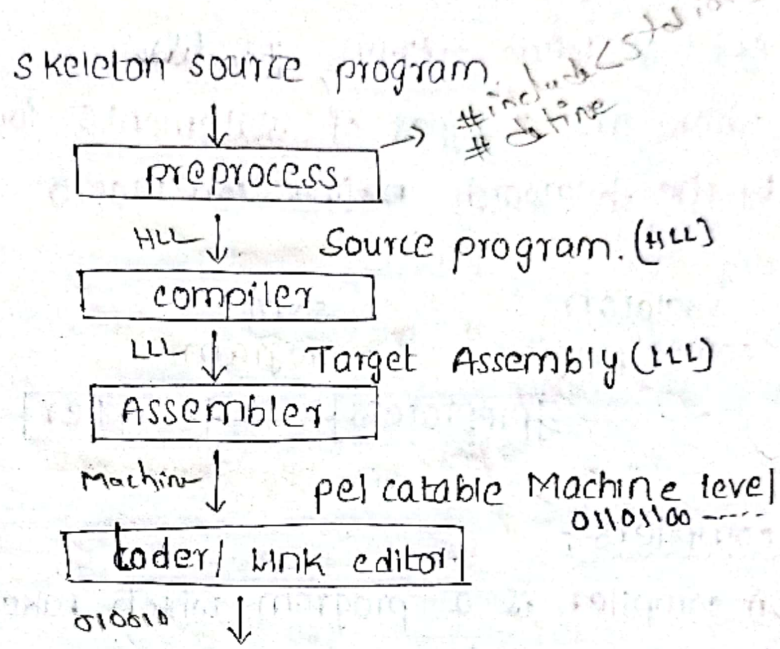
unit-1

Introduction to the compilers:

To create a executable form of your source code we may require several programs. these programs work together to produce an executable target program.

The process of execution of program is



The compiler takes a source program written in a highlevel language as an input and converts it into a target assembler language.

The assembler then takes this target assembler code as input and proceduces a relocatable Machine code as an output then the task of Loading and Link editing is called to perform the loading of the program into the memory and proper locations and link editor links the object modules and prepares the single Module from several Modules.

preprocess :-

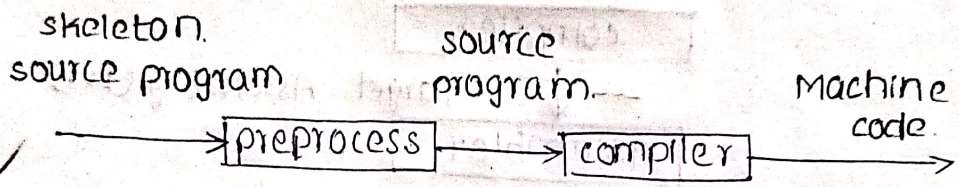
1) The output of the preprocess may be given as input to compilers.

2) preprocess allow users to access Macros in the program. Macro Means a set of instructions which can be repeatedly in the program

3) preprocess also allow users to include header file which may required to program.

ex: # define π(3.14), #include

There are 2 types of statements for Macros is given by the keyword define (or) Macro

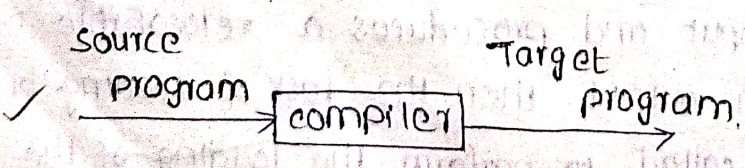


compilers :-

1) compiler is a program which takes first language (source program) as input and translates it into an equivalent another language (target program)

2) during the process of translation if some errors are encountered then compiler displays them as error mess -ags.

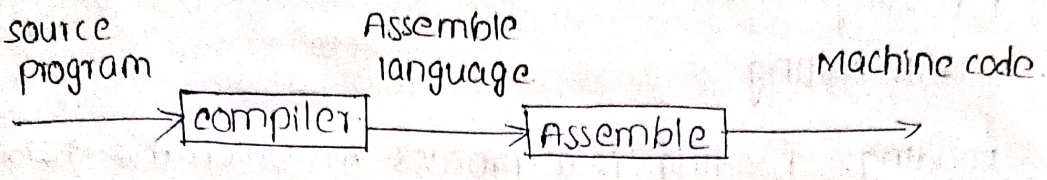
3) The model of Basic compiler represented as



The compiler takes source program as high level language (pascal, c, fortune) and converts into lowlevel (or) Machine level language such as Assemble language Assembly to Machine level language.

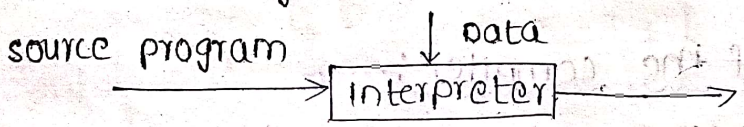
Assembler :-

Some compiler are produce Assemble code as output which is given to Assembler as input. The Assembler is a kind of translates which takes the Assembler program as output and produces.



Interpreter :-

An interpreter is a kind of translate which produces the result directly when the source program and data is given to it as input. It doesnot produce objective code but each time program needs executio -n language needs SNOBOL, LISP, can be translated using interpreter.



direct execution

The process of interpretation can be carried out in different phases 1. LEX Analysis 2. syntax Analysis 3. Symmetical Analysis 4. Direct execution.

Advantages :-

1) Modification of user program can be easily made and implemented as the execution process

2) The type of object that denotes variable may change dynamically.

3) Debugging a program and finding a error is a simplified talk.

4) The Interpreter for the language makes a Machine independent.

Disadvantages :-

1) The execution is slower.
2) Memory consumption is more.

Loaders and Linkers :-

Loaders is a program which performs two functions

1) Loading
2) Link editing

Loading :- Loading is a process in which the relocatable machine code is read and the relocatable addresses are alter. then the code with alter instructions and the data is placed in the memory at proper locations

Link editing :- The job of link editor is to make a single program from several files. if code in one file refers the location in another file then such a reference is called External reference.

* Structure of the compiler :-

The compilation can be done in 2 parts.

1) Analysis
2) synthesis part

* Analysis :- In Analysis part the source program read and broke it down into pieces. the syntax of the meaning of source string to determine and then an intermediate code is generated from the source code.

synthesis part :- In synthesis part the intermediate form of the source code is taken and converted into an equivalent target program during this process if certain code has to be optimized for efficient Execution.

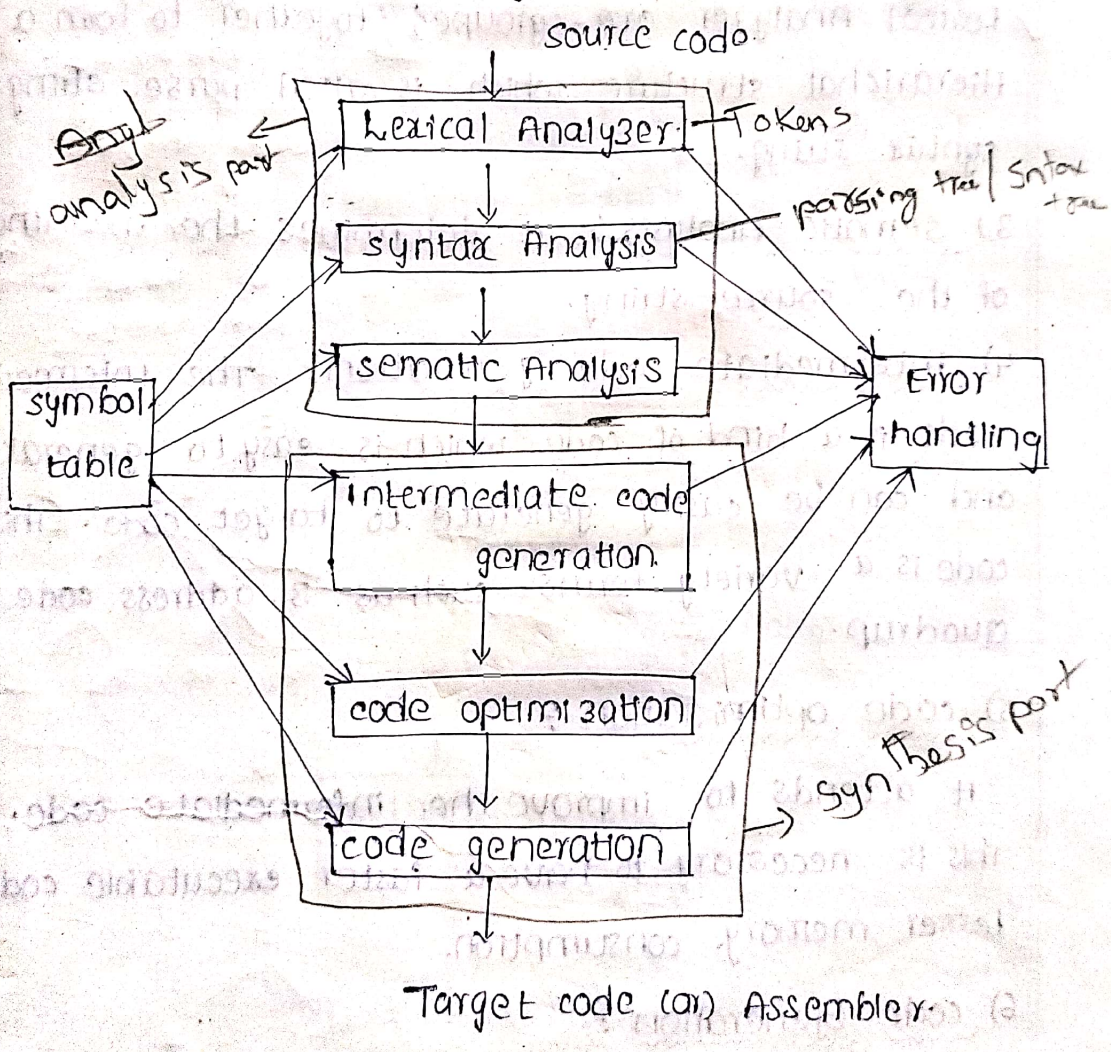
The Analysis part is carried out into 3 parts.

