L,(L,a))$ [∵ $S \to a$]
$\Rightarrow (L,(S,a))$ [∵ $L \to S$]
$\Rightarrow (L,(a,a))$ [∵ $S \to a$]
$\Rightarrow (S,(a,a))$ [∵ $L \to S$]
$\Rightarrow (a,(a,a))$ [∵ $S \to a$]

parse tree:-



Yield: (a, (a, a))

Parse tree:-

Yield: (a, (a, a))

3. Find LMD and RMD for the string aabaab using the grammar

$S \to aS / aSbS / \varepsilon$

Let $G = (V, T, P, S)$ be the given CFG where

V = set of Non-terminals $V = \{S\}$

T = set of Terminals $T = \{a, b\}$

P = production $S \to aS / aSbS / \varepsilon$

S = start symbol = S

i) aabaab

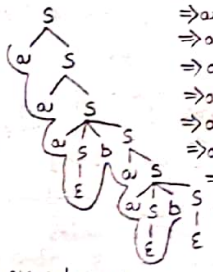
LMD:-

$S \Rightarrow aS$ [∵ $S \to aS$]
$\Rightarrow aaS$ [∵ $S \to aS$]
$\Rightarrow aaaSbS$ [∵ $S \to aSbS$]
$\Rightarrow aaa\varepsilon bS$ [∵ $S \to \varepsilon$]
$\Rightarrow aaabS$
$\Rightarrow aaabaSbS$ [∵ $S \to aSbS$]
$\Rightarrow aaabaaSbS$ [∵ $S \to aS$]
$\Rightarrow aaabaa\varepsilon bS$ [∵ $S \to \varepsilon$]
$\Rightarrow aaabaabS$
$\Rightarrow aaabaab\varepsilon$ [∵ $S \to \varepsilon$]
$\Rightarrow aaabaab$

RMD:-

$S \Rightarrow aS$ [∵ $S \to aS$]
$\Rightarrow aaSbS$ [∵ $S \to aSbS$]
$\Rightarrow aaSb\varepsilon$ [∵ $S \to \varepsilon$]
$\Rightarrow aaSb$
$\Rightarrow aaaSbSb$ [∵ $S \to aSbS$]
$\Rightarrow aaaSba\varepsilon b$ [∵ $S \to \varepsilon$]
$\Rightarrow aaaSbaab$ [∵ $S \to aS$]
$\Rightarrow aaaa\varepsilon baab$ [∵ $S \to \varepsilon$]
$\Rightarrow aaabaab$

parse tree:-

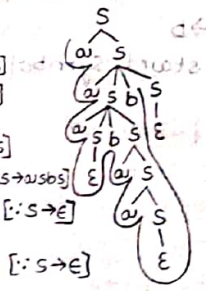


Yield: aabaab

LMD:-

$S \Rightarrow aSbS$ [∵ $S \to aSbS$]
$\Rightarrow aaSbS$ [∵ $S \to aS$]
$\Rightarrow aaa\varepsilon bS$ [∵ $S \to \varepsilon$]
$\Rightarrow aabS$
$\Rightarrow aabaSbS$ [∵ $S \to aSbS$]
$\Rightarrow aabaa\varepsilon bS$ [∵ $S \to \varepsilon$]
$\Rightarrow aabaabS$
$\Rightarrow aabaab\varepsilon$ [∵ $S \to \varepsilon$]
$\Rightarrow aabaab$

parse tree:-



Yield: aabaab

Ambiguity in CFG:-

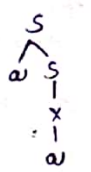
* A CFG is ambigious if there exist more than one parse tree (or) some string has more than one Left Most Derivation (or) More than one Right Most Derivation.

(Two different parse tree (or) multiple parse trees, Two different Left Most Derivations, Two different Right Most Derivations)

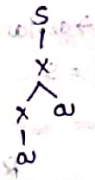
Ex:- Consider grammar G, $S \to \omega S / \omega S b / X$ And the given string is $\omega\omega$
$X \to X\omega / \omega$

String $\omega\omega$

$S \Rightarrow \omega S$
$\Rightarrow \omega X$
$\Rightarrow \omega\omega$



$S \Rightarrow X$
$\Rightarrow X\omega$
$\Rightarrow \omega\omega$



1. check whether the given grammar is ambiguous (or) not for the string ibtibtωeω using the grammar

$S \to icts / ictSeS / \omega$; $c \to b$ (ambiguous grammar)

Let $G = (V, T, P, S)$ be the given CFG where

V = set of Non-terminals $V = \{S, C\}$

T = set of Terminals $T = \{i, b, t, e, \omega\}$

P = production

$S \to icts / ictSeS / \omega$

$c \to b$

S = start symbol = S

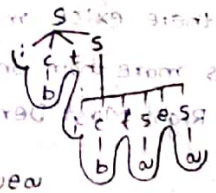
LMD:-

$S \Rightarrow icts$ [$\because S \to icts$]
$\Rightarrow ibts$ [$\because c \to b$]
$\Rightarrow ibtictses$ [$\because S \to ictses$]
$\Rightarrow ibtibtses$ [$\because c \to b$]
$\Rightarrow ibtibtωes$ [$\because S \to \omega$]
$\Rightarrow ibtibtωeω$ [$\because S \to \omega$]

RMD:-

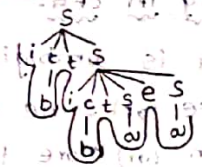
$S \Rightarrow icts$ [$\because S \to icts$]
$\Rightarrow ictictses$ [$\because S \to ictses$]
$\Rightarrow icticteω$ [$\because S \to \omega$]
$\Rightarrow icticteω$ [$\because S \to \omega$]
$\Rightarrow ictibtωeω$ [$\because c \to b$]
$\Rightarrow ibtibtωeω$ [$\because c \to b$]

parse tree:-



yield: ibtibtωeω

parse tree:-

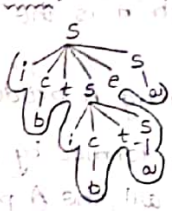


yield: ibtibtωeω

LMD:-

$S \Rightarrow icts$ [$\because S \to icts$]
$\Rightarrow ibtses$ [$\because c \to b$]
$\Rightarrow ibtictses$ [$\because S \to icts$]
$\Rightarrow ibtibtses$ [$\because c \to b$]
$\Rightarrow ibtibtωes$ [$\because S \to \omega$]
$\Rightarrow ibtibtωeω$ [$\because S \to \omega$]

parse tree:-



yield: ibtibtωeω

RMD:-

$S \Rightarrow icts$ [$\because S \to icts$]
$\Rightarrow ictseω$ [$\because S \to \omega$]
$\Rightarrow icticteω$ [$\because S \to icts$]
$\Rightarrow icticteω$ [$\because S \to \omega$]
$\Rightarrow ictibtωeω$ [$\because c \to b$]
$\Rightarrow ibtibtωeω$ [$\because c \to b$]

parse tree:-

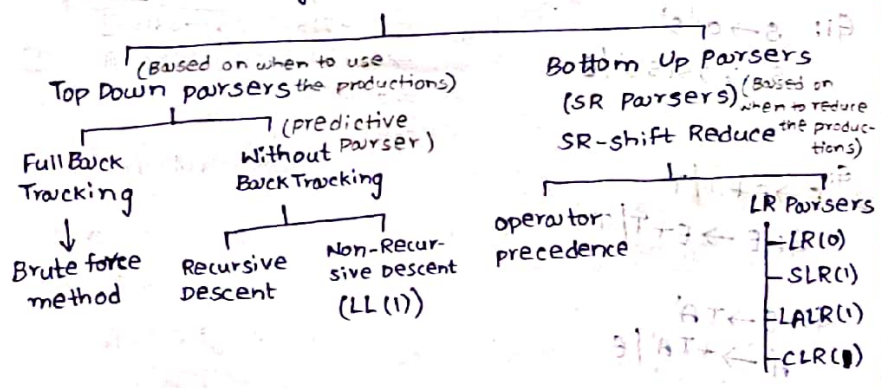
yield: ibtibtωeω

Types of parsers:-

Role of a parser:- A parser interacts with symbol table inorder to retrieve information about Tokens.

* The major role of a parser, it checks whether the Token is following the grammar of the language (or) not, if not it generates the error messages and reports to the users for corrections.

Types of parsers:-



How to Eliminate Ambiguity:

*Ambiguity in grammar can be eliminated by using
1. Elimination of Left Recursion ($A \to A\alpha|\beta$) Right recursion:- ($A \to \alpha A|\beta$)
2. Elimination of Left Factering

1. Elimination of Left Recursion:-

* Let $G=(V,T,P,S)$ be a CFG. Let $A \to A\alpha_1|A\alpha_2|----|A\alpha_r$ be the set of A productions for which A is the left most symbol of RHS

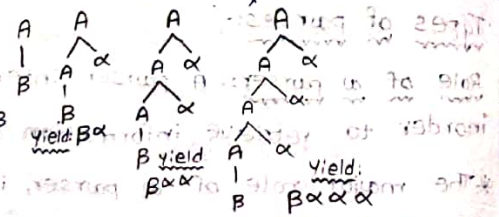
*Let $A \to \beta_1|\beta_2|----|\beta_s$ be the remaining A productions

*Let $G_1=(V\cup\{B\}, T, P_1, S)$ be the CFG formed by adding the variable B to V and replacing all the A productions by the productions: $A \to \beta_i B \mid 1 \le i \le s$ (for all) $B \to \alpha_i B \mid 1 \le i \le r$ (for all)

if $A \to A\alpha|\beta$

then $A \to \beta A'$

$A' \to \alpha A'|\varepsilon$



then $L(G)=L(G_1)$

Ex:- G: $S \to SOSIS|01$

G: $\underset{A}{S} \to \underset{A}{S}\underset{\alpha}{OSIS}|\underset{\beta}{01}$

G_1: $S \to 01S'$

$S' \to OSISS'|\varepsilon$

$S \Rightarrow 01S'$ [$\because S \to 01S'$]
$\Rightarrow 01\varepsilon$ [$\because S' \to \varepsilon$]
$\Rightarrow 01$

$S \Rightarrow 01S'$ [$\because S \to 01S'$]
$\Rightarrow 0101SS'$ [$\because S' \to OSISS'$]
$\Rightarrow 0101SIS\varepsilon$ [$\because S' \to \varepsilon$]
$\Rightarrow 0101SIS$
$\Rightarrow 0101OIOI$ [$\because S \to 01$]
$\Rightarrow 0100110101$ [$\because S \to 01$]

2. G: $E \to E+T|T$

G: $\underset{A}{E} \to \underset{A}{E}\underset{\alpha}{+}\underset{B}{T}|\underset{B}{T}$

G_1: $E \to TA'$

$A' \to +TA'|\varepsilon$

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

G: $\underset{A}{E} \to \underset{A}{E}\underset{\alpha}{+}\underset{B}{T}|\underset{B}{T}$

G_1: $E \to TE'$
$E' \to +TE'|\varepsilon$

G: $\underset{A}{T} \to \underset{A}{T}\underset{\alpha}{*}\underset{B}{F}|\underset{B}{F}$

G_1: $T \to FT'$
$T' \to *FT'|\varepsilon$

4. $S \to (L)|a$
$L \to L,S|S$

G: $\underset{A}{L} \to \underset{A}{L}\underset{\alpha}{,}\underset{B}{S}|\underset{B}{S}$

G_1: $L \to SL'$
$L' \to ,SL'|\varepsilon$

5. $expr \to expr+expr|expr*expr|id$

G: $\underset{A}{expr} \to \underset{A}{expr}\underset{\alpha_1}{+}\underset{A}{expr}|\underset{A}{expr}\underset{\alpha_2}{*}\underset{A}{expr}|\underset{B}{id}$

G_1: $expr \to id\ expr'$
$expr' \to +expr\ expr'|*expr\ expr'|\varepsilon$

2. Elimination of Left Factering:-

*A grammar contains Left Factoring if it contains productions in the form G: $A \to \alpha\beta_1|\alpha\beta_2|----|\gamma_1|\gamma_2|----$

* Top Down Parsers cannot handle the grammars with Left Factering.

