ROHINI COLLEGE OF ENGINEERING & TECHNOLOGY
Near Anjugramam Junction, Kanyakumari Main Road, Palkulam, Variyoor P.O - 629401
Kanyakumari Dist, Tamilnadu., E-mail : admin@rcet.org.in, Website : www.rcet.org.in

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

NAME OF THE SUBJECT : COMPILER DESIGN

Subject code : CS8602

Regulation : 2017

UNIT I- INTRODUCTION TO COMPILERS

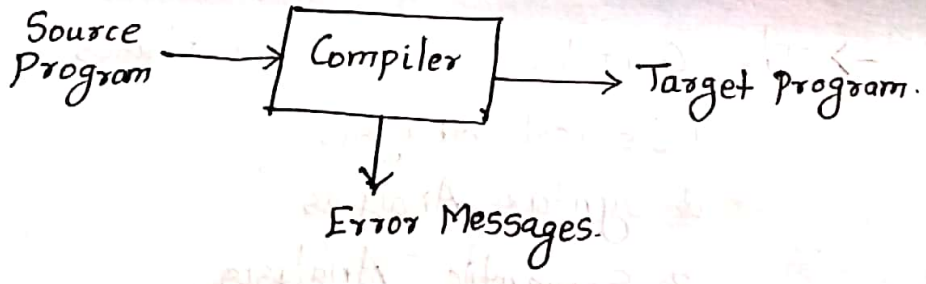
CS8602 - Compiler Design

Unit - I

Structure of a compiler – Lexical Analysis – Role of Lexical Analyzer – Input Buffering – Specification of Tokens – Recognition of Tokens – LEX – finite Automata – Regular Expressions to Automata – Minimizing DFA

1. Introduction to Compiler.

* Compiler is a program which translate a program written in source language to an equivalent program in the target languages.



* An important role of the compiler is to report any errors in the source program, that it detects during the translation process.

ie) The main task of the compiler is to translate an error free program.

* The compilers are classified as

1. Single pass Compiler
2. Multipass Compiler
3. Load and go Compiler
4. Optimizing Compilers
5. Debugging Compilers.

The structure of a Compiler (Phases of a Compiler)

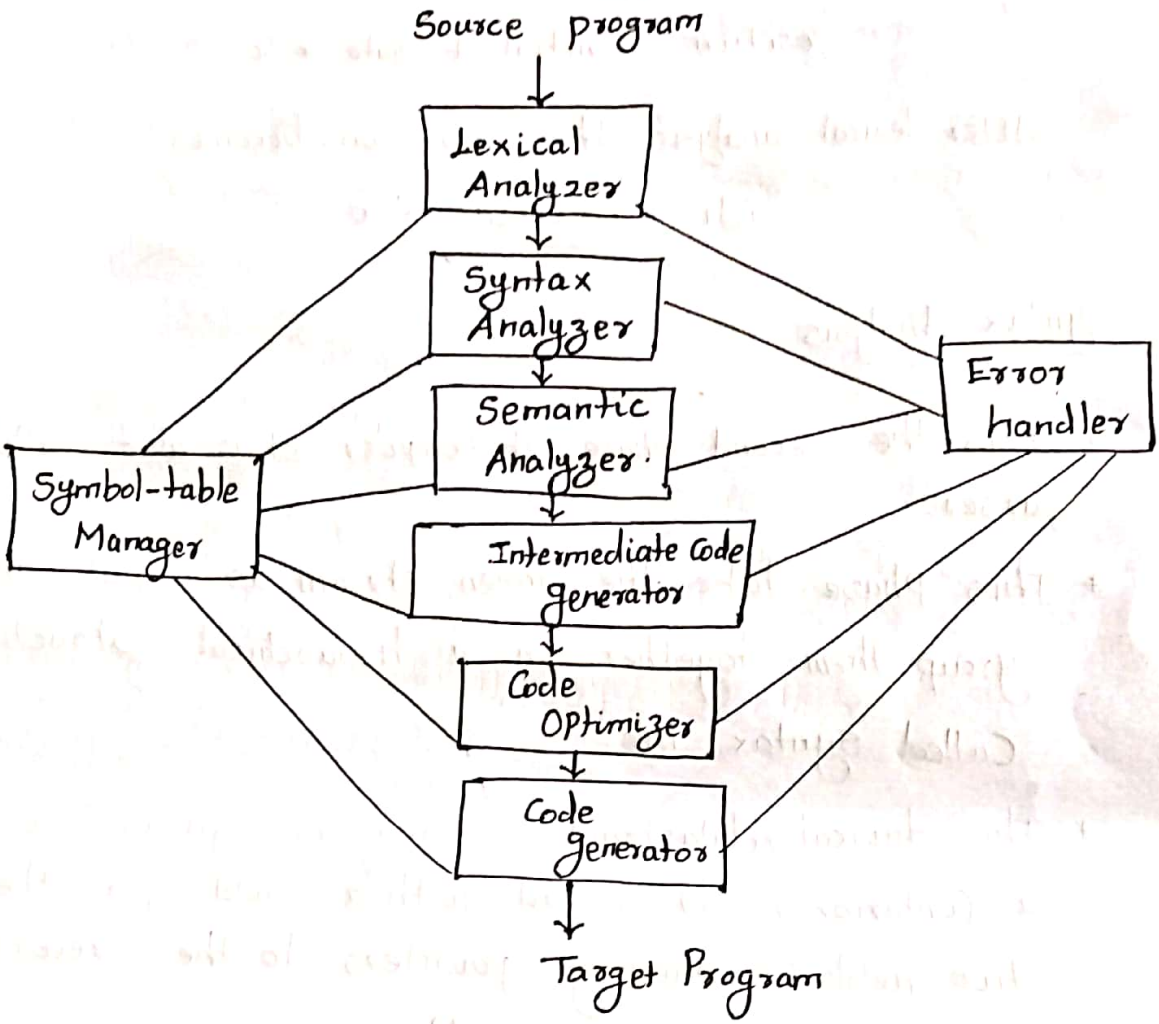
* A Compiler operates in phases, each of which transforms the source program from one representation to another.