1) Lexical Analysis :- The source program is read and broken into stream of tokens. strings which are called tokens.

2) Syntax Analysis :- The tokens are arranged in a Hierarichal structure that helps for finding the syntax of the source string.

3) sematic Analysis :- The meaning of the source string is determined.

phases of the compiler :-



1) Lexical Analysis :- It is also called scanning. the complete source code is scanned and it broken up into group of strings called tokens. The token is a sequence of characters having a collective meaning

Ex:- total = count + rate * 60

* The blank characters which are used in the programming statements are eliminated during lexical analysis.

2) Syntax Analysis :- The syntax analysis is also called parsing. In these phase the tokens are generated by Lexical Analyzer are grouped together to form a Hierarichal structure which is called parse string (or) syntax string.

3) sematic Analysis :- It determines the meaning of the source string.

4) Intermediate code generation :- The intermediate code is a kind of code which is easy to generate and can be easily generate to target code. This code is a variety forms such as 3 address code, Quadrup.)

5) code optimization :-

It attends to improve the intermediate code. This is necessary to have a faster executable code (or) lesser memory consumption.

6) code Generation :-

In this phase, the target code gets generated the intermediate code is translated to the machine instructions.

Ex:- total = count + rate * 60

(total → Identifier; = → Assignment operator; count → id; + → Arithmetic; rate → id; 60 → int | float)

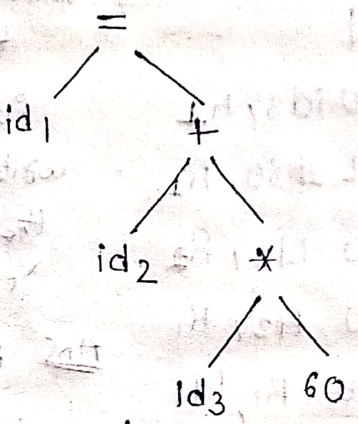
[Lexical Analyzer]

↓

$id_1 = id_2 + id_3 * 60$

[syntax Analyzer]

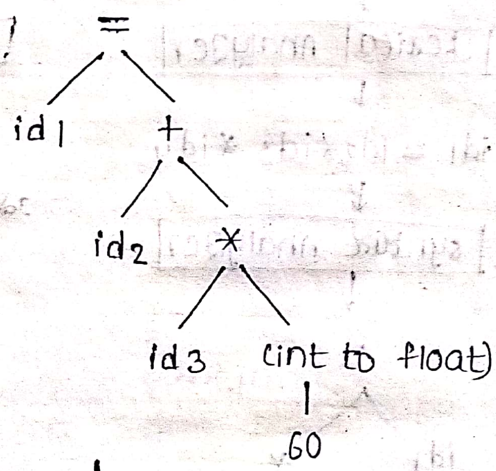
↓



↓

[sematic Analyzer]

↓



↓

[intermediate code generation]

↓

t_1 = int to float(60)

$t_2 = id_3 * t_1$

$t_3 = id_2 + t_2$

$id_1 = t_3$

↓

code optimizer

↓

$t_1 = id_3 * 60$

$id_1 = id_2 + t_1$

↓

code generation

↓

```
MOV id3, R1
MUL #60, R1
MOV id2, R2
ADD R2, R1
MOV R1, id1
```

Ex-2 :- a = b * c * d.

a = b * c * d

↓

Lexical Analyzer

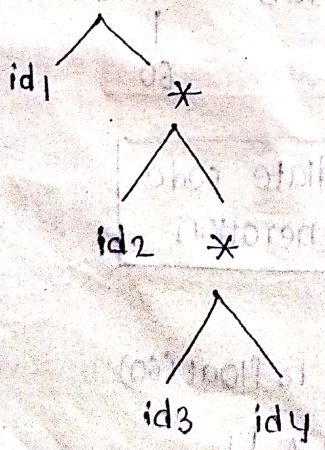
↓

$id_1 = id_2 * id_3 * id_4$

↓

syntax Analyzer

↓



↓

Ex Describe the o/p for the various phases of compiler with respect to following statements:

position = initial + rate * 60

Ex consider the following format of 'c' code:

```
bloat i, j;
i = i * 70 + j + 2;
```

write the o/p of all phases of the compiler for the above 'c' code

Ans i = i * 70 + j + 2.

↓

Lexical analyzer

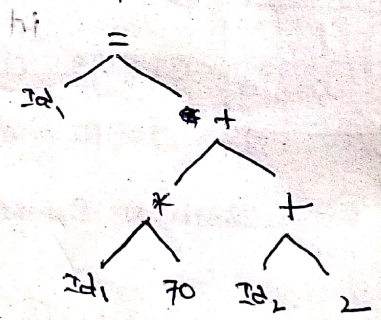
↓

$id_1 = id_1 * 70 + id_2 + 2$

↓ token stream

Syntax analyser

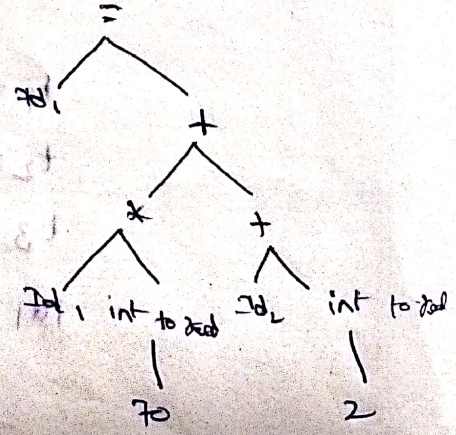
↓



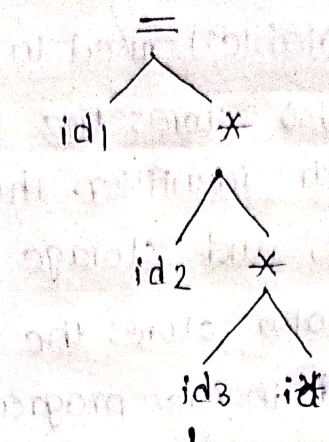
↓

Semantic analyser

↓



semantic analysis



intermediate code generation

$t_1 = id_3 * id_4$

$t_2 = id_2 * t_1$

$id_1 = t_2$

code optimization

$t_1 = id_4 * id_3$

$t_2 = id_2 * t_1$

code generation

```
MOV id3, R2
MUL id4, R2
MOV id2, R1
MUL R2, R1
MOV R1, id1
```

intermediate code generator

$t_1 = int\ to\ real(70)$

$t_2 = id_1 * t_1$

$t_3 = int\ to\ real(2)$

$t_4 = id_2 + t_3$

$t_5 = t_2 + t_4$

$id_1 = t_5$

code optimizer

$t_1 = id_1 * 70.0$

$t_2 = id_2 + 2.0$

$t_3 = t_1 + t_2$

$id_1 = t_3$

code generator

```
MOV id1, R1
MUL 70.0, R1
MOV id2, R2
ADD 2.0, R2
ADD R2, R1
MOV R1, id1
```

* Symbol table Management

To support the phases of the compiler a symbol table is maintained. The task of the symbol table is store identifiers (variables) used in the program.

* The symbol table also stores the information about the attributes of each identifier. The attributes such as time, scope, value, and storage capacity.

* The symbol table also stores the information about the subroutines used in the program. In case of subroutine the symbol table stores the name of the subroutine, number of arguments, type of the arguments, method of passing are (call by value, call by reference and return type).

* During compilation the lexical analyzer detects the identifiers and makes an entry into the symbol table.