*Left Factering can be eliminated by adding the following productions to grammar

G_1: $A \to \alpha A'|\gamma_1|\gamma_2|----$
$A' \to \beta_1|\beta_2|$

then $L(G)=L(G_1)$

Ex:- $A \to aAB|aA|a$
$B \to bB|b$

G: $\underset{A}{A} \to \underset{\alpha}{a}\underset{\beta}{AB}|\underset{\alpha}{a}\underset{\gamma_1}{A}|\underset{\gamma_2}{a}$

G: $\underset{A}{B} \to \underset{\alpha}{b}\underset{\beta}{B}|\underset{\gamma}{b}$

G_1: $B \to bB|b$
$B' \to B|\varepsilon$

After Eliminating Left Factering

G_1: $A \to aAA'|a$
$A' \to B|\varepsilon$

2. $S \to ictse \mid ictses \mid a$

A G: $S \to \underbrace{ict}_{\alpha}\underbrace{se}_{\beta} \mid \underbrace{ict}_{\alpha}\underbrace{ses}_{\beta_2} \mid \underbrace{a}_{\gamma}$

G1: $S \to ictses' \mid a$

$S' \to s \mid \epsilon$

$L(G) = L(G1)$

First and Follow:-

First:-

i) if $A \to a\alpha$, $\alpha \in (V \cup T)^*$ then $First(A) = \{a\}$

ii) if $A \to \epsilon$ then $First(A) = \{\epsilon\}$

iii) if $A \to Bc$ then $First(A) = First(B)$ if $First(B)$ does not contain ϵ.

if $First(B)$ contains ϵ then $First(A) = First(B) \cup First(c)$

Ex: 1. $E \to TE'$

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

$T \to FT'$

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

$F \to (E) \mid id$

$V = \{E, T, E', T', F\}$ $T = \{+, *, (,), id\}$

$First(E) = \{(, id\}$ (Parenthesis)

$First(E') = \{+, \epsilon\}$

$First(T) = \{(, id\}$ (parenthesis)

$First(T') = \{*, \epsilon\}$

$First(F) = \{(, id\}$ (parenthesis)

2. $S \to Bb/Cd$ $V = \{S, B, C\}$

$B \to aB \mid \epsilon$ $T = \{a, b, c, d\}$

$C \to cC \mid \epsilon$

$First(S) = \{a, b, c, d\}$

$First(B) = \{a, \epsilon\}$

$First(C) = \{c, \epsilon\}$

$S \to Bb$ $S \to Cd$

$\to \epsilon b$ $\to \epsilon d$

$\to b$ $\to d$

$S \to Bb/cd/b/d$

3. $S \to aABb$

$A \to c \mid \epsilon$

$B \to d \mid \epsilon$

$V = \{S, A, B\}$ $T = \{a, b, c, d\}$

$First(S) = \{a\}$

$First(A) = \{c, \epsilon\}$

$First(B) = \{d, \epsilon\}$

4. $S \to ACB \mid CbB \mid Ba$

$A \to da \mid BC$

$B \to g \mid \epsilon$

$C \to h \mid \epsilon$

$V = \{S, A, B, C\}$ $T = \{a, b, d, g, h\}$

$First(S) = \{d, g, h, \epsilon, a, b\}$

$First(A) = \{d, g, h, \epsilon\}$

$First(B) = \{g, \epsilon\}$

$First(C) = \{h, \epsilon\}$

Follow:-

i) if S is a start symbol $Follow(S) = \{\$\}$

ii) if $A \to \alpha B\beta$, $\alpha, \beta \in (V \cup T)^*$ then $Follow(B) = First(\beta)$ if $First(\beta)$ does not contain ϵ.

iii) if $A \to \alpha B$ then $Follow(B) = Follow(A)$

iv) if $A \to \alpha B\beta$, where $\beta \to \epsilon$, then $Follow(B) = Follow(A)$

Ex:- 1. $E \to TE'$

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

$T \to FT'$

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

$F \to (E) \mid id$

$V = \{E, T, E', T', F\}$ $T = \{+, *, (,), id\}$ $S = E$

Follow (E) = {$,)} First (E') = {+, ∈}

Follow (E') = {$,)} First (T') = {*, ∈}

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

Follow (T') = {+, $,)}

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

2. $S \to ABCDE$ $D \to d/\epsilon$

$A \to a/\epsilon$ $E \to e/\epsilon$

$B \to b/\epsilon$

$C \to c$

$V = \{S, A, B, C, D, E\}$ $T = \{a, b, c, d, e\}$

Follow (S) = {$} First (B) = {b, ∈}

Follow (A) = {b, c} First (c) = {c}

Follow (B) = {c} First (D) = {d, ∈}

Follow (C) = {d, e, $} First (E) = {e, ∈}

Follow (D) = {e, $}

Follow (E) = {$}

Top-Down Parsers:-

* When a parse is constructed from a root node and it expanded to leaf node then the parser is called Top-Down Parser. The left Most Derivation matches this requirement.
* In Top-Down parsing selection of proper rule is very important. This selection is based on trail and error technique.

Problems with Top-Down Parsing / Top-Down Parsers:-

1. Back Tracking / Brute force Technique - Trying every possible alternative to get a string

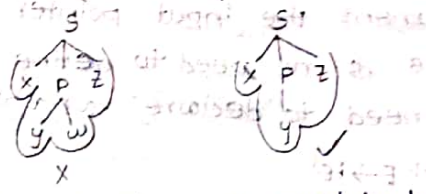
Backtracking:-

Backtracking is a technique in which for expansion of non-terminal symbol, we choose one alternative and if some mismatch occurs we have to go for another alternative this process is repeated until we get the correct input string.

Ex:- $S \to xPz$

$P \to yw/y$

$w = xyz$



Disadvantages:-

* To get the correct derivation we need to try several alternatives.
* Need to move some levels upward (Backtracking) inorder to check possibilities this increases the overhead in implementation of parsing.
* Even though it is a powerful Top-Down technique it is slower and it requires exponential time in general.
* It is not preferred for practical compilers.

2. Left Recursion-

* It is Left Recursion leads to infinite loop.

3. * Left Factering - If a grammar contains left factering it is not possible for us to take a decision whether to choose first rule (or) second

4. Ambiguity:- Ambiguous grammars are not desirable in Top-Down parsing.

Recursive Descent Parser:-

* It is a Top-Down parsing technique. It contains recursive procedures.

procedures:-

1. If the input is non-terminal call the corresponding procedure for that non-terminal.
2. If the input is a terminal then compare the terminal with input symbol and if they are same increment the input pointer
3. There is no need to define main function and no need to declare variables.

Ex:- 1. $E \to iE'$

$E' \to +iE' / \epsilon$

$V = \{E, E'\}$, $T = \{i, +\}$ $S = E$

① E()
```
{
  if(input == 'i')
  {
    input++;
    EPRIME();
  }
}
```

② EPRIME()
```
{
  if (input == '+')
  {
    input++;
    if(input == '$')
    {
      input++;
    }
    EPRIME();
  }
  else
    return;
}
```

③ main()
```
{
  E();
  if (input == '$')
  {
    print("The input string is accepted and the
           parsing is successful");
  }
}
```

2. construct a Recursive Descent parser for the following grammar $E \to TE'$

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

$T \to FT'$

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

$F \to (E) / id$

$V = \{E, E', T, T', F\}$ $T = \{+, *, (,), id\}$ $S = E$

① E()
```
{
  T();
  EPRIME();
}
```

② EPRIME()
```
{
  if (input == '+')
  {
    input++;
    T();
    EPRIME();
  }
  else
    return;
}
```

③ T()
```
{
  F();
  TPRIME();
}
```

④ TPRIME()
```
{
  if (input == '*')
  {
    input++;
    F();
    TPRIME();
  }
  else
    return;
}
```

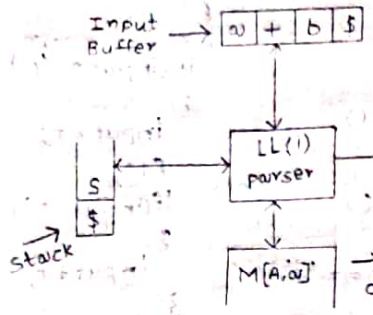
⑤ F()
```
{
  if (input == '(')
  {
    input++;
    E();
    if(input == ')')
    {
      input++;
    }
  }
  else
  {
    if (input == 'id')
    {
      input++;
    }
  }
}
```

⑥ main()
```
{
  E();
  if (input == '$')
  {
    print("The input string is accepted and the
           parsing is successful");
  }
}
```

Non-Recursive Descent Parser (or) LL(1) Parser :- (left to right, LMD)

*This is a Top-Down parser. It is of non-recursive type. In this a table is built. For LL(1), The 1st L means the input is scanned from Left to Right, The 2nd L means it uses Left Most Derivation to derive the input string, and 1 means it uses only one input symbol and 1 input symbol is also called (look ahead) to predict the parsing process.

Block diagram of LL(1) parser/model of LL(1) Parser:-



* LL(1) Parser uses input buffer to store the input Tokens.
* The stack is used to hold the left sential form. The symbols in RHS of a rule are pushed on the stack in the reverse that is from Right to Left. The use of stack makes the algorithm Non-Recursive.
* The parsing table is basically a 2 dimensional array rows specify Non-terminals and columns specify terminals. The table is represented as $M[A, a]$ where A = Non-terminal, a = Terminal

procedure to construct LL(1) Parser:-

1) Eliminate Left Recursion if any in the grammar
2) Eliminate Left Factering if any in the grammar
3) calculate First and Follow
4) construction of parsing table
5) check whether the input string is accepted by the parser (or) not.

Ex:- construct Non-Recursive Descent Parser for the following grammar.

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

Let $G=(V,T,P,S)$ be the given grammar where

$V=\{E,T,F\}$, $T=\{+,*,(,),id\}$, $S=E$

$P= E \to E+T/T$; $T \to T*F/F$; $F \to (E)/id$

Step-1:- Elimination of Left Recursion if any in G

Consider the production, $E \to E+T/T$, where Left hand side variable and RHS Left most variable are same.

∴ Left Recursion is there in the grammar.

To eliminate the Left Recursion, introduce the new variable E' and the productions are replaced with

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

$A \to A\alpha/\beta$ ⇓ $A \to \beta A'$, $A' \to \alpha A'/\epsilon$

consider the production, $T \to T*F/F$ is replaced with

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

After eliminating Left recursion the grammar is

G_1: $E \to TE'$
$E' \to +TE'/\epsilon$
$T \to FT'$
$T' \to *FT'/\epsilon$
$F \to (E)/id$

Step-2:- Eliminate the Left Factering if any in G_1

∴ There is no Left factering in G_1.

Step-3:- calculation of First and Follow

$V_1=\{E, E', T, T', F\}$ $T_1=\{+, *, (,), id\}$

First(E) = { (, id } Follow(E) = { $,) }

First(E') = { +, ε } Follow(E') = { $,) }

First(T) = { (, id } Follow(T) = { +, $,) }

First(T') = { *, ε } Follow(T') = { +, $,) }

First(F) = { (, id } Follow(F) = { *, +, $,) }

step-ii:- Constructing of parsing table

1. if the production is of the form $A \to \alpha$, $\alpha \in (V \cup T)^*$ then if First(α) contains a terminal 'a' then add $A \to \alpha$ to $M[A, a]$
2. if the production is of the form $A \to \varepsilon$ (or) First(α) contains ε then if Follow(A) = b then add $A \to \varepsilon$ to $M[A, b]$
3. The remaining all the entries of the parsing table are filled with errors.

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

1) $E \to TE'$

First(α) = { (, id }

add $E \to TE'$ to $M[E, (]$

$M[E, id]$

2) $E' \to +TE' / \varepsilon$

a) $E' \to +TE'$

First($+TE'$) = { + }

add $E' \to +TE'$ to $M[E, +]$

b) $E' \to \varepsilon$

Follow(E') = { $,) }

add $E' \to \varepsilon$ to $M[E',)]$

$M[E', \$]$

3) $T \to FT'$

First(FT') = { (, id }

add $T \to FT'$ to $M[T, (]$

$M[T, id]$

4) $T' \to *FT' / \varepsilon$

a) $T' \to *FT'$

First($*FT'$) = { * }

add $T' \to *FT'$ to $M[T', *]$

b) $T' \to \varepsilon$

Follow(T') = { +, $,) }

add $T' \to \varepsilon$ to $M[T', +]$

$M[T', \$]$

$M[T',)]$

5) $F \to (E) | id$

a) $F \to (E)$

First((E)) = { (}

add $F \to (E)$ to $M[F, (]$

b) $F \to id$

First(id) = { id }

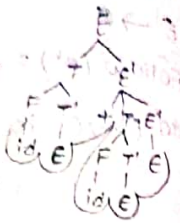
add $F \to id$ to $M[F, id]$

Step-5:- check whether the input string is accepted (or) not by the parser

if id+id$

Stack	I/p string	Action
$E	id+id$	E→TE'
$E'T	id+id$	T→FT'
$E'T'F	id+id$	F→id
$E'T'id	id+id$	pop
$E'T'	+id$	T'→ε
$E'	+id$	E'→+TE'
$E'T+	+id$	pop
$E'T	id$	T→FT'
$E'F	id$	F→id
$E'id	id$	pop
$E'	$	E'→ε
$	$	accept

$ E T' E	id $	F→id
$ E T' id	id $	pop
$ E T'	$	T'→ϵ
$ E'	$	E'→ϵ
$	$	Accepted



2. construct the LL(1) parser for the following grammar S→(L)|ω
L→L,S|S

Let G=(V,T,P,S) be the given grammar where
V={L,S} T={(,),,,ω} S=S
P= S→(L)|ω; L→L,S|S

S→(L)|ω →①
L→L,S|S →②

Step-1:- Elimination of Left Recursion
consider the production S→(L)|ω
∴ There is no Left Recursion
consider the production L→L,S|S. there is left recursion
$L \to L,S \mid S$ is replaced with
$L \to SL'$
$L' \to ,SL' \mid \epsilon$ } →③
∴ After eliminating left recursion the grammar is
G1: S→(L)|ω
L→SL'
L'→,SL'|ϵ

Step-2:- Eliminating the left factering if any in G1
∴ There is no left factering in G1.