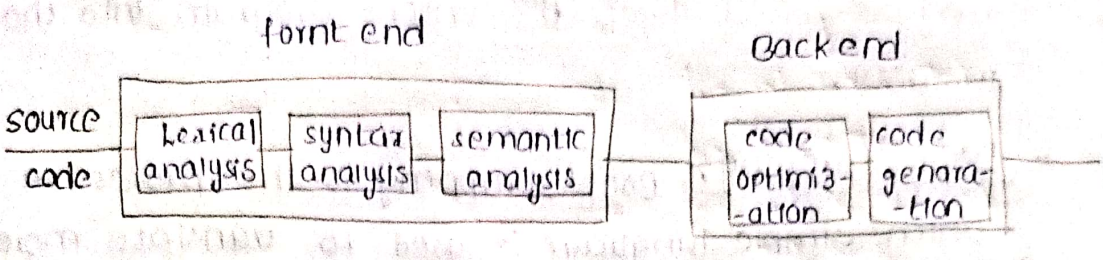
* varies phases can use the symbol tables in varies ways while doing the semantic analysis and intermediate code we need to know what type of identifiers.

* Error detection and handling :

In compilation each phase detects errors. These errors must be reported to error handler which is used to handle the errors so that the compilation can be produced.

large number of errors can be detected in syntax analysis such errors are called syntax errors. During semantic analysis type mismatch kind of error is usually detected.

Grouping of phases :-



Different phases of the compiler can be grouped together to form the front end and backend. The front end consists of phases primarily dependent on the source language and independent on the target language. The front end consists of the analysis part.

The back end consists of phases dependent on the target language and it is independent on the source language.

By keeping same front end and different backends here, produce a compiler for the same source language on different machines.

By keeping different front ends and same Backend-s one can compile different language on the same machines.

pass :-

one complete scan of the source language is called pass. It includes reading an input file and writing an output file.

Many phases grouped by using one pass.

It is difficult to compile source code into

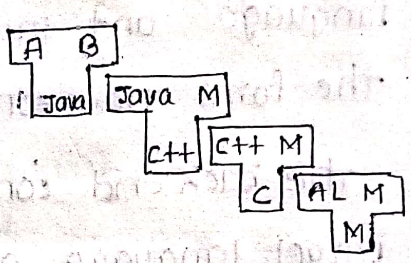
if we group several phases into one pass we may be forced keep the entire program into the memory.

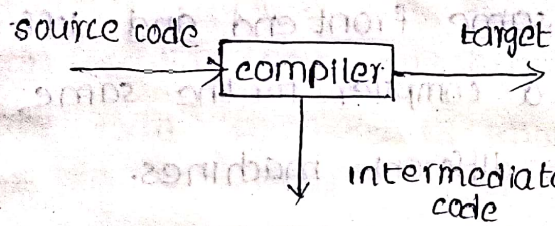
* **Boot strapping:-** Boot strapping is the process in which simple language is used to translate more complicated program which intern can handle more complicated programs and so on.

* writing a highlevel language is a complicated process it takes a lot of time to write a compiler.

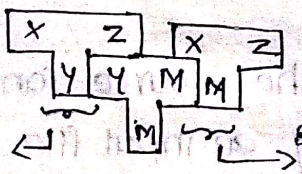
* Boot strapping is represented as 'T' diagram it consists of '3' languages

1) source language.
2) Target language.
3) Implementation language.





* To understand the Boot strapping technique the method is called cross compiler.



Tewo language must be same

2 languages must be same

suppose we want to write a cross compiler for a new language 'x' the implementation language of this is 'y' and the target code generated in the language.

↓ compiler

is 'z'

Fix. existing compiler 'y' runs on machine 'M' and generates code for machine 'm'.

if we run x, y, z i.e x_y^z using y_M^M then we get a compiler x_z^M that means a compiler a source language 'x' generates the target code in 'z' and which runs on the machine 'M'.

Lexical analysis: It is the first phase of the compiler. The process of compilation starts first phase called LA in this phase the input is scanned completely to identify the tokens. so the lexical analysis is also called as scanning. The token structure can be represented with the help of some diagrams. These diagrams are probably known as finite automata.

* To construct finite automata regular expressions are used. These diagrams translated into programs to identify the tokens
* The lexical analysis separates the input 'C' program into various types of tokens types like, keywords, identifiers and operators and so on.

Ex:
```
int main()
{
int i;
  for(i=0; i<10; i++)
  {
   printf("Hello world\n");
  }
}
```

Lexemes	Tokens
int	keyword
main	identifier
(	operator
)	closing bracket
{	symbol
int	keyword
i	identifier
for	keyword
printf	identifier
}	symbol
;	symbol

* Functions of Lexical analysis

1) It is
2) It generates symbol table which is a type of data structure to store symbol related information.
3) It can count line numbers of the source code.
4) It eliminates black spaces, white space and comments
5) it reportes the error encountered while generating the tokens.
6) It produces the stream of tokens.

* Tokens Lexemes and patterns:

Lexem is a group of characters, token is a class or categorical like keyword, identifiers, operators and some constants.

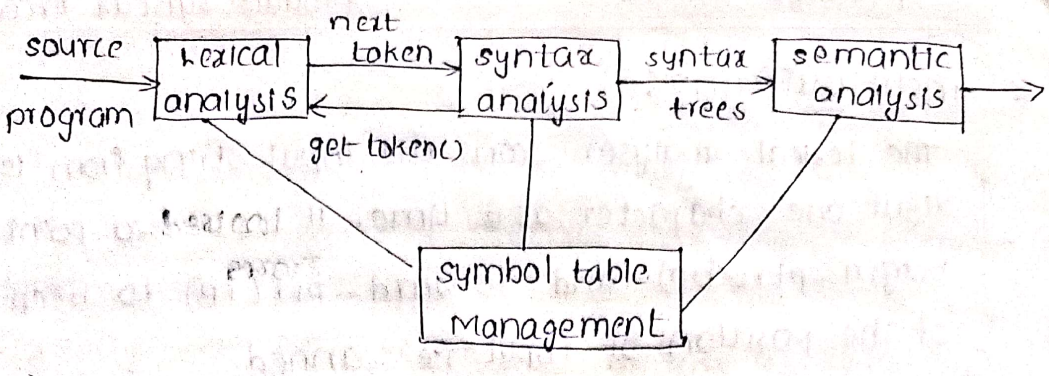
pattern is a set of rules to describe the token which is like a a to z, A to Z, 0 to 9 digits.

* lexical errors :

These type of errors can be detected during lexical analysis. some of the errors are :

1) Exceeding the length of the identifier. for example in the length of the identifier is limited to '10' characters.
2) Appereance of illegal characters.
   ex: printf("Hello world"); $
3) unMatched string
   ex: This is a comment */

* Role of lexical analysis :



The process of compilation uses lexical analysis has the first phase of the compiler. The lexical analysis bought the request from the syntax analysis as get token(), then syntax analysis gives responce to the lexical analysis as next token().

Lexical analysis produce the tokens and syntax analysis consume the tokens so it is called as producer-cosumer pair. These two phases also related to the symbol table. in which the maintains about the identifiers to store and recreates.

some times the errors may be generated in the lexical analysis which are called as lexical errors and one handled by the error handler.

* Differences between lexical analysis and syntax analysis

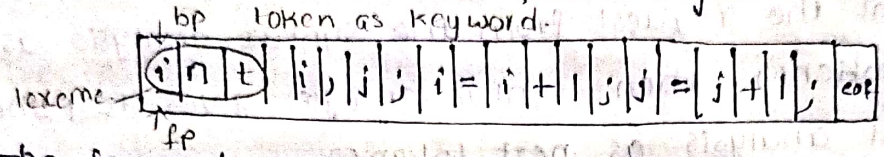
Lexical analysis	syntax analysis
1) It is the first phase of the compiler.	1) It is the second phase of the compiler.
2) Lexical analysis takes the input as source program and produces output as tokens	2) syntax analysis takes the input as tokens and produce output as syntax trees.
3) The error produced as lexical errors	3) The errors are produced as syntax errors
4) It uses finite automata and regular expression to	4) It uses the grammars and hierarchical structures to

produce tokens	produce syntax trees.

* Input Buffering :-

The lexical analyser scans the input string from left to right one character at a time. It uses two pointers begin-ptr(bp) and forward-ptr(fp) to keep track of the position of input i.e scanned.

Initially both the pointers are pointing to the one(a) first character of the input string as



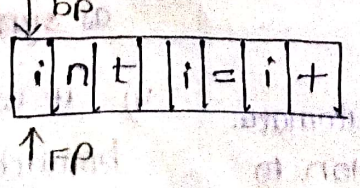
The forward pointer moves a head to search for the end of the lexeme.