Step-3:- Calculation of First and Follow
V1={S,L,L'} T1={(,),ω,,}
First(S)={(,ω} Follow(S)={,,),$}
First(L)={(,ω} Follow(L)={)}
First(L')={,,ϵ} Follow(L')={)}

Step-4:- constructing of parsing table

M	ω	(	)	,	$
S	S→ω	S→(L)			
L	L→SL'	L→SL'			
L'			L'→ϵ	L'→,SL'	

1) S→(L)|ω
a) S→(L) (α)
First(α)={(}
add S→(L) to M[S,(]
b) S→ω (α)
First(α)={ω}
add S→ω to M[S,ω]

2) L→SL' (A α)
First(SL')={(,ω}
add L→SL' to M[L,(], M[L,ω]
3) L'→,SL'|ϵ
a) L'→,SL' (A α)
First(,SL')={,}
add L'→,SL' to M[L',,]
b) L'→ϵ (A α)
Follow(L')={)}
add L'→ϵ to M[L',)]

Step-5:- check whether the input string is accepted by the parser (or) not
L={ω, (ω), ((ω),(ω)), ----}
Language generated by the grammar
$L(G_1)$={ω, (ω), ((ω),(ω)), ----}

Stack	I/p string	Action
$S	(ω,(ω))$	S→(L)
$)L(	(ω,(ω))$	pop

$)L	a,(a))$	L→SL'
$)L'S	a,(a))$	S→a
$)L'a	a,(a))$	pop
$)L'	,(a))$	L'→,SL'
$)L'S,	,(a))$	pop
$)L'S	(a))$	S→(L)
$)L')L(	(a))$	pop
$)L')L	a))$	L→SL'
$)L')L'S	a))$	S→a
$)L')L'a	a))$	pop
$)L')L'	))$	L'→ε
$)L'	)$	pop
$)L'	)$	L'→ε
$)	)$	pop
$	$	Accepted

(a,(a,ε)) = (a,(a))

Error Recovery in Predictive Parsing:-

1) E→TE'
E'→+TE'/-TE'/ε
T→FT'
T'→*FT'/ /FT'/ε
F→GG'
G'→^F/ε
G→(E)/id

Let G=(V,T,P,S) be the given grammar where
V={E,E',T,T',F,G,G'} T={+,-,*,/,^,(,),id} S=E
P: E→TE'; E'→+TE'/-TE'/ε ; T→FT'; T'→*FT'/FT'/ε
F→GG'; G→^F/ε; G'→(E)/id

step-1:- Elimination of left Recursion
∴ There is no left recursion in the given grammar

step-2:- Elimination of left Factoring if any in G.
∴ There is no left factoring in the given grammar G.

step-3:- calculation of First and follow

First(E) = {(, id} Follow(E) = {$,)} Follow = POP
First(E') = {+, -, ε} Follow(E') = {$,)}
First(T) = {(, id} Follow(T) = {+, -, $,)}
First(T') = {*, /, ε} Follow(T') = {+, -, $,)}
First(G') = {^, ε} Follow(G') = {*, /, +, -, $,)}
First(G) = {(, id} Follow(G) = {^, *, /, +, -, $,)}
First(F) = {(, id} Follow(F) = {*, /, +, -, $,)}

step-4:- constructing of parsing table

M	+	-	*	/	^	(	)	id	$
E	skip	skip	skip	skip	skip	E→TE'	pop	E→TE'	pop
E'	E'→+TE'	E'→-TE'	skip	skip	skip	skip	E'→ε	skip	E'→ε
T	pop	pop	skip	skip	skip	T→FT'	pop	T→FT'	pop
T'	T'→ε	T'→ε	T'→*FT'	T'→/FT'	skip	skip	T'→ε	skip	T'→ε
F	pop	pop	pop	pop	skip	F→GG'	pop	F→GG'	pop
G	pop	pop	pop	pop	pop	G→(E)	pop	G→id	pop
G'	G'→ε	G'→ε	G'→ε	G'→ε	G'→^F	skip	G'→ε	skip	G'→ε

*Syntactic error occurs when the stream of Tokens from lexical analyser disobeys syntactic rules (or) CFG of a language.

Error Recovery strategies in predictive parser:-

1. Panic mode recovery
2. phrase mode recovery

Panic mode recovery:- Rules:-

Rule 1:- If parser looks up entry M[A,a] and finds it empty then the input symbol (a) is skipped.

Rule 2:- If the entry is synch then the Non-terminal

on top of the stack is popped in an attempt to resume parsing.

Rule - 3:- If a token on top of the stack does not match the input symbol then we pop the token from the stack.

Ex:-(panic mode of Error Recovery)

Stack	I/p string	Action
$E	()id+*id$	$E \to TE'$
$E'T	()id+*id$	$T \to FT'$
$E'T'F	()id+*id$	$F \to GG'$
$E'T'G'G	()id+*id$	$G \to (E)$
$E'T'G')E(	()id+*id$	pop
$E'T'G')E	)id+*id$	pop (E)
$E'T'G'	)id+*id$	pop
$E'T'G'	id+*id$	skip
$E'T'G'	+*id$	$G' \to \epsilon$
$E'T'	+*id$	$T' \to \epsilon$
$E'	+*id$	$E' \to +TE'$
$E'T+	**id$	pop
$E'T	*id$	skip
$E'T	id$	$T \to FT'$
$E'T'F	id$	$F \to GG'$
$E'T'G'G	id$	$G \to id$
$E'T'G'id	id$	pop
$E'T'G'	$	$G' \to \epsilon$
$E'T'	$	$T' \to \epsilon$
$E'	$	$E' \to \epsilon$
$	$	Accepted

phrase mode Recovery:-

*In phrase mode of error Recovery the parser corrects the local errors like , ; etc.

*It acts as (or) uses auto correction Mechanism like in word processor

UNIT-2:-

Syntax Analysis:-

Discussion on CFG, LMD, RMD, Ambiguity, Parse tree, role of the parser, classification of parsing Techniques, Backtracking / Brute force method, left Recursion, Left factoring.

Top Down parsing: First and Follow, LL(1) grammars, Non-recursive predictive parsing, recursive descent parser, Error Recovery predictive parsing.

UNIT-III TYPES OF BOTTOM UP APPROACHES

Top Down:-

Start Symbol (root)

↓

I/p string (leaf node)

Bottom up:-

Start (root)

↑

I/p string (leaf node)

*In Bottom up parsing the input string is reduced to start symbol and the derivation used is Reverse of RMD.

Handle and Handle pruning:-

*The crucial task in bottom up parsing is to find handle, the substring that could be reduced by appropriate Non-terminal.

*The process of detecting handles and using them in reduction is called Handle pruning.

Ex:- 1) $S \to aABe$
$A \to Abc | b$
$B \to d$

Input string is abbcde

Sentential form	Handle	Reducing production
a**b**bcde	b	$A \to b$
a**Abc**de	Abc	$A \to Abc$
a**A**de	d	$B \to d$
a**ABe**	aABC	$S \to aABe$
S		

2) $E \to E+E | id$ input string is id+id+id

Sentential form	Handle	Reducing production
id+id+id	id	$E \to id$
E+**id**+id	id	$E \to id$
E+E+**id**	id	$E \to id$

E+E+E E+E E→E+E

E+E E+E E→E+E

E

3) E→E+T/T
T→T*F/F
F→(E)/id input string is id*id

Sentential form	Handle	Reducing Production
id*id	id	F→id
F*id	F	T→F
T*id	id	F→id
T*F	T*F	T→T*F
T	T	E→T
E		

Operator Precedence Parser:-

* It is one of the Bottom-up parser, it is used to define the mathematical operators for the compiler.

operator grammar:-

*A grammar which is used to generally define the Mathematical operations is called operator grammar with some restrictions on CFG.

i) No two variables on RHS are adjacent.
ii) There should not be any ε productions

Ex:- 1) E→E*E/E+E/id - operator grammar (ambiguous grammar for id+id*id

2) S→SAS/a, A→bSb/b — not an operator grammar, it can be written as S→SbSb/Sbs/a - operator grammar
⇓
This is a CFG

3) E→EAE/id, A→+/* — not an operator grammar, it can be written as E→E+E/E*E/id
⇓
This is CFG

*operator precedence parser can parse ambiguous grammar also.

1) construct operator precedence parser for the grammar
E→E*E/E+E/id.

* Here the terminal id will be given highest precedence compared to others.
* Here +/* are left Associative and highest precedence will be given to left most one.
* Among + and *, * will be given highest precedence.
* $ will have least precedence to any other operator.

construction of operator precedence/Relation Table:-

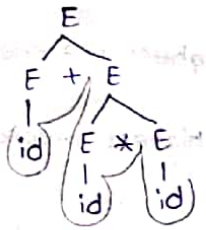
↓f / g→	id	*	+	$
id	-	⋗	⋗	⋗
*	⋖	⋗	⋗	⋗
+	⋖	⋖	⋗	⋗
$	⋖	⋖	⋖	-

Let G=(V,T,P,S) be the given grammar, where
V=set of Non-terminals={E}
T=set of Terminals={$,+,*,id}
P=productions
P: E→E*E/E+E/id
S=E

* The operator precendence parser parses the input using the operator precedence table.

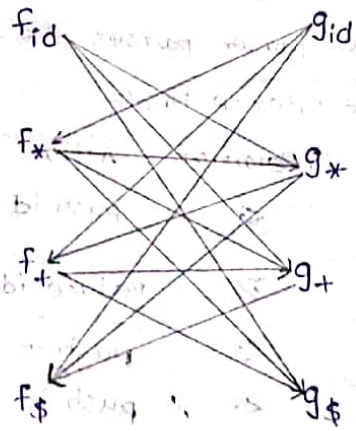
Stack (row)	I/p String (column)	Relation	Action
$	id+id*id $	⋖	push id
$id	+id*id $	⋗	Reduce id by E→id
$E	+id*id $	⋖	push +
$E+	id*id $	⋖	push id
$E+id	*id $	⋗	Reduce id by E→id
$E+E	*id $	⋖	push *
$E+E*	id $	⋖	push id
$E+E*id	$	⋗	Reduce id by E→id
$E+E*E			

parse tree:-



yield : id + id * id

* The disadvantage with operator precedence table is if at all there are 4 operators the size of the table is 16, if at all there are 5 operators the size of the table is 25, if at all there are n operators the size of the table is n^2 and the time complexity is $O(n^2)$.
* To avoid this problem operator function table is used.



$f_{id} \to g_* \to f_+ \to g_+ \to f_\$$

$g_{id} \to f_* \to g_* \to f_+ \to g_+ \to f_\$$

	id	*	+	$
f	4	4	2	0
g	5	3	1	0

operator function table

2) construct operator precedence parser for the grammar

$S \to iEtS \mid iEtSeS \mid a$

$E \to b \mid c \mid d$

* The given grammar obeys the rules of the operator precedence.

* So, the given grammar is a operator precedence grammar.

$V = \{S, E\}$.

$T = \{i, t, e, a, b, c, d\}$

i = if
t = then
e = else

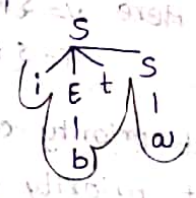
if > then > else

a, b, c, d have equal precedence and a, b, c, d has highest precedence than i, t, e

↓ / →	i	t	e	a	b	c	d	$
i	-	⋗	⋗	⋖	⋖	⋖	⋖	⋗
t	⋖	-	⋗	⋖	⋖	⋖	⋖	⋗
e	⋖	⋖	-	⋖	⋖	⋖	⋖	⋗
a	⋗	⋗	⋗	-	-	-	-	⋗
b	⋗	⋗	⋗	-	-	-	-	⋗
c	⋗	⋗	⋗	-	-	-	-	⋗
d	⋗	⋗	⋗	-	-	-	-	⋗
$	⋖	⋖	⋖	⋖	⋖	⋖	⋖	-

Stack (row)	I/p string (column)	Relation	Action
$	ibta$	⋖	push i
$i	bta$	⋖	push b
$ib	ta$	⋗	Reduce b by E→b
$iE	ta$	⋖	push t
$iEt	a$	⋖	push a
$iEta	$	⋗	Reduce a by S→a
$iEtS			

parse tree:-



yield :- ibta

f_i	g_i
f_t	g_t
f_e	g_e
f_ω	g_ω
f_b	g_b
f_c	g_c
f_d	g_d
$f_\$$	$g_\$$

3) construct the operator precedence parser for the grammar

$P \to SR|S$
$R \to bSR|bS$
$S \to WbS|W$
$W \to L*W|L$
$L \to id$

Here $V = \{P, R, S, W, L\}$
$T = \{b, *, id\}$

* The given grammar is not an operator precedence grammar so convert the given grammar into operator precedence grammar

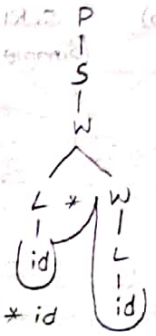
$P \to SbP|SbS|S$
$S \to WbS|W$
$W \to L*W|L$
$L \to id$

Here $V = \{P, S, W, L\}$
$T = \{id, b, *\}$

* id will be given highest priority compared to others.
* * will be given highest priority than 'b'.
* 'b' is right associative, right hand side operator will given highest priority
* '*' is also right associative.
* '$' will be given least priority.