* As soon as the blackspace is encountered it indicates the end of the lexeme and lexeme int is identified.
* The FP will be moved a head at white space when FP encounters a white space it ignores and move a head.
* Then both begining and forward pointer are set at next next token i.
* The input character is read from the memory is very costly. Hence we use two bufferring techniques

1) one buffer scheme.
2) Two buffer scheme.

1) one buffer scheme :-

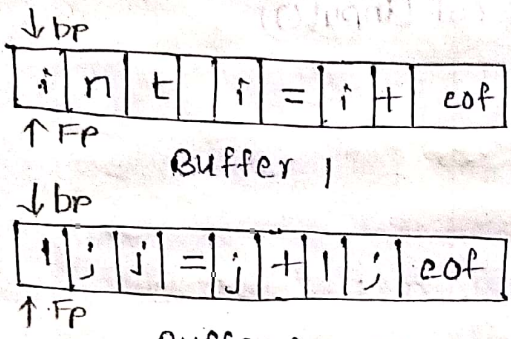
In these one buffer scheme only one buffer is used to store the input string. if lexeme is very long, it crosses the buffer boundary to scan the rest of the lexeme. The buffer has to be refilled it that makes the overwriting the first part of the lexeme.



2) Two Buffer scheme:-

To overcome the problem of the one Buffer scheme, two Buffers are used to store the input string. the first buffer and second buffer scan alternatively. when end of the current buffer is reached the other is buffer is filled.

* Initially both bp and Fp are pointing to the first character of the first buffer.



* Then Fp moves towards right in search of end of the lexeme, As soon as blank character is recognized. the string between. begining pointer and fp is recognized.
* To identify the boundary of the first buffer end of the buffer (eof) ↓ character should be placed at the end of the first buffer. Similarly end of the second buffer is also recogni-zed with (eof) at the end of the second buffer.
* when Fp encounters first eof it recognized end of the first of buffer and filling of the second buffer is started.
* Alternatively both the buffers can be filled up until end of the input program and stream of tokens is identified.
* This eof introduced at the end is called "sentinel"

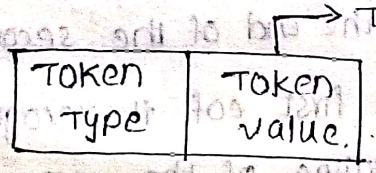
Code for input Buffereing:-

```
if FP== eof(buffer1)
{
  FP++;
else if FP== eof(buffer2)
  {
    fp++;
  }
else if FP== eof(input())
  {
    return;
  }
else
    {
      fp++;
    }
```

* Recognization of tokens:-

For a programming languages there are various types of tokens such as identifiers keywords constants

* The token is usually represented by a pair, token type and token value.



* The token type tells us the category of token. and the token value gives us the information about the token and it also called as token attribute.

* During lexical analysis process the symbol table is maintained. The token value can be a pointer to the symbol table. Because of :

- In case of identifiers and constants it

* The lexical analyser reads the input program and generates the symbol table for tokens.

we will consider the encoding of the program.

Token	code	value.
if	1	–
else	2	–
while	3	–
for	4	–
identifier	5	ptr to symbol table
constant	6	ptr to symbol table
<	7	1
<=	7	2
>	7	3
>=	7	4
!=	8	5
(	8	1
)	8	2
+	9	1
-	9	2
=	10	1

The corresponding symbol table for identifiers

Location counter	Type	value.
100	identifier	a
⋮		
105	constant	10
⋮		
107	identifier	i
⋮		
110	constant	2

program.

```
if(a<10)
    i=i+2;
else
    i=i-2;
```

1, (8,1) (5,100) (7,1) (6,105) (8,2)
(5,107) (10) (5,107) (9,1) (6,110)
2
(5,107) (10) (5,107) (9,2) (6,110)

* In the above example scanner scans the input string and recognise the 'if' as keyword and returns token type '1'. since '1' indicates keyword 'if'. and hence 'i' is at the begining of the token string

* Next pair (8,1) where '8' indicates parathesis and '1' indicates opening paranthesis then we can scan the input 'a'. it recognises it has identifier and searches the symbol table to check weather the same entries is present

* if not it inserts the information about this identifier in symbol table and returns 100.

* if the same identifier or the variable is already present in the symbol table then lexical analyzer does not insert it into the table instead it returns the location where it is present.

* Specification of tokens :-

For specifying the tokens we use the following :-

1) Alphabets
2) strings
3) languages
4) operations the languages
5) Regular expressions
6) Regular grammar.

Description :-

1) Alphabet is a symbol or a set of symbols and it is represented as Σ i.e $\Sigma = \{0, 1 \ldots\}$ (or) $\Sigma = \{a, b, c, \ldots\}$

2) string is defined as a set of alphabets

$\Sigma = \{a, aa, ab, baa\}$

i) the types of the strings are empty string, prefix string, string length $-|w|$, suffix string, substring, string concatenation.

ii) length of the string means to count number of characters $w = 000$, $|w| = 3$

iii) Empty string means string does not have any characters $s = 0$

iv) prefix string means starting characters of the string.

v) suffix string means last characters of the string.

vi) substring means removes the both suffix and prefix of the string. which is given.

vii) string concantenation means concantaing two strings

3) Language is a finite number of strings.

4) operations on the languages. [union, concantenation] i.e ∪, •

K en closure (*) which represents 0 or more

positive closure (+) which represents 1 or more.

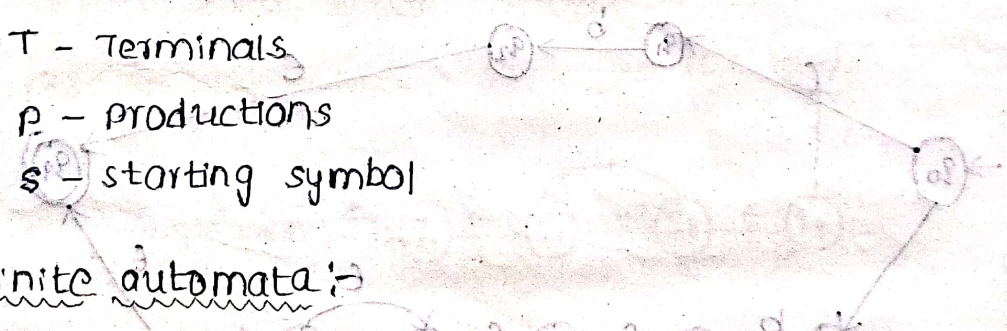
5) Regular expressions are used to defined as tokens and also defines the grammar i.e regular which is represented as $G_1 = (V, T, P, S)$

V – variables

T – Terminals

P – productions

S – starting symbol



* Finite automata :-

* From the regular expressions it is used to (or) possible to build a finite automata.

* the finite automata is usually a non-determinitic finite automata and is represented by the 'F' (or) M. By using the transition diagram.

The represents as follows :-

For $M = \{Q, \Sigma, \delta, q_0, F\}$

Q – finite number of states $\{q_0, q_1, q_2 \cdots\}$

Σ – Input alphabet $\{0, 1 \cdots\}$

δ - Transition function. - $\{\delta = (Q, \Sigma) \rightarrow Q\}$

q_0 - initial state.

F - final state.