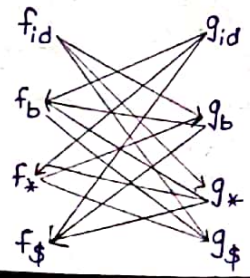
$f \downarrow$ / $g \to$	id	b	*	$
id	-	$\gtrdot$	$\gtrdot$	$\gtrdot$
b	$\lessdot$	$\lessdot$	$\lessdot$	$\gtrdot$
*	$\lessdot$	$\gtrdot$	$\lessdot$	$\gtrdot$
$	$\lessdot$	$\lessdot$	$\lessdot$	-

parse tree:-



yield:- id * id

Stack (row)	I/p string (column)	Relation	Action
$	id * id $	$\lessdot$	push id
$id	* id $	$\gtrdot$	Reduce id by $L \to id$
$L	* id $	$\lessdot$	push *
$L*	id $	$\lessdot$	push id
$L*id	$	$\gtrdot$	Reduce id by $L \to id$
$L*L (Handle pruning)	$ (stop)		
$L*W	L*W		$W \to L*W$
$W	W		$S \to W$
$S	S		$P \to S$
$P			

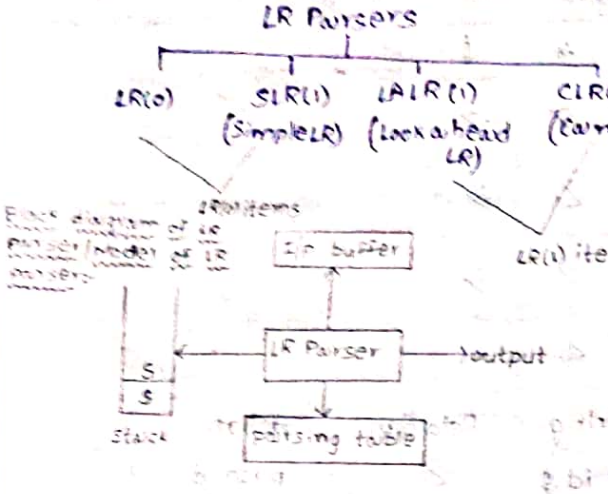


$f_{id} \to g_b \to f_b \to g_\$$

$g_{id} \to f_b \to g_\$$

	id	b	*	$
f	3	1	0	0
g	2	2	0	0

LR Parsers:- Where 'L' is input is scanned from left to Right and 'R' is RMD is constructed in Reverse order. There are four types of LR parsers.



* In order to construct LR(0) and SLR(1) parsing table a canonical collection of LR(0) items are used.
* In order to construct LALR(1) and CLR(1) parsing table a canonical collection of LR(1) items are used.

Item:-

*Any production with a . (dot) in the RHS is called an item.

1) construct LR(0) parser for the grammar $S \to AA$, $A \to aA/b$

Step1:- Convert the given grammar into Augmented grammar

$S' \to S$
$S \to AA$
$A \to aA/b$

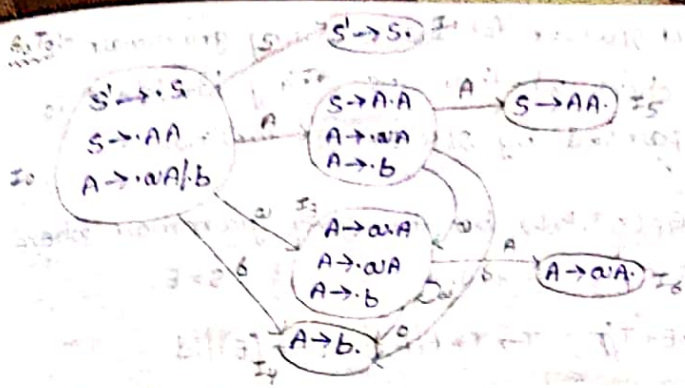
$S' \to S$
$S \to AA \to 0$
$A \to aA \to 1$
$A \to b \to 2$

Step-2:- closure and goto

Closure
$S' \to .S$
$S \to .AA$
$A \to .aA/.b$

Step-3: Calculate of Follow

Follow(S) = {$}
Follow(A) = {a, b, $}



LR(0) parsing table:-

	Action			GoTo	
	a	b	$	S	A
0	S_3	S_4		1	2
1			Accept		
2	S_3	S_4			5
3	S_3	S_4			6
4	r_3	r_3	r_3		
5	r_1	r_1	r_1		
6	r_2	r_2	r_2		

I_1: $S' \to S.$
I_4: $A \to b. \to r_3$
I_5: $S \to AA. \to r_1$
I_6: $A \to aA. \to r_2$

SLR(1) Parsing table:-

	Action			GoTo	
	a	b	$	S	A
0	S_3	S_4		1	2
1			Accept		
2	S_3	S_4			5
3	S_3	S_4			6
4	r_3	r_3	r_3		
5			r_1		
6	r_2	r_2	r_2		

1) construct SLR(1) parser for the following grammar
E→E+T/T check whether the string id*id+id is
T→T*F/F parsed by SLR parser (or) not.
F→(E)/id

step-1:- Let G=(V,T,P,S) be the given grammar where

V={E,T,F} T={id, *, +, (,)} S=E

P: E→E+T/T ; T→T*F/F ; F→(E)/id

step-2:- Calculation of Follow

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

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

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

step-3:- The Augmented grammar for the given grammar G is

E'→E T→T*F→③ F→id→⑥

E→E+T→① T→F→④

E→T→② F→(E)→⑤

step-4:- Canonical collection of LR(0) items.

I0: E'→.E, E→.E+T, E→.T, T→.T*F, T→.F, F→.(E), F→.id

GoTo(I0, E) I1: E'→E., E→E.+T

GoTo(I0, T) I2: T→T.*F, E→T.

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

GoTo(I2, *) I7: T→T*.F, F→.(E), F→.id

GoTo(I4, E) I8: F→(E.), E→E.+T

GoTo(I4, T) I2

GoTo(I4, F) I3

GoTo(I4, ( ) I4

GoTo(I4, id) I5

GoTo(I6, T) I9: E→E+T., T→T.*F

GoTo(I6, F) I3

GoTo(I6, ( ) I4

GoTo(I6, id) I5

GoTo(I7, F) I10: T→T*F.

GoTo(I7, ( ) I4

GoTo(I7, id) I5

GoTo(I8,)) I11: F→(E).

GoTo(I8, +) I6

GoTo(I9, *) I7

I1: E'→E.

I2: E→T.→r2

I3: T→F.→r4

I5: F→id.→r6

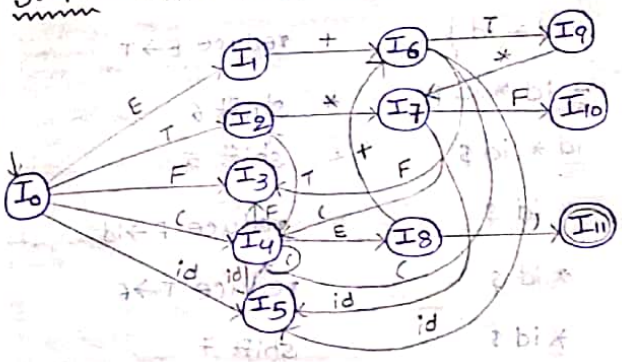
I9: E→E+T.→r1

I10: T→T*F.→r3

I11: F→(E).→r5

step-5:- items that are ended

step-6:- Finite Automata



step-7:- construction of SLR parsing table

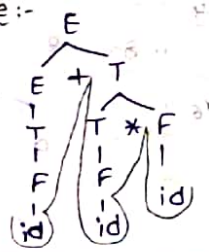
	Action						GoTo		
	id	+	*	(	)	$	E	T	F
I_0	S_5			S_4			1	2	3
I_1		S_6				Accept			
I_2		r_2	S_7		r_2	r_2			
I_3		r_4	r_4		r_4	r_4			
I_4	S_5			S_4			8	2	3
I_5		r_6	r_6		r_6	r_6			
I_6	S_5			S_4				9	3
I_7	S_5			S_4					10

	Action						GoTo		
	id	+	*	(	)	$	E	T	F
I_8		S_6			S_{11}				
I_9		r_1	S_7		r_1	r_1			
I_{10}		r_3	r_3		r_3	r_3			
I_{11}		r_5	r_5		r_5	r_5			

step-8:- check whether the string is accepted (or) not.

Stack	I/p String	Action
$0	id+id*id$	shift 5
$0id5	+id*id$	reduce F→id
$0F3	+id*id$	reduce T→F
$0T2	+id*id$	reduce E→T
$0E1	+id*id$	shift 6
$0E1+6	id*id$	shift 5
$0E1+6id5	*id$	reduce F→id
$0E1+6F3	*id$	reduce T→F
$0E1+6T9	*id$	shift 7
$0E1+6T9*7	id$	shift 5
$0E1+6T9*7id5	$	reduce F→id
$0E1+6T9*7F10	$	reduce T→T*F
$0E1+6T9	$	reduce E→E+T
$0E1	$	Accept

parse tree:-



yield:- id+id*id

* If the RHS contains 1 symbol we have to pop two symbols from the top of the stack and push LHS on to the stack (or) (or) Reduce

Q construct SLR(1) parsing table for the following grammar $S \to (L) / \omega$ check whether the string $(\omega,(\omega,\omega))$ is $L \to L,S / S$ parsed by SLR parser (or) not

step-1:- Let $G=(V,T,P,S)$ be the given grammar where

$V=\{S,L\}$ $T=\{(,),\omega,,\}$ $S=S$

P: $S \to (L) / \omega$

$L \to L,S / S$

step-2:- Calculation of Follow

Follow(S) = {$,), ,}

Follow(L) = {, ,)}

step-3:- The Augmented grammar for the given Grammar G is

$S' \to S$

$S \to (L) \to ①$

$S \to \omega \to ②$

$L \to L,S \to ③$

$L \to S \to ④$

step-4:- canonical collection of LR(0) items

I_0: $S' \to .S$
$S \to .(L)$
$S \to .\omega$

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

GoTo(I_0,()
I_2: $S \to (.L)$
$L \to .L,S$
$L \to .S$
$S \to .(L)$
$S \to .\omega$

GoTo(I_0,ω)
I_3: $S \to \omega.$

GoTo(I_2,L)
I_4: $S \to (L.)$
$L \to L.,S$

GoTo(I_2,S)
I_5: $L \to S.$

GoTo(I_2,()
I_2

GoTo(I_2,ω)
I_3

GoTo(I_4,))
I_6: $S \to (L).$

GoTo(I_4,,)
I_7: $L \to L,.S$
$S \to .(L)$
$S \to .\omega$

GoTo(I_7,S)
I_8: $L \to L,S.$

GoTo(I_7,()
I_2

GoTo(I_7,ω)
I_3

step-5:- Items that are ended

I_1: $S' \to S. \to r_2$

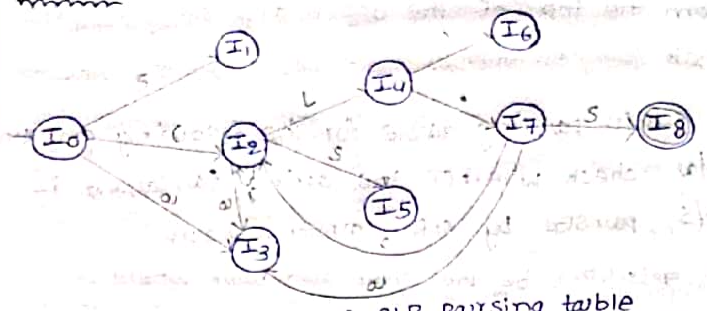
I_3: $S \to \omega. \to r_2$

I_5: $L \to S. \to r_4$

I_6: $S \to (L). \to r_1$

I_8: $L \to L,S. \to r_3$

Step-6:-



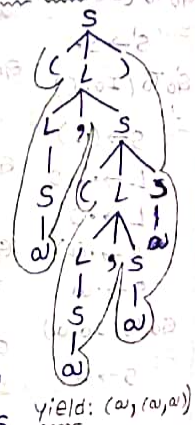
Step-7:- Construction of SLR parsing table

	Action					GoTo	
	(	)	a	,	$	S	L
I_0	S_2		S_3			1	
I_1					Accept		
I_2	S_2		S_3			5	4
I_3		r_2		r_2	r_2		
I_4		S_6		S_7			
I_5		r_4		r_4			
I_6		r_1		r_1	r_1		
I_7	S_2		S_3			8	
I_8		r_3		r_3			

Step-8:- check whether the string is accepted (or) not

Stack	I/p String	Action
$0	(a,(a,a))$	shift 2
$0(2	a,(a,a))$	shift 3
$0(2a3	,(a,a))$	reduce S→a
$0(2S5	,(a,a))$	reduce L→S
$0(2L4	,(a,a))$	shift 7
$0(2L4,7	(a,a))$	shift 2
$0(2L4,7(2	a,a))$	shift 3
$0(2L4,7(2a3	,a))$	reduce S→a
$0(2L4,7(2S5	,a))$	reduce L→S

Parse tree:-



yield: (a,(a,a))

$0(2L4,7(2L4	,a))$	shift 7
$0(2L4,7(2L4,7	a))$	shift 3
$0(2L4,7(2L4,7a3	))$	reduce S→a
$0(2L4,7(2L4,7S8	))$	reduce L→L,S
$0(2L4,7(2L4	))$	shift 6
$0(2L4,7(2L4)6	)$	reduce S→(L)
$0(2L4,7S8	)$	reduce L→L,S
$0(2L4	)$	shift 6
$0(2L4)6	$	reduce S→(L)
$0S1	$	Accept

2) construct the shift-Reduce parser for the grammar S→(L)|a L→L,S|S check whether the string (a,(a,a)) is accepted by parser (or) not.

Stack	I/p Buffer	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

Shift-Reduce parser:- It is a bottom up parser mainly two data structures are used here

1) Stack:- which stores the symbols of the grammar and the bottom of the stack is $.

2) I/p Buffer:- stores the input string to be parsed and every string ends with $.

Actions of shift-Reduce parser:-

1) Shift-push operation:- shift pushes I/p symbol on to the stack.

2) Reduce:- If the top of the stack matches with RHS of a production then it is reduced to left hand side Non-terminal.

3) Accept:- After parsing the entire i/p string if the stack has $S and i/p string contain $ then i/p string is accepted by the parser. The corresponding i/p string belong to the language of the grammar.

*Two types of problems are associated with shift-Reduce parser.

*If parser has a choice of both shift and Reduce Actions but it can select only one choice.

*If parser has a choice of more than one reduce actions but only one choice can be selected.

1) Construct shift-Reduce parser for the grammar
E→E+T|T
T→T*F|F
F→(E)|id
check whether the string id*id is accepted by parser (or) not.

Stack	Buffer I/p String	Action
$	id * id $	shift
$id	* id $	reduce F→id
$F	* id $	reduce T→F
$T	* id $	shift
$T*	id $	shift
$T*id	$	reduce F→id
$T*F	$	reduce T→T*F
$T	$	reduce E→T
$E	$	Accept

UNIT-3:-

Types of Bottom-up approaches: Introduction to Bottom-up parser, shift-Reduce parser, Operator precedence parser, Why LR parsers, Model of an LR parsers, construction of SLR tables, More powerful LR parsers: construction of CLR(1), LALR parsing tables, differences between LR and LL parser, Dangling Else ambiguity, Error Recovery in LR parsing, comparision of all Bottom-up approaches with all Top-Down approaches.

Differences between LR Parsers and LL Parsers:-

LR Parser - it can handle a larger range of languages than LL Parsing	LL Parser
*These are Bottom-up parsers	*These are Top-Down parsers.
*This is complex to implement	*This is simple to implement.
*For LR(K) the first L means the input is read (or) scanned from left to Right R means it uses RMD in reverse order for the input string. K indicates a number, no.of look a head symbols to predict the parsing process where k is either 0 (or) 1.	*for LL(1), the first L means the input is read (or) scanned from left to Right the second L means it uses LMD for the input string, the number 1 indicates one look a head symbol to predict the parsing process.
*These are efficient parsers	*These are less efficient parsers.

*It is applied to a large class of programming language.	*It is applied to small class of programming language.

SLR-Parser Procedure:-

1) Calculation of canonical collection of LR(0) items

* For the given grammar G, (initially add S) write Augmented grammar G'
* For the grammar G', initially add S'→·S to the LR(0) items
* For each set of items I_i and for each grammar symbol X 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 is not in LR(0) items.
* The set of LR(0) items has to be constructed until no more set of items can be added.

2) Construction of SLR(1) parsing table

* Initially construct set of LR(0) items

$$C = \{I_0, I_1, \dots, I_n\}$$

* The parsing actions are based on each item I_i the actions are given as follows

a) If $A \to \alpha \cdot a\beta$ is in I_i and goto(I_i, a) = I_j then set action[i, a] as Shift j

b) If there is a rule $A \to \alpha \cdot$ is in I_i then set action[i, a] to reduce $A \to \alpha$ for all symbols a, $a \in$ Follow(A)

c) If $S' \to S \cdot$ is in I_i then the entry in the action table then set action[i, $] = "accept"

3) The goto part of SLR(1) table can be filled as

* The goto transitions for state i is considered for Non-Terminals only.
* If goto[I_i, A] = I_j then goto[i, A] = j

4) All the entries not defined by rule 2 and rule 3 are considered to be an error.

3 construct SLR(1) Parser for the grammar

S→cA/ccB
A→cA/a
B→ccB/b

Step-1:- Let G=(V,T,P,S) be the given grammar where

V={S, A, B} T={a, b, c} S=S

P: S→cA/ccB
A→cA/a
B→ccB/b

Step-2:- Calculation of follow

Follow(S) = {$} Follow(A) = {$} Follow(B) = {$}

Step-3:- The Augmented grammar for the given grammar G is

S'→S
S→cA →①
S→ccB →②
A→cA →③
A→a →④
B→b →⑥
B→ccB →⑤

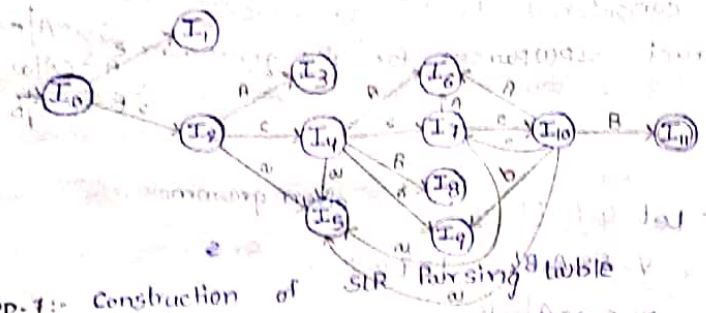
Step-4:- Canonical collection of LR(0) items

I_0: S'→·S
S→·cA
S→·ccB
GoTo(I_0, S)
I_1: S'→S·
GoTo(I_0, c)
I_2: S→c·A
A→·cA; A→·a
S→c·cB
GoTo(I_2, A)
I_3: S→cA·
GoTo(I_2, c)
I_4: A→c·A
A→·cA
A→·a
S→cc·B
B→·ccB
B→·b
GoTo(I_2, a)
I_5: A→a·
GoTo(I_4, A)
I_6: A→cA·
GoTo(I_4, c)
I_7: A→c·A
A→·cA
A→·a
B→c·cB
GoTo(I_4, a)
I_5
GoTo(I_4, B)
I_8: S→ccB·
GoTo(I_4, b)
I_9: B→b·
GoTo(I_7, A)
I_6
GoTo(I_7, c)
I_7
GoTo(I_7, a)
I_5
GoTo(I_7, B)
I_{10}: B→ccB·
GoTo(I_7, b)
I_9
GoTo(I_{10}, A)
I_6
GoTo(I_{10}, c)
I_7
GoTo(I_{10}, a)
I_5
GoTo(I_{10}, B)
I_{10}: B→ccB·
GoTo(I_{10}, b)
I_9

Step-5:- Items that are ended

I_1: $S' \to S.$ → r0
I_2: $S \to cA.$ → r1
I_5: $A \to a.$ → r4
I_6: $A \to cA.$ → r3
I_8: $S \to ccB.$ → r2
I_9: $B \to b.$ → r6
I_{10}: $B \to ccB.$ → r5

Step-6:- Finite Automaton



Step-7:- Construction of SLR Parsing table

	Action				GoTo		
	a	b	c	$	S	A	B
I_0			S_2		1		
I_1				Accept			
I_2	S_5		S_4			3	
I_3				r_1			
I_4	S_5	S_9	S_7			6	8
I_5				r_4			
I_6				r_3			
I_7	S_5		S_{10}			6	
I_8				r_2			
I_9				r_6			
I_{10}		S_9	S_7			6	11
I_{11}				r_5			

Language accepted by the given grammar is

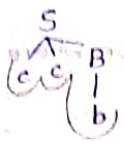
$L(G) = \{ca, ccb, cccb, ccca, \ldots\}$

Step-8:- check whether the string is accepted (or) not

Input String is ccb

Stack	I/p Buffer	Action
$0	ccb$	shift 2
$0c2	cb$	shift 4
$0c2c4	b$	shift 9
$0c2c4b9	$	reduce B→b
$0c2c4B8	$	reduce S→ccB
$0S1	$	Accept

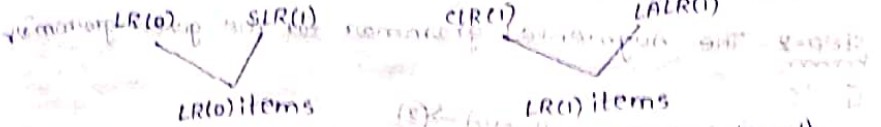
Step-9:- Parse tree



yield:- ccb

CLR(1) Parser:-

* It a bottom-up parser. It uses LR(1) items



* The canonical collection of set of items is a parsing technique in which a look ahead symbol is generated while constructing set of items therefore these collection of set of items is referred to as LR(1). The value 1 is referred to 1 look ahead symbol in the set of items.

Construction of canonical collection of LR(1) items:-

Step-1:- For the grammar G, initially add $S' \to S$ in the

set of items C.

Step-2:- For each set of Items I_i, and for each grammar Symbol X and closure(I_i, X)

Step-3:- The closure function can be computed as for each item $A \to \alpha \cdot X\beta a$ and rule $X \to Y$ and $b \in First(\beta a)$ such that $X \to \cdot Y$ and b is not in I_i then add $X \to \cdot Y, b$ to I

step-4:- Similarly the GoTo function can be computed as $[A \to \alpha \cdot X\beta, a]$ is in I and rule $[A \to \alpha X \cdot \beta, a]$ is not in I then add $[A \to \alpha X \cdot \beta, a]$ to I

1) construct CLR(1) parser for the following grammar

$S \to AA$

$A \to aA|b$

Step-1:- Let $G=(V,T,P,S)$ be the given grammar where

$V=\{S,A\}$ $T=\{a,b\}$ $S=S$

P: $S \to AA$

$A \to aA|b$

$L(G)=\{bb, abb, \dots\dots\dots\}$

Step-2:- The Augmented grammar for the given grammar G is

$S' \to S \to (0)$ $A \to aA \to (2)$

$S \to AA \to (1)$ $A \to b \to (3)$

Step-3:- Construction of canonical collection of LR(1) items.

I_0: $S' \to \cdot S, \$$
$S \to \cdot AA, \$$
$A \to \cdot aA, a/b$
$A \to \cdot b, a/b$

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

GoTo(I_0, A)
I_2: $S \to A \cdot A, \$$
$A \to \cdot aA, \$$
$A \to \cdot b, \$$

GoTo(I_0, a)
I_3: $A \to a \cdot A, a/b$
$A \to \cdot aA, a/b$
$A \to \cdot b, a/b$

GoTo(I_0, b)
I_4: $A \to b \cdot, a/b$

GoTo(I_2, A)
I_5: $S \to AA \cdot, \$$

GoTo(I_2, a)
I_6: $A \to a \cdot A, \$$
$A \to \cdot aA, \$$
$A \to \cdot b, \$$

GoTo(I_2, b)
I_7: $A \to b \cdot, \$$

GoTo(I_3, A)
I_8: $A \to aA \cdot, a/b$

GoTo(I_3, a)
I_3

GoTo(I_3, b)
I_4

GoTo(I_6, A)
I_9: $A \to aA \cdot, \$$

GoTo(I_6, a)
I_6

GoTo(I_6, b)
I_7

step-4:- Items that are ended

I_1: $S' \to S \cdot, \$ \to (r_0)$

I_4: $A \to b \cdot, a/b \to (r_3)$

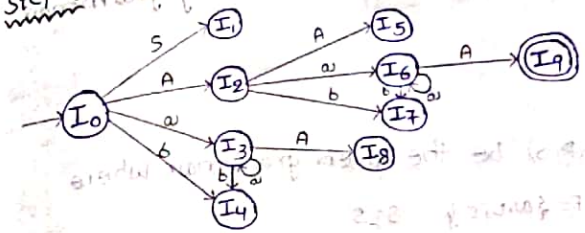
I_5: $S \to AA \cdot, \$ \to (r_1)$

I_7: $A \to b \cdot, \$ \to (r_3)$

I_8: $A \to aA \cdot, a/b \to (r_3)$

I_9: $A \to aA \cdot, \$ \to (r_2)$

step-5:- Finite Automata



step-6:- Construction of CLR parsing table

	Action			GoTo	
	a	b	$	S	A
I_0	S_3	S_4		1	2
I_1			Accept		
I_2	S_6	S_7			5
I_3	S_3	S_4			8
I_4	r_3	r_3			
I_5			r_1		
I_6	S_6	S_7			9
I_7			r_3		
I_8	r_2	r_2			
I_9			r_2		

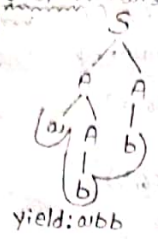
Step-7:- check whether the string is accepted (or) not

string is abb

stack	I/p Buffer	Action
$0	abb$	shift 3
$0a3	bb$	shift 4
$0a3b4	b$	reduce A→b

$0a3A8	b$	reduce A→aA
$0A2	b$	shift 7
$0A2b7	$	reduce A→b
$0A2A5	$	reduce S→AA
$0S1	$	Accept

step-8: Parse tree



yield: abb

2) Construct CLR(1) Parser for the following grammar

S→cA|ccB
A→cA|a
B→ccB|b

Step-1:- Let G=(V,T,P,S) be the given grammar where

V={S,A,B} T={a,b,c} S=S

P: S→cA|ccB
A→cA|a
B→ccB|b

L(G)={ca, ccb, cca, ccca, ---------}

Step-2:- The Augmented grammar for the given grammar G is

S'→S→(0) A→cA→(3) B→b→(6)
S→cA→(1) A→a→(4)
S→ccB→(2) B→ccB→(5)