* construct on NFA with ϵ-moves for the regular Expression b+ba*

Soln

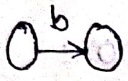
Given, regular expression (R.E) = b+ba*

Let, $r_1 = b$

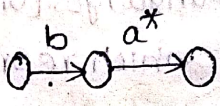
$r_2 = ba^*$

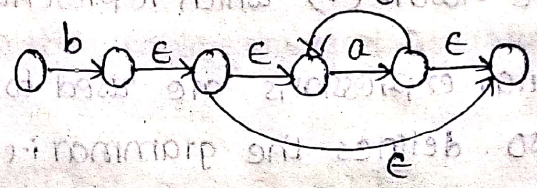
$r_3 = r_1 + r_2 = b + ba^*$

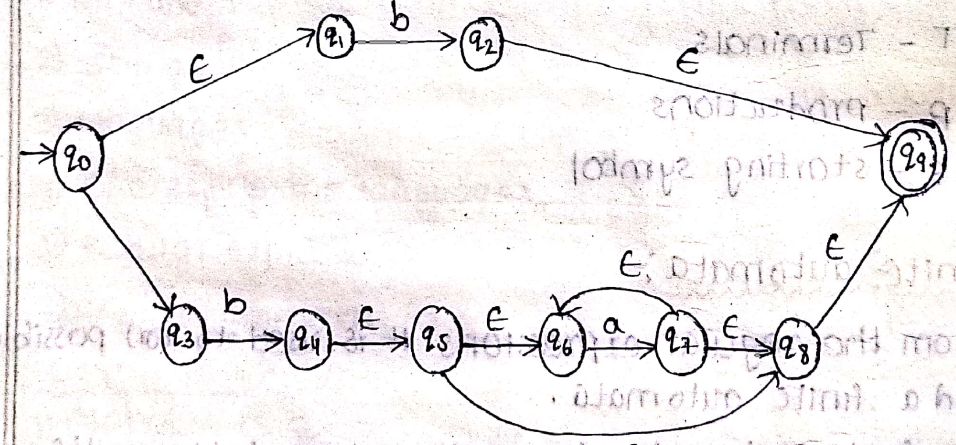
$r_1 = b$



$r_2 = ba^*$







* construct an NFA with ϵ-moves for the R.E is (0+1)*

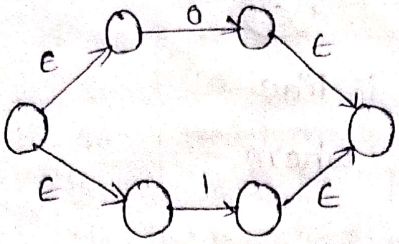
Soln

Given, R.E = (0+1)*

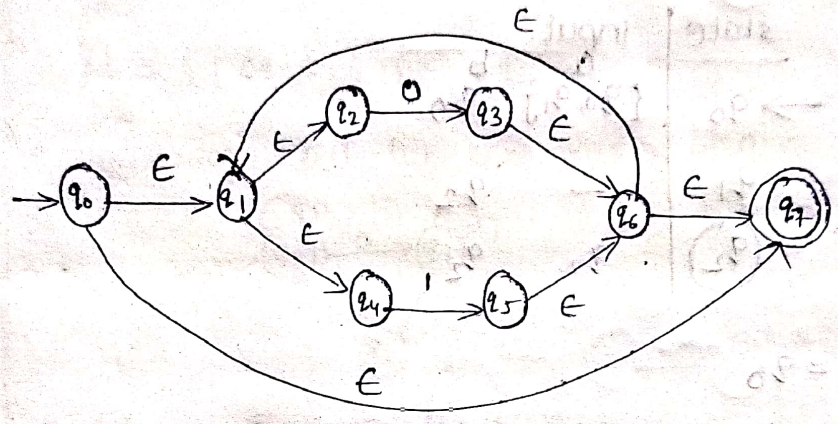
$r_1 = 0+1$

$r_2 = (0+1)^*$

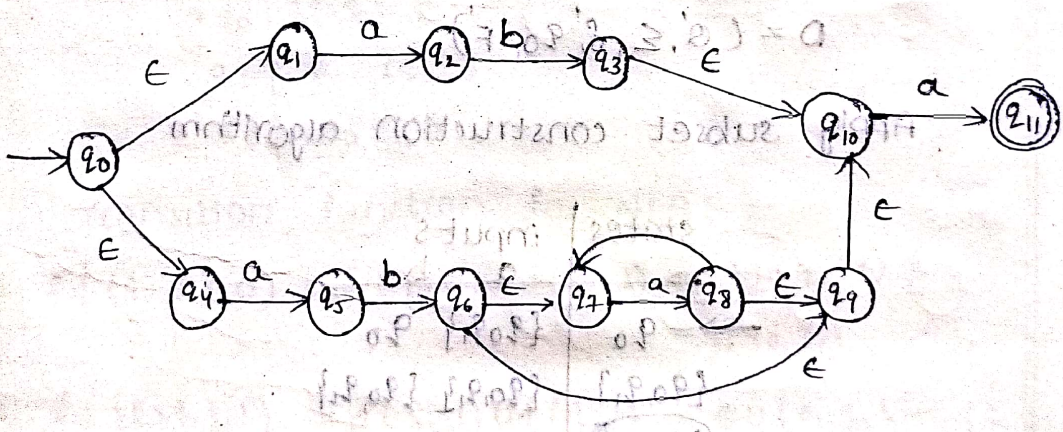
$r_1 = 0+1$



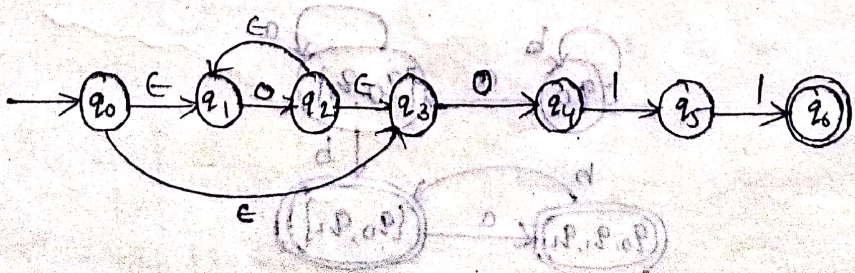
$r_2 = (0+1)^*$



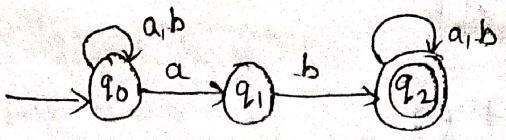
* construct an NFA with ϵ for the R.E is (ab∪aba*)a



* construct (0|1)*011 for these Regular expression draw the NFA with ϵ.



* Construct DFA from the given NFA



Sol:- The given NFA N' is like

$N = \{Q, \Sigma, \delta, q_0, F\}$ where

$Q = \{q_0, q_1, q_2\}$

$\Sigma = \{a, b\}$

$\delta =$

state	input a	input b
→ q_0	$\{q_0, q_1\}$	q_0
q_1	—	q_2
← q_2	q_2	q_2

$q_0 = q_0$

$F = \{q_2\}$

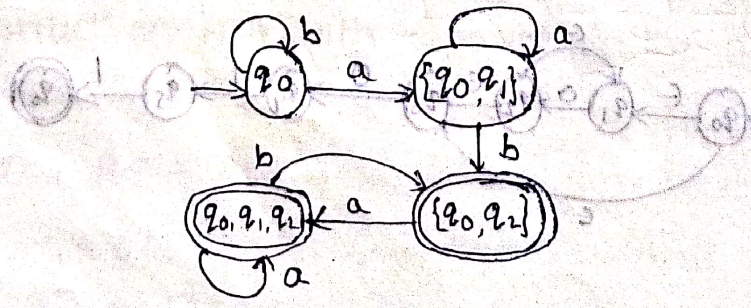
Let us consider the DFA D' is like

$D = (Q', \Sigma, \delta', q_0, F')$

Apply subset construction algorithm

states	inputs a	inputs b
→ q_0	$\{q_0 q_1\}$	q_0
$\{q_0, q_1\}$	$\{q_0, q_1\}$	$\{q_0, q_2\}$
← $\{q_0, q_2\}$	$\{q_0, q_1, q_2\}$	$\{q_0, q_2\}$
← $\{q_0, q_1, q_2\}$	$\{q_0, q_1, q_2\}$	$\{q_0, q_2\}$

∴ the DFA is



$Q_1' = \{ q_0, \{q_0, q_1\}, \{q_0, q_2\}, \{q_0, q_1, q_2\} \}$

$\Sigma = \{a, b\}$

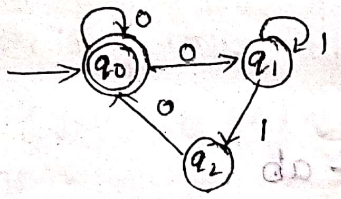
δ' =

states	input a	input b
q_0	$\{q_0, q_1\}$	q_0
$\{q_0, q_1\}$	$\{q_0, q_1\}$	$\{q_0, q_2\}$
$\{q_0, q_2\}$	$\{q_0, q_1, q_2\}$	$\{q_0, q_2\}$
$\{q_0, q_1, q_2\}$	$\{q_0, q_1, q_2\}$	$\{q_0, q_2\}$

$q_0 = q_0$

$F' = \{\{q_0, q_2\}, \{q_0, q_1, q_2\}\}$

* construct DFA for the given NFA



Transition table for NFA :-

	0	1
* q_0	$\{q_0, q_1\}$	ϕ
q_1	ϕ	$\{q_1, q_2\}$
q_2	q_0	ϕ

Transition function for NFA :-

$\delta(\{q_0, q_1\}, 0) = \delta(q_0, 0) \cup \delta(q_1, 0) = \{q_0, q_1\} \cup \phi$
$= \{q_0, q_1\}$

$\delta(\{q_0, q_1\}, 1) = \delta(q_0, 1) \cup \delta(q_1, 1) = \phi \cup \{q_1, q_2\} = \{q_1, q_2\}$

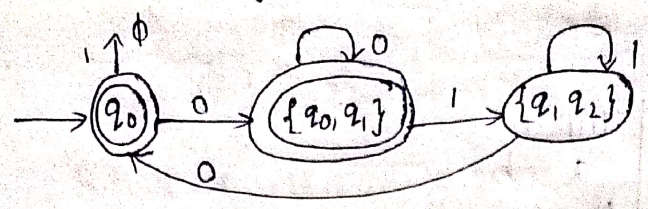
$\delta(\{q_1, q_2\}, 0) = \delta(q_1, 0) \cup \delta(q_2, 0) = \phi \cup q_0 = \{q_0\}$