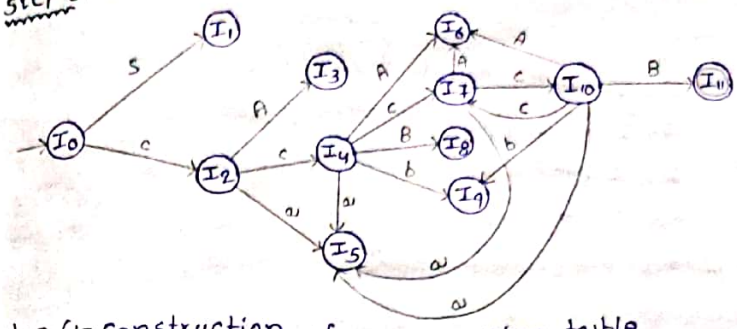
Step-3:- Construction of canonical collection of LR(1) items

I_0: S'→.S,$
S→.cA,$
S→.ccB,$
GoTo(I0,S)
I_1: S'→S.,$
GoTo(I0,c)
I_2: S→c.A,$
A→.cA,$
A→.a,$
S→c.cB,$
GoTo(I2,A)
I_3: S→cA.,$
GoTo(I2,c)
I_4: A→c.A,$
A→.cA,$
A→.a,$
S→cc.B,$
B→.ccB,$
B→.b,$
GoTo(I2,a)
I_5: A→a.,$
GoTo(I4,A)
I_6: A→cA.,$
GoTo(I4,c)
I_7: A→c.A,$
A→.cA,$
A→.a,$
B→c.cB,$
GoTo(I4,a)
I5
GoTo(I4,B)
I_8: S→ccB.,$
GoTo(I4,b)
I_9: B→b.,$
GoTo(I7,A)
I6
GoTo(I7,c)
I_{10}: A→c.A,$
A→.cA,$
A→.a,$
B→cc.B,$
B→.ccB,$
B→.b,$
GoTo(I7,a)
I5
GoTo(I10,A)
I6
GoTo(I10,c)
I7
GoTo(I10,a)
I5
GoTo(I10,B)
I_{11}: B→ccB.,$
GoTo(I10,b)
I9

Step-4:- Items that are ended

I_1: S'→S.,$ →(r_0)
I_3: S→cA.,$ →(r_1)
I_5: A→a.,$ →(r_4)
I_6: A→cA.,$ →(r_3)
I_8: S→ccB.,$ →(r_2)
I_9: B→b.,$ →(r_6)
I_{11}: B→ccB.,$ →(r_5)

Step-5:- Finite Automata



Step-6:- construction of CLR parsing table

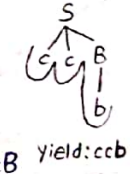
	Action				GoTo		
	a	b	c	$	S	A	B
I_0			S_2		1		
I_1				Accept			
I_2	S_5		S_4			3	
I_3				r_1			
I_4	S_5	S_9	S_7			6	8
I_5				r_4			
I_6				r_3			
I_7	S_5		S_{10}			6	
I_8				r_2			
I_9				r_6			
I_{10}	S_5	S_9	S_7				11
I_{11}				r_5			

Step-7:- check whether the string is accepted (or) not.

I/p string is ccb

Stack	I/p Buffer	Action
$0	ccb$	shift 2
$0c2	cb$	shift 4
$0c2c4	b$	shift 9
$0c2c4b9	$	reduce B→b
$0c2c4B8	$	reduce S→ccB
$0S1	$	Accept

step-8:- Parse tree



yield: ccb

LALR(1) Parser:-

*It means look ahead LR Parser. It is a bottom up parser. It uses LR(1) items.

1) construct LALR(1) Parser for the following grammar

$S \to L=R \mid R$

$L \to *R \mid id$ id=id

$R \to L$

step-1:- Let $G=(V,T,P,S)$ be the given grammar where

$V=\{S,L,R\}$ $T=\{=, *, id\}$ $S=S$

P: $S \to L=R \mid R$

$L \to *R \mid id$

$R \to L$

$L(G)=\{id, id=id, *id, \text{---------}\}$

step-2:- The Augmented grammar for the given grammar G is

$S' \to S \to (0)$

$S \to L=R \to (1)$

$S \to R \to (2)$

$L \to *R \to (3)$

$L \to id \to (4)$

$R \to L \to (5)$

step-3:- Calculation of canonical collection of LR(1) items.

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

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

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

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

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

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

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

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

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

GoTo(I_4, *)
I_4

GoTo(I_4, id)
I_5

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

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

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

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

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

GoTo(I_{11}, L)
I_{10}

GoTo(I_{11}, *)
I_{11}

GoTo(I_{11}, id)
I_{12}

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

Step-4:- The states with same LR(0) items and different look ahead

$I_4, I_{11} \rightarrow I_{411}$

$I_5, I_{12} \rightarrow I_{512}$

$I_7, I_{13} \rightarrow I_{713}$

$I_8, I_{10} \rightarrow I_{810}$

Step-5:- Items that are ended

$I_1: S' \rightarrow S\cdot, \$ \rightarrow (r_0)$

$I_3: S \rightarrow R\cdot, \$ \rightarrow (r_2)$

$I_5: L \rightarrow id\cdot, =/\$ \rightarrow (r_4)$

$I_7: L \rightarrow *R\cdot, =/\$ \rightarrow (r_3)$

$I_8: R \rightarrow L\cdot, =/\$ \rightarrow (r_5)$

$I_9: S \rightarrow L=R\cdot, \$ \rightarrow (r_1)$

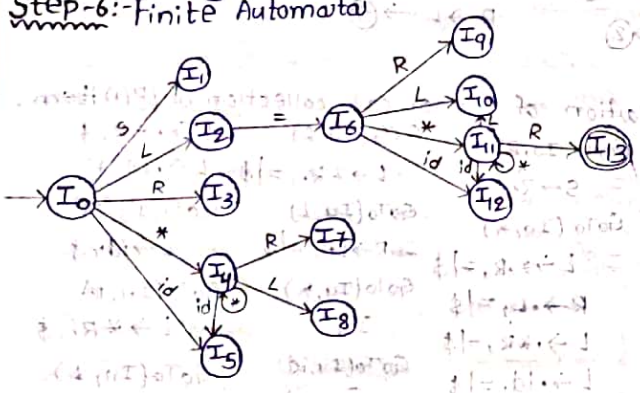
$I_{10}: R \rightarrow L\cdot, \$ \rightarrow (r_5)$

$I_{12}: L \rightarrow id\cdot, \$ \rightarrow (r_4)$

$I_{13}: L \rightarrow *R\cdot, \$ \rightarrow (r_3)$

$I_2: R \rightarrow L\cdot, \$ \rightarrow (r_3)$

Step-6:- Finite Automata



Step-7:- construction of CLR Parsing table

	Action				GoTo		
	id	=	*	$	S	L	R
I_0	S_5		S_4		1	2	3
I_1				Accept			
I_2		S_6		r_5			
I_3				r_2			
I_4	S_5		S_4			8	7
I_5		r_4		r_4			
I_6	S_{12}		S_{11}			10	9
I_7		r_3		r_3			
I_8		r_5		r_5			
I_9				r_1			
I_{10}				r_5			
I_{11}	S_{12}		S_{11}			10	13
I_{12}				r_4			
I_{13}				r_3			

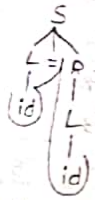
Step-8:- construction of LALR Parsing table

	Action				GoTo		
	id	=	*	$	S	L	R
I_0	S_{512}		S_{411}		1	2	3
I_1				Accept			
I_2		S_6		r_5			
I_3				r_2			
I_{411}	S_{512}		S_{411}			810	713
I_{512}		r_4		r_4			
I_6	S_{512}		S_{411}			810	9
I_{713}		r_3		r_3			
I_{810}		r_5		r_5			
I_9				r_1			

Step-9:- check whether the string is accepted (or) not
String is id=id (LALR)

Stack	I/p Buffer	Action
$0	id=id$	shift 512
$0id512	=id$	reduce L→id
$0L2	=id$	shift 6
$0L2=6	id$	shift 512
$0L2=6id512	$	reduce L→id
$0L2=6L8	$	reduce R→L
$0L2=6R9	$	reduce S→L=R
$0S1	$	Accept

Parse tree:-

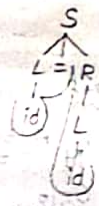


yield:- id=id

Step-10:- check whether the string is accepted (or) not
String is id=id (CLR)

Stack	I/p Buffer	Action
$0	id=id$	shift5
$0id5	=id$	reduce L→id
$0L2	=id$	shift6
$0L2=6	id$	shift 12
$0L2=6id12	$	reduce L→id
$0L2=6L10	$	reduce R→L
$0L2=6R9	$	reduce S→L=R
$0S1	$	Accept

Parse tree:-



yield:- id=id

Dangling else ambiguity:-

*It is a problem where the 'else' does not know to which 'if' it belong to.

*When there are multiple 'if's and one 'else', there is an ambiguity about which if, the else should be assigned.

Code snippet:-

```
if<condition 1> then
  if<condition 2> then
    statement 1
  else
    statement 2
```

Interpretation-1:-

```
if<condition-1> then
{
  if<condition-2> then
    statement-1
}
else
  statement-2
```

Interpretation-2:-

```
if <condition-1> then
{
  if<condition-2> then
    statement-1
}
  else
    statement-2
```

How to eliminate this ambiguity:-

1. By rewriting the grammar
2. Use ambiguous grammar with additional rules to resolve ambiguity

Dangling else grammar:-

stmt → if expr then stmt | if expr then stmt else stmt | other s

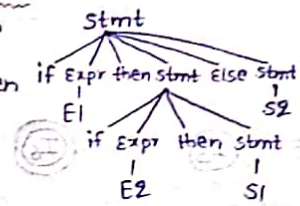
Parse tree for dangling else:-

I/p string:- if E_1 then if E_2 then S_1 else S_2

* The additional rule added here to resolve ambiguity is match 'else' with nearest 'if' 'then'.

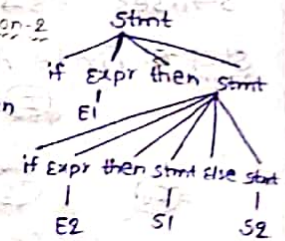
Interpretation-1

```
if<E1> then
{
  if<E2> then
    S1
}
else
  S2
```



Interpretation-2

```
if<E1> then
{
  if <E2> then
    S1
  else
    S2
}
```



1) Prove that given grammar has ambiguity using SLR

$S \to is/ises/a$ (S→ifes, e→if|es|others)

step-1:- Let $G=(V,T,P,S)$ be the given grammar where

$V=\{S\}$ $T=\{i,e,a\}$ $S=S$

$P: S \to is/ises/a$

step-2:- Calculation of Follow

Follow(S) = {$, e}

step-3:- The Augmented grammar for the given grammar G is

$S' \to S$ → (0) $S \to ises$ → (2)

$S \to is$ → (1) $S \to a$ → (3)

Step-4:- Canonical collection of LR(0) items

I_0:
$S' \to .S$
$S \to .is$
$S \to .ises$
$S \to .a$

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

GoTo(I_0,i)
I_2: $S \to i.s$
$S \to .is$
$S \to .ises$
$S \to .a$
$S \to i.ses$

GoTo(I_0,a)
I_3: $S \to a.$

GoTo(I_2,S)
I_4: $S \to is.$
$S \to is.es$

GoTo(I_2,i)
I_2

GoTo(I_2,a)
I_3

GoTo(I_4,e)
I_5: $S \to ise.s$
$S \to .is$
$S \to .ises$
$S \to .a$

GoTo(I_5,S)
I_6: $S \to ises.$

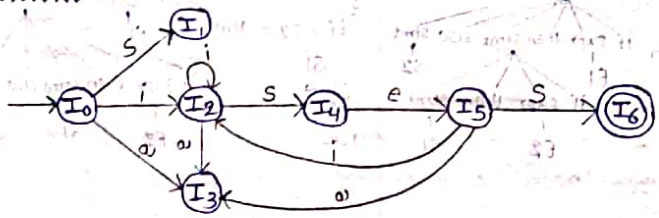
GoTo(I_5,i)
I_2

GoTo(I_5,a)
I_3

Step-5:- Items that are ended

I_1: $S' \to S.$
I_3: $S \to a.$ → (r_3)
I_4: $S \to is.$ → (r_1)
I_6: $S \to ises.$ → (r_2)

Step-6:- Finite Automata



Step-7:- Construction of SLR Parsing table

	Action				GoTo
	i	e	a	$	S
I_0	S2		S3		1
I_1				Accept	
I_2	S2		S3		4
I_3		r_3		r_3	
I_4		S_5, r_1		r_1	
I_5	S2		S3		6
I_6		r_2		r_2	

Differences among Types of LR Parser:-

SLR(1)	LALR (1)	CLR(1)
1) It is a Simple LR	1) It is a Lookahead LR.	1) It is a Canonical LR
2) Canonical collection of LR(0) items have to be calculated.	2) Canonical collection of LR(1) items have to be calculated.	2) canonical collection of LR(1) items have to be calculated
3) The size of the table is small compared to CLR	3) The size of the table is small compared to CLR.	3) size of the table is large compared to other LR parsers.
4) Time and Space Complexity is less.	4) Time and Space complexity is less.	4) Time and Space Complexity is large compared to other
5) Error detection is not immediate	5) Error detection is not immediate.	5) Error detection is immediate.
6) It is an easy method based on follow function	6) Look aheads are to be calculated and this method is applicable to Wider class than SLR.	6) Look aheads have to be calculated and this method is most powerful than other LR parsers.

Error Detection Recovery in LR Parsers:- The LR Parser is a table-driven parsing method in which the blank entries are treated as error.

* There are two strategies of error Recovery
  1. Panic mode error recovery (Synchronized tokens are used)
  2. Phrase mode error recovery (local errors are corrected)

1. Consider the following grammar to illustrate error recovery in LR Parser (SLR Parser)

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

Step-1:- Let $G=(V,T,P,S)$ be the given grammar where

$V=\{E\}$ $T=\{+, *, (,), id\}$ $S=E$

P: $E \to E+E$
$E \to E*E$
$E \to (E)$
$E \to id$

Step-2:- calculation of follow

$Follow(E)=\{\$,+,*,)\}$

Step-3:- The Augmented grammar for the given grammar G is

$E' \to E$
$E \to E+E \to (1)$
$E \to E*E \to (2)$
$E \to (E) \to (3)$
$E \to id \to (4)$

Step-4:- Canonical collection of LR(0) items.

I_0: $E' \to \cdot E$, $E \to \cdot E+E$, $E \to \cdot E*E$, $E \to \cdot (E)$, $E \to \cdot id$

GoTo(I_0, E): I_1: $E' \to E \cdot$, $E \to E \cdot +E$, $E \to E \cdot *E$

GoTo(I_0, (): I_2: $E \to (\cdot E)$, $E \to \cdot E+E$, $E \to \cdot E*E$, $E \to \cdot (E)$, $E \to \cdot id$

GoTo(I_0, id): I_3: $E \to id \cdot$

GoTo(I_1, +): I_4: $E \to E+\cdot E$, $E \to \cdot E+E$, $E \to \cdot E*E$, $E \to \cdot (E)$, $E \to \cdot id$

GoTo(I_1, *): I_5: $E \to E*\cdot E$, $E \to \cdot E+E$, $E \to \cdot E*E$, $E \to \cdot (E)$, $E \to \cdot id$

GoTo(I_2, E): I_6: $E \to (E \cdot)$, $E \to E \cdot +E$, $E \to E \cdot *E$

GoTo(I_2, (): I_2

GoTo(I_2, id): I_3

GoTo(I_4, E): I_7: $E \to E+E \cdot$, $E \to E \cdot +E$, $E \to E \cdot *E$

GoTo(I_4, (): I_2

GoTo(I_4, id): I_3

GoTo(I_5, E): I_8: $E \to E*E \cdot$, $E \to E \cdot +E$, $E \to E \cdot *E$

GoTo(I_5, (): I_2

GoTo(I_5, id): I_3

GoTo(I_6,)): I_9: $E \to (E) \cdot$

GoTo(I_6, +): I_4

GoTo(I_6, *): I_5

GoTo(I_7, +): I_4

GoTo(I_7, *): I_5

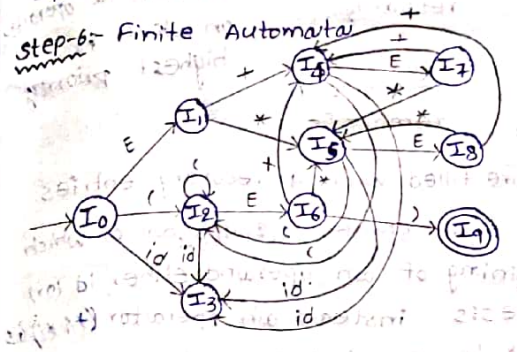
GoTo(I_8, +): I_4

GoTo(I_8, *): I_5

Step-5:- Items that are ended.

I_1: $E' \to E \cdot$
I_3: $E \to id \cdot \to (r_4)$
I_7: $E \to E+E \cdot \to (r_1)$
I_8: $E \to E*E \cdot \to (r_2)$
I_9: $E \to (E) \cdot \to (r_3)$

Step-6:- Finite Automata:-



Step-7:- Construction of SLR Parsing table

	Action						GoTo
	+	*	(	)	id	$	E
I_0	e_1	e_1	S_2	e_2	S_3	e_1	1
I_1	S_4	S_5	e_3	e_2	e_3	Accept	
I_2	e_1	e_1	S_2	e_2	S_3	e_1	6
I_3	r_4	r_4	r_4	r_4	r_4	r_4	
I_4	e_1	e_1	S_2	e_2	S_3	e_1	7
I_5	e_1	e_1	S_2	e_2	S_3	e_1	8
I_6	S_4	S_5	e_3	S_9	e_3	e_4	
I_7	~~S_4~~,r_1	~~S_5~~,r_1	r_1	r_1	r_1	r_1	
I_8	~~S_4~~,r_2	~~S_5~~,r_2	r_2	r_2	r_2	r_2	
I_9	r_3	r_3	r_3	r_3	r_3	r_3	

Shift-reduce conflict:-

$(7,+) - E \to E+E$ retain r_1 remove S_4

(Same priority) (ignore shift (remove))

(7, *) — E→E+E
retain S_5 remove, (Highest priority) (ignore reduce)

(8, +) — E→E*E
remove S_4 retain r_2

(8, *) — E→E*E
retain r_2 remove S_5

Retain the operation which has highest priority

* Remaining entries are filled with 4 recovery entries

1) **e1**: e1 is called from states 0, 2, 4, 5 all of which expect the begining of an operand either id (or) an left parenthesis instead an operator (+, *) is + (or) * (or) $ (or)) is found. So issue diagonised is missing operand.

Action:- Push an imaginary id on the stack and cover it with state 3.

2) **e2**:- This routine is called from states 0, 1, 2, 4, 5 on finding a right parenthesis ()), issue diagonised is unbalanced parenthesis.

Action:- Remove the ')' from the input string

3) **e3**:- This routine is called from states 1 (or) 6 when expecting an operator, on id (or) '(' is found, issue diagonised is missing operator

Action:- Push operator (+ (or) *) onto the stack and cover it with state 4.

4) **e4**:- This routine is called from states '6' when the end of the input is found ($), issue diagonised is missing parenthesis.

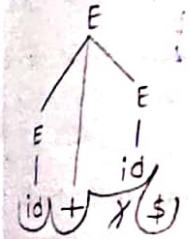
Action:- Push a ')' onto the stack and cover it with state 9.

step-8:- Check whether the input string is accepted (or) not.

I/p string is id+) $ → error string

Stack	I/p Buffer	Action
$0	id+) $	Shift 3
$0id3	+) $	reduce E→id
$0E1	+) $	Shift 4
$0E1+4	) $	e2 (remove ')' from the input string)
$0E1+4	$	e1 (Push an imaginary id on the stack and cover it with state 3).
$0E1+4id3	$	reduce E→id
$0E1+4E7	$	reduce E→E+E
$0E1	$	Accept

step-9:- Parse tree



yield: id+id$

UNIT-4

Semantic Analysis: SDT schemes, evaluation of semantic rules, intermediate codes, three address codes - quadruples, triples, abstract syntax trees, types and declarations, type checking.

Symbol Table: Use and need of symbol tables, runtime environment storage organization, stack allocation, access to non-local data, heap management.

Semantic Analysis:-

*Semantic Analysis phase Analyzes the input Semantically.

*In this phase compiler Analyzes the meaning of the program. Extra Syntactic are imposed in this phase and it is static in nature.

*The properties that can't be capture by CFG are captured by semantic rules. These properties could be Name, Scope, Type checking and Type Conversion etc....

Syntax Directed Definition(SDD):-

SDD = CFG + Semantic Rules

*An SDD is a CFG with semantic Rules

*Attributes are associated with grammar symbols. And Semantic rules are associated with productions.

*If X is a symbol and 'a' is one of it's attributes then X.a denotes value at symbol X.

*Attributes may be Numbers, strings, Memory locations, datatypes etc....

Types of Attributes:-

There are two types of Attributes

1. Synthesized Attribute
2. Inherited Attribute

Synthesized Attribute:-

*If a symbol takes value from it's children then it is Synthesized Attribute.

Inherited Attribute:-

*If a symbol takes value from it's parent (or) siblings then it is called Inherited Attribute.

Types of SDD:- There are 2 types of SDD

1. S-attributed SDD (or) s-attributed definition (or) S-attributed grammar.
2. L-attributed SDD (or) L-attributed definition (or) L-attributed grammar.

S-attributed SDD	L-attributed SDD
*An SDD that uses only Synthesized attributes is called as s-attributed SDD Ex:- A→BCD A.val = B.val A.val = C.val A.val = D.val	*A SDD that uses both Synthesized and Inherited attributes is called as L-attributed SDD. But each inherited attribute is restricted to inherit from parent (or) left most symbol of RHS only. Ex:- A→BCD C.val = A.val ✓ C.val = B.val ✓ C.val = D.val ✗
*Semantic Actions are always placed at right end of the production. It is also called as post-fix SDD.	*Semantic rules are placed anywhere on RHS.
*Attributes are evaluated with bottom-up parser.	*Attributes are evaluated by Traversing parse tree depth first left to Right.

Syntax Directed Translation(SDT):-

*SDT is used to evaluate the order of semantic rules. In Translation scheme the semantic rules are embedded within the right side of the productions.

The position at which an Action is to be executed is shown by enclosed within braces({}).

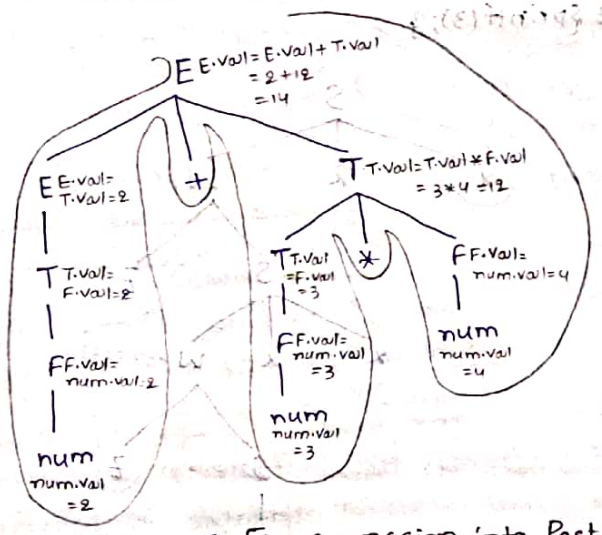
*SDT is implemented by constructing a parse tree performing the actions in a Top-Down left to right order (or) Depth first left to right order.

*SDT is implemented by parsing the input and producing a parse tree as a result.

SDT for evaluation of Expression:-

1. E→E+T {E.val = E.val + T.val}
   E→T {E.val = T.val}
   T→T*F {T.val = T.val * F.val}
   T→F {T.val = F.val}
   F→num {F.val = num.val}

Expression = 2+3*4 = 14

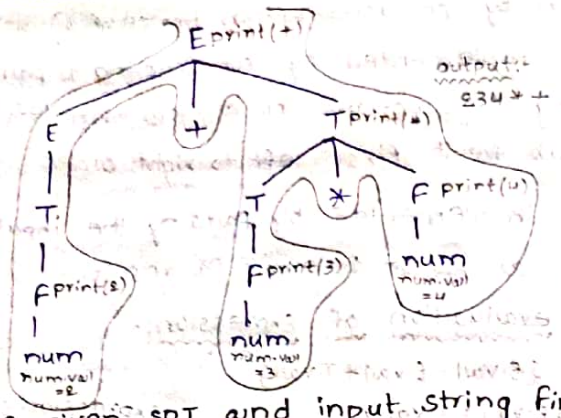


SDT for converting InFix Expression into PostFix Expression:-

2. E→E+T {Printf('+');}
   E→T {}
   T→T*F {printf('*');}
   T→F {}
   F→num {printf(num.val);}

InFix Expression is 2+3*4

PostFix Expression is 234*+

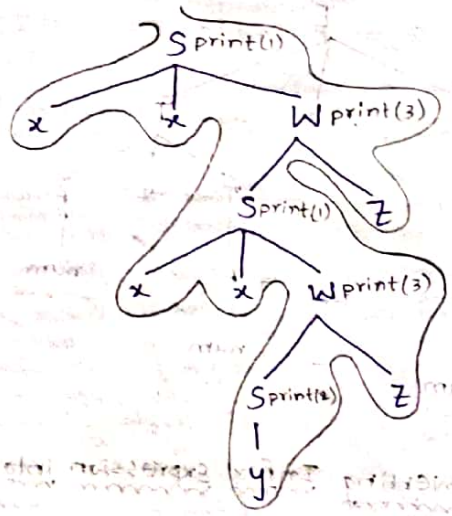


3. For the given SDT and input string find the output using SDD.

$S \to xxW$ {printf(1);}
$S \to y$ {printf(2);}
$W \to Sz$ {printf(3);}

string is xxxxyzz

23131



output:
23131

4. For the given SDT and Input string Find the output using SDD.

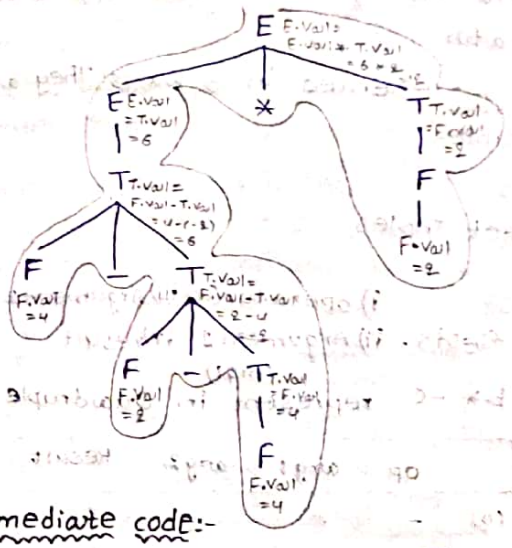
$E \to E * T$ {E.val = E.val * T.val}
$E \to T$ {E.val = T.val}
$T \to F - T$ {T.val = F.val - T.val}
$T \to F$ {T.val = F.val}

$W = 4-2-4*2$
$= 2-4*2$
$= 2-8$
$= -6$
$= 4-2=2 \times 2$
$= 4-(-2) \times 2 = 6 \times 2 = 12$

$F \to 2$ {F.val = 2}
$F \to 4$ {F.val = 4}

- is right associate operator
* is left associate operator

For this grammar '-' is given high priority than '*'.