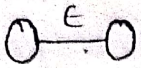
$\delta(\{q_1, q_2\}, 1) = \delta(q_1, 1) \cup \delta(q_2, 1) = \delta\{q_1, q_2\} \cup \phi = \{q_1, q_2\}$

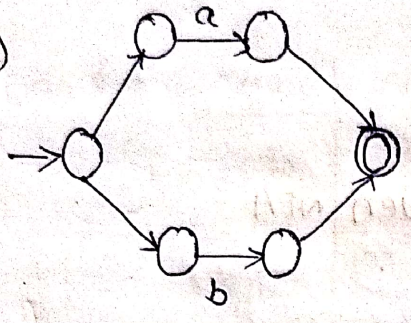
Transition table for DFA :-

	0	1
* → q_0	(q_0, q_1)	ϕ
* $\{q_0, q_1\}$	$\{q_0, q_1\}$	$\{q_1, q_2\}$
$\{q_1, q_2\}$	q_0	$\{q_1, q_2\}$

Transition Diagram for DFA :-



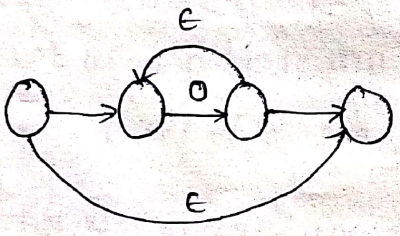
1) ∈  ∈

2) 

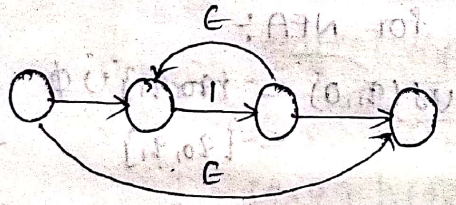
— a+b

3) 

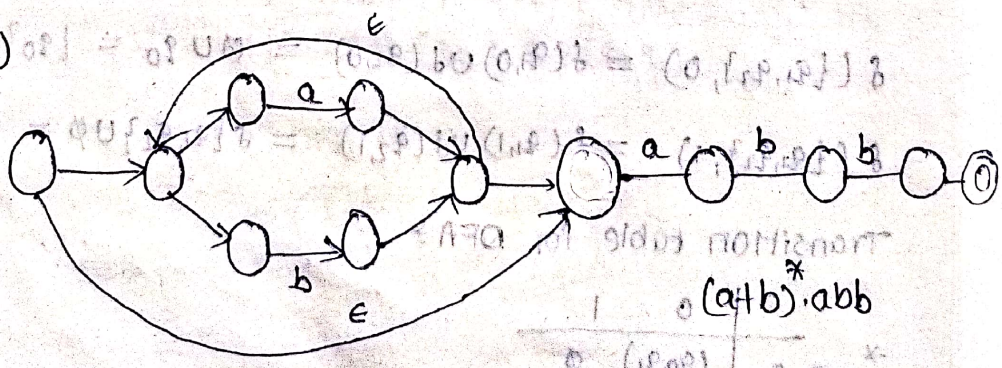
— ab

4) 

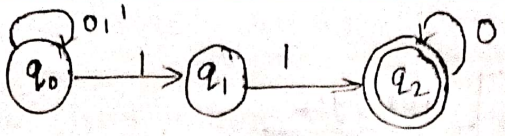
— 0^*

5) 

— 1^*

6) 

— $(a+b)^*.abb$



sol: Grammar = (V, T, P, S)

$q_0 \rightarrow 1q_1$ $q_1 \rightarrow 1q_2$ $q_2 \rightarrow 0q_2$

$q_0 \rightarrow 0q_0$ $q_1 \rightarrow 1$ $q_2 \rightarrow 0$

$q_0 \rightarrow 1q_0$ ∴ there are 7 grammar.

* lexical analyzer generator :- LEX

various tools have been built for constructing lexical analyzers using special purpose notations called regular expressions which are used in recognizing the tokens. A tool called LEX which is a unix utility which generates the lexical analyzer. A LEX is a very mush faster in finding the tokens.

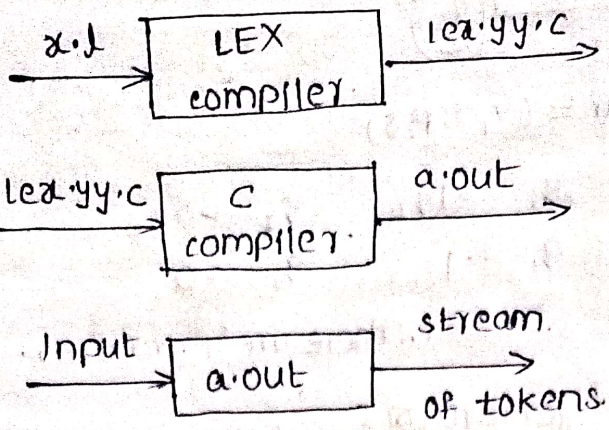
*the LEX specification file can be created using the extension (.l).

For example, the LEX specification file can be (x.l) which is given to the lex compiler which gives lex.yy.c which is a c program and is also a lexical analyzer program.

*The LEX specification file shows the regular expressions for the tokens and the lex.yy.c file consists of tabular representation of the transition diagram constructed for a regular expressions.

*In regular expression lexical LEX specification file actions of the pieces of 'c' code. which is directly carried over to the lex.yy.c. finally the compiler compiles the lex.yy.c and produces an object program a.out.

*when some input is given to the a.out then some sequence of tokens generated.



$ lex x.l
$ cc lex.yy.c
$ ./a.out

* Design of lexical analyzer LEX :-

The lex specification file consists of three different sections.

1) Declaration section.
2) Transition rule section.
3) Auxillary procedure section.

1) Declaration section :-

In this section declaration of variables constants can be done.

Some regular expression can also be done in the form of regular expression.

```
%{ int a,b;
   float count=0;
%}
```

2) Transition rule section :-

The rule section consists of regular expressions with associated actions.

These translation can be given as

```
%%
Rule1   {action1}
Rule2   {action2}
Rule3   {action3}
%%
```

3) Auxiliary procedure section :-

The third section is the auxiliary procedure section in which all require procedures are defined.

```
%{      %}
%%
"RAM" |
"RAVI" |    printf ("In noun");
"sing" |
"dance" |   printf ("In verb");
%%

main()
{
 yylex();
}
int yywrap()
{
 return1;
}
```

*In the above program, the third section consists of two functions

i) The main function and the yywrap function.

ii) The main function called the 'yy' which is the function that defined by the yylex i.e LEX.yy.c.

*First we compile the above program x.l using LEX compiler and then the LEX compiler will generate a ready 'c' program name LEX.yy.c

*This LEX.yy.c makes use of regular expressions and corressponding actions defining the [lex.l]. To run the LEX program LEX.l i.e x.l.

```
[localhost@CSE539] $ LEX x.l
[localhost@CSE539] $ lex.yy.c
```

[localhost@CSE539]$./a.out.

* Syntax analysis:- the second phase of compiler is syntax analysis. It is also called as parsing (or) parser. It checks the syntax of the language.

* The syntax analyzer takes the tokens from the lexical analyzer and groups them in such a way that some programming structure can be recoganized.

* After grouping the tokens syntax can't be recgonized, then syntax error will be generated.

praser:-

A parser is a process which takes the input string and produces a parse tree (or) generates the syntax errors.

* they are two important issues in parse string

1) specification of the syntax:-

-It means how to write any programming statement

* specifications should be presize and ambiguous should be in detail, should cover all the programic details of the programming languages, should complete such specification is called free grammar.

2) Representation of the input:-

* After passing, all the subsequent phases of the compiler takes the information from the parse tree being generated.

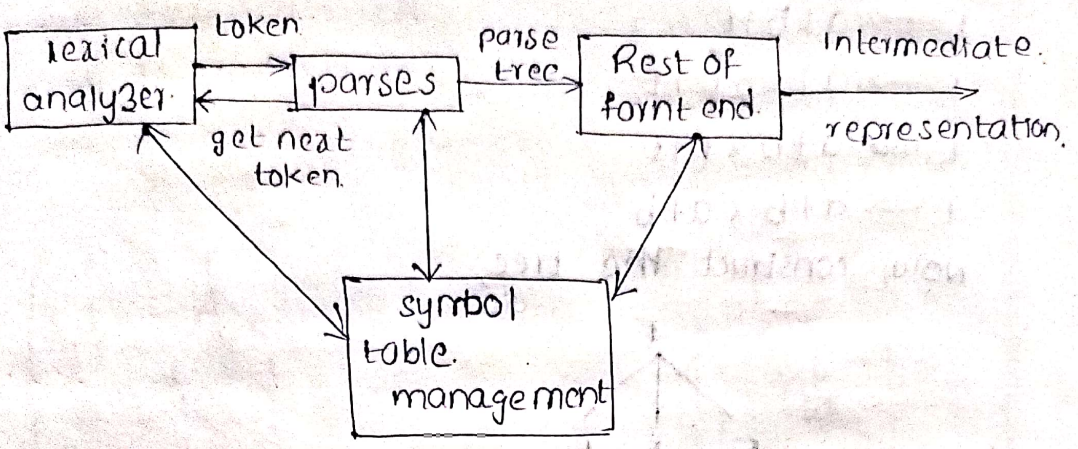
* There are two types of algorithms based on the parsing

1) Top-down parsing
2) Bottom-up parsing

1) Role of parser :-

In the process of compilation, the parse and lexical analyzer work together, when parser requires string

of tokens it involves lexical analyzer. the lexical analyzer supplies tokens to the syntax analyzer.



A grammar $G = (V, T, P, S)$ is CFG where the production rules can be formed. $V \rightarrow (VUT)^*$

Derivational parse trees:-

Derivation of the variables means generating strings. They are two types of derivations.

1) left most derivation.
2) right most derivation.

1) left most derivation:-

Replacing the left most side of the variable in the left hand side of tree is called left most tree.

2) right most derivation:-

Replacing the right most side of the variable in the right hand side of tree is called right most tree.

* Examples:- obtain the left most and right most tree.

derivation of a string $a+b*a+b$

$E \rightarrow E+E \mid E*E \mid E-E \mid E \mid a \mid b$ is the string.

Given string and grammar is

Sol:- $E \rightarrow E+E \mid E-E \mid E*E \mid E \mid a \mid b$ and.

the string is $a+b*a+b$.

a+b * a+b

E → E+E | E-E | E*E | E | a | b

E → E+E

E → E+E+E

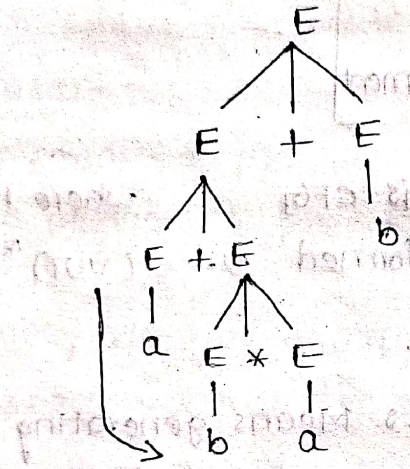
E → a+E+E

E → a+E*E+E

E → a+b*E+E

E → a+b*a+b

Now, construct the tree,



2) E → E+E | (E*E) | E*E | E | id and the string id+id*id both left and right most derivation.

3) S → (L) | a

L → L,S | S obtain left most and right most derivation

i) (a,a)

ii) (a,(a,a))

iii) (a,((a,a),(a,a)))

4) Consider the grammar.

S → aS | asbs | ∈ obtain the string aaabaab

5) S → aAB | bbA | ∈

A → aAb | ∈

B → bB | ∈

obtain the string aabb

2 Sol:

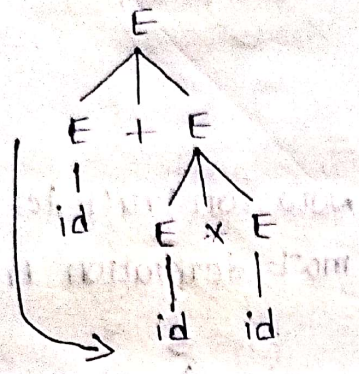
Given,

$E \rightarrow E+E$

$E \rightarrow id+E*E$

$E \rightarrow id+id*id$

Now, construct the left most derivation tree.



$E \rightarrow E+E$

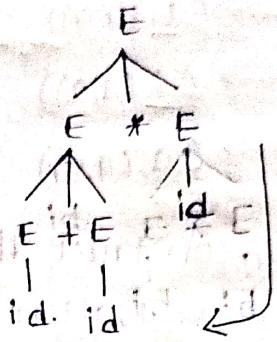
$E \rightarrow E+E*E$

$E \rightarrow E+E*id$

$E \rightarrow E+id*id$

$E \rightarrow id+id*id$

Now, construct the right most derivation tree



---

3 Sol: i) Given,

$S \rightarrow (L)$

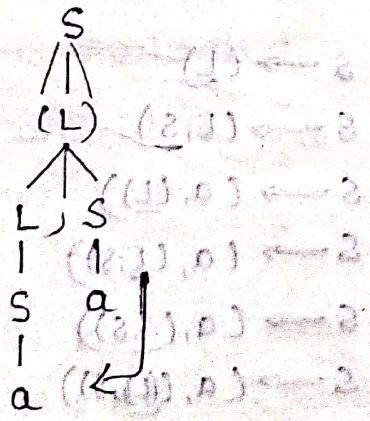
$S \rightarrow (L,S)$

$S \rightarrow (L,a)$

$S \rightarrow (S,a)$

$S \rightarrow (a,a)$

Now, construct the right most derivation tree.



$S \rightarrow (L)$

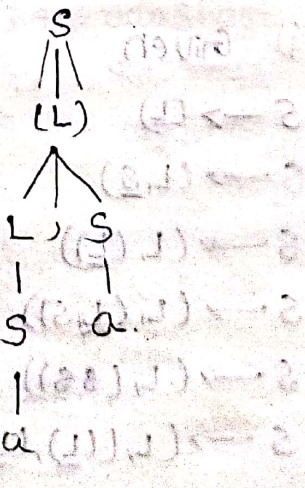
$S \rightarrow (L,S)$

$S \rightarrow (S,S)$

$S \rightarrow (a,S)$

$S \rightarrow (a,a)$

Now, construct the left most derivation tree.



ii) Given,

$S \longrightarrow (L)$

$S \longrightarrow (L,S)$

$S \longrightarrow (L,(L))$

$S \longrightarrow (L,(L,S))$

$S \longrightarrow (L,(L,a))$

$S \longrightarrow (L,(S,a))$

$S \longrightarrow (L,(a,a))$

$S \longrightarrow (S,(a,a))$

$S \longrightarrow (a,(a,a))$

$S \longrightarrow (L)$

$S \longrightarrow (L,S)$

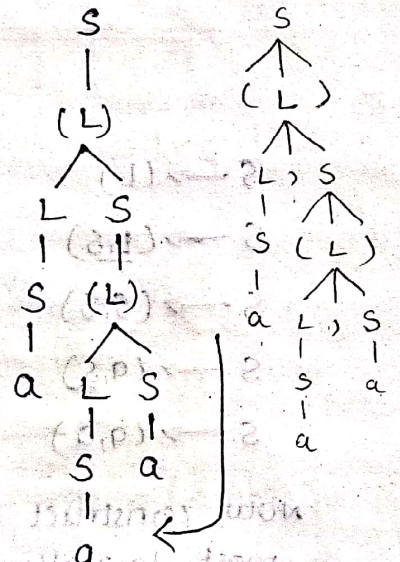
$S \longrightarrow (S,(L))$

$S \longrightarrow (a,(L,S))$

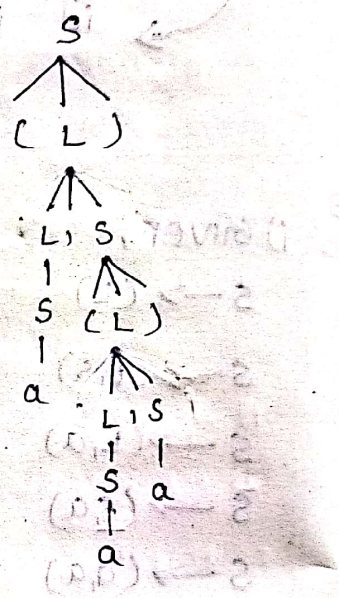
$S \longrightarrow (a,(S,S))$

$S \longrightarrow (a,(a,a))$

Now, construct the right most derivation tree.