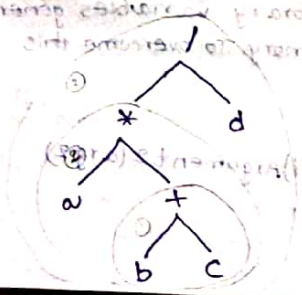
Intermediate code:-

Forms of Intermediate code:- There are 3 forms of intermediate code we can represent the intermediate code in 3 ways.

i) Syntax tree (or) Abstract syntax tree:-

* Each internal node represents an operator and leaf node represents operands.

Ex:- $a * (b+c) \mid d$



ii) Post Fix Notation:-

Ex:- 1. $(a+b) * c \rightarrow ab+c*$

2. $(a-b) * (c/d) \rightarrow ab-cd/*$

iii) 3-address code representation:- In 3-address code instruction, each instruction should contain atmost 3-addresses. And the right hand side should consists of atmost 1 operator.

*3-address is represented in 3-ways. They are

1. quadruples
2. Triples
3. Indirect Triples

1. Quadruples:-

*It contains 4 fields. i) operator(op) ii) Argument 1 (Arg1) iii) Argument 2 (arg2) iv) result

Ex:- $a=b*-c+b*-c$ represent in Quadruple Expression.

$t_1 = -c$

$t_2 = -c$

$t_3 = b * t_1$

$t_4 = b * t_2$

$t_5 = t_3 + t_4$

	op	arg1	arg2	Result
(0)	-	c		t_1
(1)	-	c		t_2
(2)	*	b	t_1	t_3
(3)	*	b	t_2	t_4
(4)	+	t_3	t_4	t_5

Disadvantages of Quadruples:-

*In the above example t_1, t_2, t_3, t_4, t_5 are temporary variables. These temporary variables are stored in symbol table. If there are so many temporary variables generated it requires more amount of memory. To overcome this disadvantage triples are used.

2. Triples:- It contains 3 fields.

i) operator (op) ii) Argument 1 (arg 1) iii) Argument 2 (arg 2)

Ex:- $a=b*-c+b*-c$

	op	arg1	arg2
(0)	-	c	
(1)	-	c	
(2)	*	b	(0)
(3)	*	b	(1)
(4)	+	(2)	(3)

*So, Here in Triples temporary variables are not required.

*So, it requires less amount of memory we can execute the instructions.

3. Indirect Triples:-

*A pointer table is used in addition to Triple table.

pointer	Triple address
101	(0)
102	(1)
103	(2)
104	(3)
105	(4)

*Another implementation of 3-address code is that listing of pointers to triples, rather than listing the triples themselves. This implementation is called indirect triples

SDT to convert Binary Number into a Decimal Number:-

1) 11.01

$.01 = 0 \times 2^{-1} + 1 \times 2^{-2} = \frac{1}{2^2} = 1/4$

2) 01.11

$.11 = 1 \times 2^{-1} + 1 \times 2^{-2} = \frac{2+1}{2^2} = 3/4$

3) 01.111

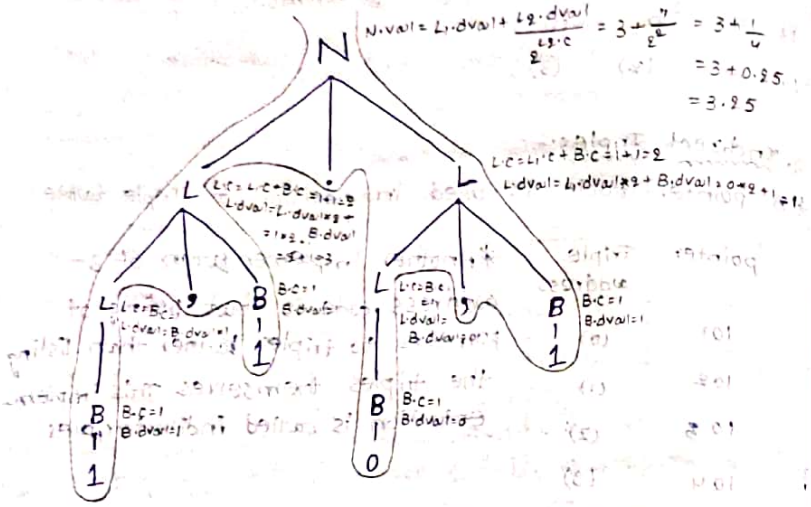
$.111 = 1 \times 2^{-3} + 1 \times 2^{-2} + 1 \times 2^{-1} = \frac{4+2+1}{3^2} = 7/9$

Productions	Semantic Rules/Actions
$N \rightarrow L_1 . L_2$	$\{N.dval = L_1.dval + \frac{L_2.dval}{2^{L_2.c}}\}$
$L \rightarrow L_1 B$	$\{L.c = L_1.c + B.c;\ L.dval = L_1.dval * 2 + B.dval;\}$
$\mid B$	$\{L.c = B.c;\ L.dval = B.dval;\}$

B → 0 {B.c = 1; B.dval = 0;}

/1 {B.c = 1; B.dval = 1;}

11.01 = 3.25



SDAT for Simple type Declaration:-

D → TL {L.inh = T.type}

T → int {T.type = integer}

T → float {T.type = float}

L → L1, id {L1.inh = L.in
addType (id.entry, L.inh)}

L → id {addType (id.entry, L.inh)}

*Here D represents a declaration which consists of a type T followed by a list L attributes.

*T has 1 one attribute T.type which is the type in declaration in D.

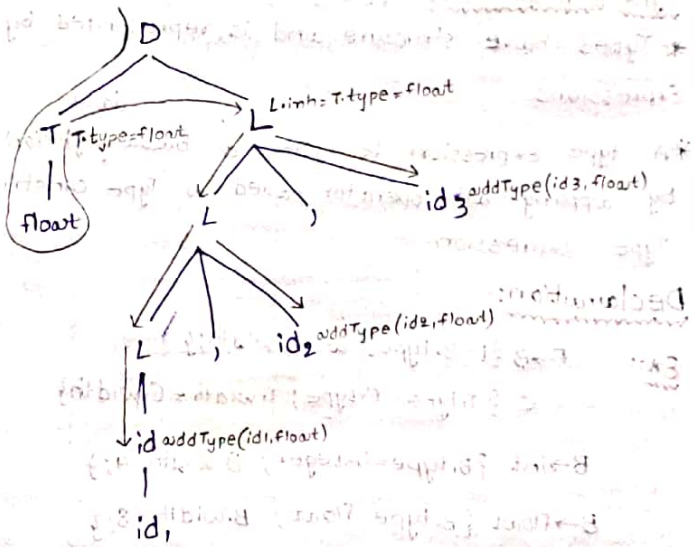
*L also has 1 attribute inh, to emphasize that it is an inherited attribute. The purpose of L.inh is to pass the declared type down the list of identifiers. So, that it can be added to the appropriate symbol table entries.

*A function add type is called with two arguments

1) id.entry, a lexical value that points to a symbol table object.

2) L.inh, the type is assigned to every identifier on the list.

float id1, id2, id3



Types and Declarations:-

*The actual storage for a procedure call (or) an object is allocated at run time.

*The applications of types can be grouped under checking and Translation

Type checking:-

*Type checking uses rules to reason about the behaviour

of a program at run time. It ensures that the types of the operands match the type expected by the operator.

Ex: && operator. This expects it's two operands to be Boolean. The result is also of type Boolean.

Translation:-

* From the type of a name, a compiler can determine the storage that will be needed for that name at run time.

Type Expressions:-

* Types have structure and is represented by using type Expressions.
* A type expression is either a Basic type (or) is formed by applying an operator called a Type constructor to a Type expression.

Declaration:-

Ex:- T → B { t = B.type; w = B.width; }
C { T.type = C.type; T.width = C.width }

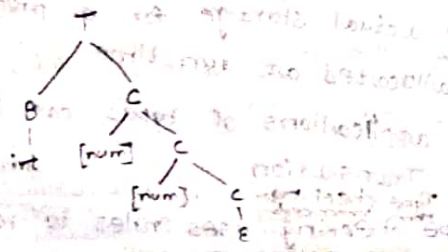
B → int { B.type = integer; B.width = 4; }

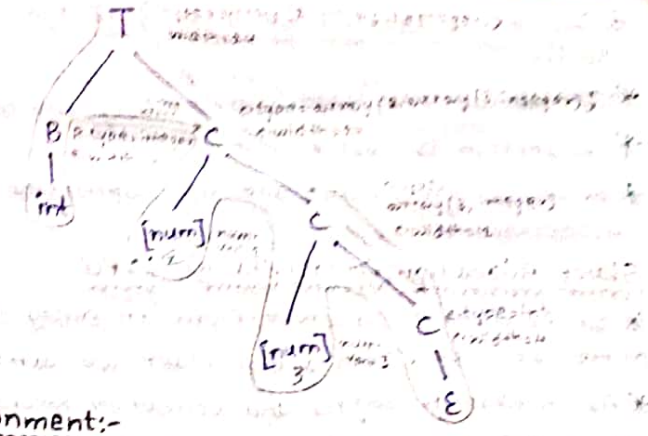
B → float { B.type = float; B.width = 8; }

C → ε { C.type = t; C.width = w; }

C → [num] C1 { array (num.value, C.type);
C.width = num.value × C1.width; }

array int[2][3]

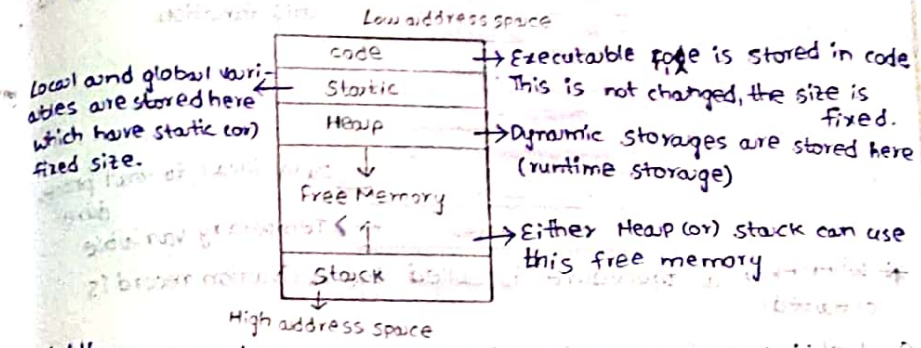




Runtime environment:-

* The compiler demands a block of main memory to the OS this memory is utilized for executing the compiled program. This block of memory is called Runtime Storage. Runtime Storage is divided into 4 parts:

Storage Organization:-



* When procedures are called this stack is used.
* Size of stack and heap is not fixed it may grow (or) shrink during program execution.
* stack is used to store DataStructures called Activation records gets created during procedure calls.

Storage Organisation strategies:-

→ Static allocation
→ stack allocation
→ Heap allocation

Static Allocation:-

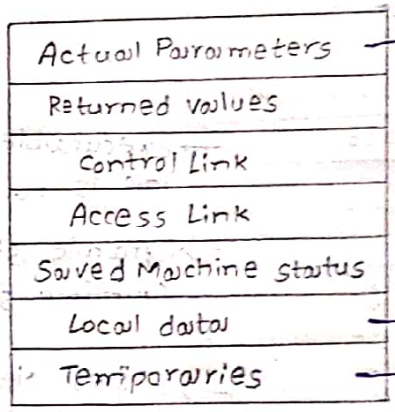
* Here disadvantage is we have to say the size in precisely (or) in advance.

* If we declare small size it is not possible to increase it.
* If we declare large size memory will be wasted.
* Insertion is costly.
* Recursive procedures are not supported by this static Allocation.

Stack Allocation (Control Stack) "LIFO"

* It is a data structure. From an empty stack we can't delete an item. Into a full stack we can't add an item.
* As Activation begins the activation records are pushed on to the stack and on completion the record is popped out.
* Here memory addressing is done using pointers for index registers hence slow than static allocation

Activation record:-



* Whenever a procedure is called the Activation record is created.

Control Link:- It is optinal field. It points to Activation record of calling procedure. This link is also called as dynamic link.

Access Link:- It may be needed by the called procedure but found else where in another activation record.

Heap allocation:-

* Heap allocation eliminates the problem of stack Allocation i.e. in Heap allocation retaining of Activation record is done.

Ex:- Factorial of a number