Now construct left most derivation tree.



iii) Given,

$S \longrightarrow (L)$

$S \longrightarrow (L,S)$

$S \longrightarrow (L,(L))$

$S \longrightarrow (L,(L,S))$

$S \longrightarrow (L,(S,S))$

$S \longrightarrow (L,((L),(L)))$

$S \longrightarrow (L,((L,S),(L,S)))$

$S \longrightarrow (L,((L,S),(S,a)))$

$S \longrightarrow (L,((S,a),(a,a)))$

$S \longrightarrow (L)$

$S \longrightarrow (L,S)$

$S \longrightarrow (a,(L))$

$S \longrightarrow (a,(L,S))$

$S \longrightarrow (a,(S,S))$

$S \longrightarrow (a,((L),S))$

$S \longrightarrow (a,((L,S),S))$

$S \longrightarrow (a,((a,a),S))$

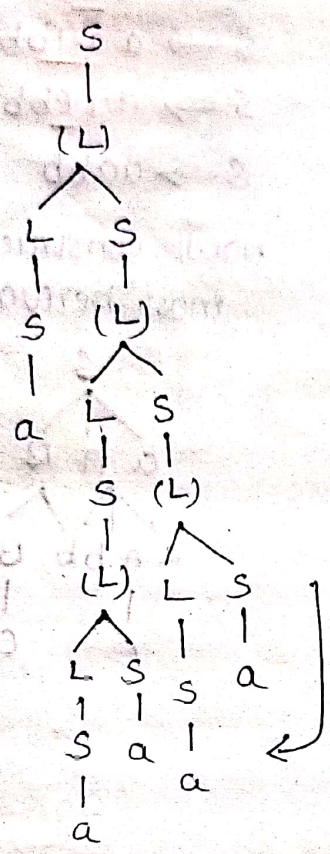
$S \longrightarrow (a,((a,a),(L)))$

$S \longrightarrow (a,((a,a),(L,S)))$

$S \longrightarrow (S,((a,a),(a,a)))$

$S \longrightarrow (a,((a,a),(a,a)))$

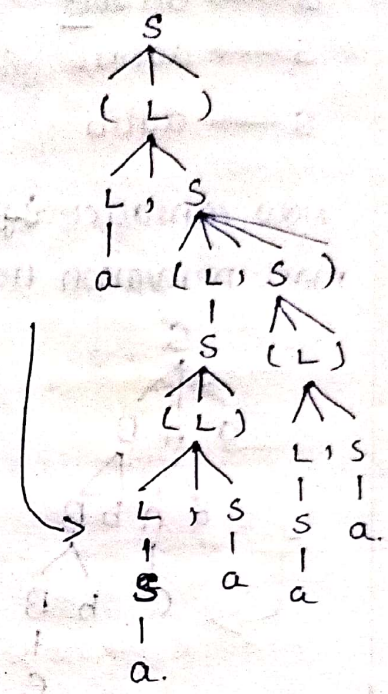
Now, construct the right most derivation tree.



$S \longrightarrow (a,((a,a),(S,a)))$

$S \longrightarrow (a,((a,a),(a,a)))$

left most derivation tree.



---

4. Soln:

Given, $S \longrightarrow as \mid asbS \mid \epsilon$ string aaabaab

$S \longrightarrow \underline{as}$

$S \longrightarrow aa\underline{sbs}$

$S \longrightarrow aaaba\underline{sbs}$

$S \longrightarrow aaabaab\epsilon$

$S \longrightarrow aaabaab$

$S \to \underline{as}bs$

$S \to aa\underline{sbs}$

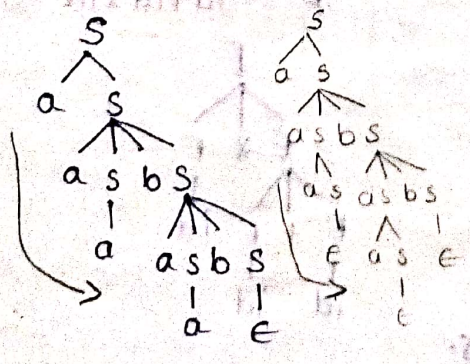
$S \to aaa\epsilon b\underline{asbs}$

$S \to aaab\epsilon\underline{asb}\epsilon$

$S \to aaabaa\epsilon b\epsilon$

$S \to aaabaab$

Now, construct left most derivation tree.



$S \longrightarrow asbs$

$S \longrightarrow a\underline{sb}\epsilon$

$S \longrightarrow aa\underline{sb}$

$S \longrightarrow aaasb\underline{sb}$

$S \longrightarrow aaasba\underline{sb}$

$S \longrightarrow aaaSba\underline{Sb}$

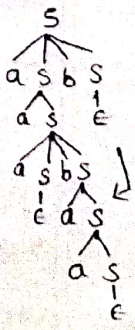
$S \longrightarrow aaa\underline{Sbaa}\underline{Sb}$

$S \longrightarrow aaa\underline{S}baa\epsilon b$

$S \longrightarrow aaa\epsilon baa\epsilon b$

$S \longrightarrow aaabaab$

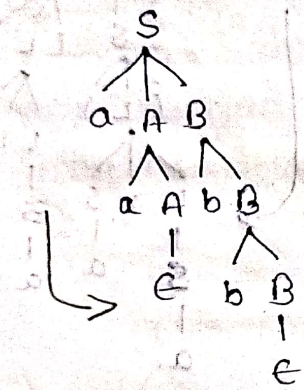
Now, construct right most derivation tree.



5) Sol:- Given, $S \rightarrow aAB \mid bBA \mid \epsilon$, $A \rightarrow aAb \mid \epsilon$ $B \rightarrow bB \mid \epsilon$ aabb

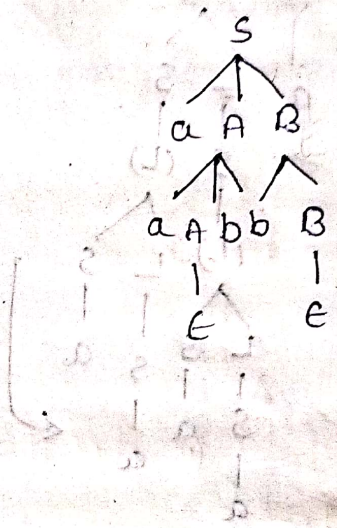
$S \rightarrow \underline{a}\underline{A}B$
$S \rightarrow aa\underline{A}\underline{B}$
$S \rightarrow aa\epsilon\underline{b}B$
$S \rightarrow aabb\underline{B}$
$S \rightarrow aabb\epsilon$
$S \rightarrow aabb$

Now, construct left most derivation tree



$S \rightarrow \underline{a}\underline{A}B$
$S \rightarrow aA\underline{b}\underline{B}$
$S \rightarrow a\underline{A}b\epsilon$
$S \rightarrow aa\underline{A}bb$
$S \rightarrow aa\underline{\epsilon}bb$
$S \rightarrow aabb$

Now, construct right most derivation tree

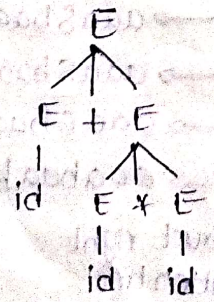


* Ambiguous grammar :-

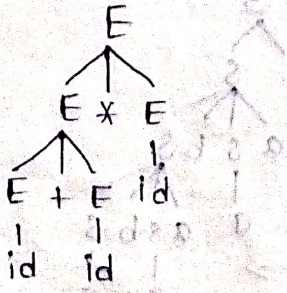
A grammar is said to be ambiguous if it generates more than one parse tree for the language.

Ex:- $E \rightarrow E+E \mid E*E \mid id$ generate the id+id*id then find Ambiguous or not?

$E \rightarrow E+E$
$E \rightarrow E+E*E$
$E \rightarrow E+id*id$
$E \rightarrow id+id*id$



$E \rightarrow E+E$
$E \rightarrow E+E*E$
$E \rightarrow E+E*id$
$E \rightarrow E+id*id$
$E \rightarrow id+id*id$



∴ It is ambiguous.

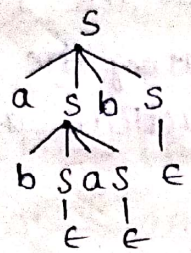
2) $S \to asbS$
$S \to bsaS$
$S \to \epsilon$

generate the string abab and prove that ambiguous or not

Sol:

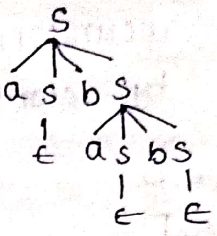
$S \to \underline{a}sbS$
$S \to ab\underline{s}asbS$
$S \to abe\underline{a}sbS$
$S \to ababe$
$S \to abab$

LMDS



$S \to asb\underline{S}$
$S \to a\underline{s}basbS$
$S \to asbaebe$
$S \to abab$
$S \to abab$

RMDT



3) $S \to AB$
$B \to ab$
$A \to aa$
$A \to a$
$B \to b$

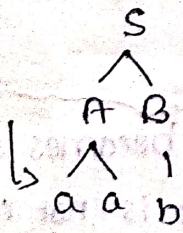
generate the string aab then prove that ambiguous or not:

$S \to \underline{AB}$
$S \to aa\underline{B}$
$S \to aab$

$S \to \underline{AB}$
$S \to \underline{A}ab$
$S \to aab$

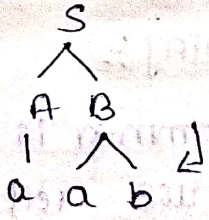
Left most derivation

Left most derivation tree



Right most derivation

Right most derivation tree



∴ It is a ambiguous.