→ The Compiler includes six phases

1. Lexical Analysis
2. Syntax Analysis
3. Semantic Analysis
4. Intermediate Code generation
5. Code Optimization
6. Code generation.

* The two other activities are Symbol table management and error handling phases, these two parts interacting with all the other Six phases of Compiler.

Fig: phases of a Compiler



1. Lexical Analysis

* It is also called as linear analysis or scanner. It is the first phase of the Compiler.
* This phase reads the characters in the source program from left to right and group them into stream of tokens.
* The group of characters forming a token is called the lexeme for the token.

For example

consider the expression

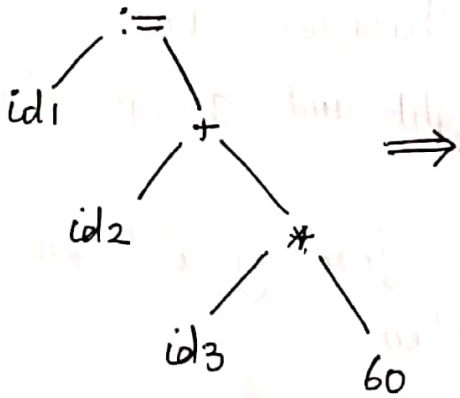
position := initial + rate * 60

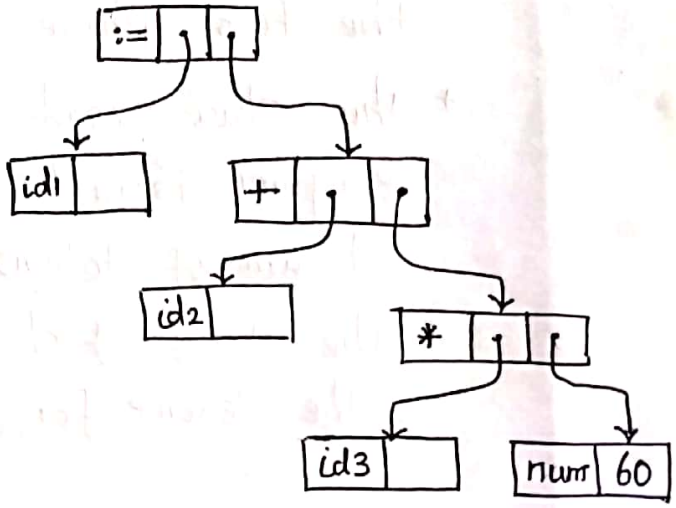
After lexical analysis, the expression becomes

id1 := id2 + id3 * 60

Syntax Analysis

* It is the second phase of Compiler. It is also called as parser.
* This phase take the token stream as input and group them together in a hierarchical structure called Syntax tree.
* The typical datastructure for the syntax tree contains a (interior node) record with a field for the operator & two fields containing pointers to the records for the left and right children.
* A leaf is a record with 2 or more fields, one to identify the token and other to record information about the token.

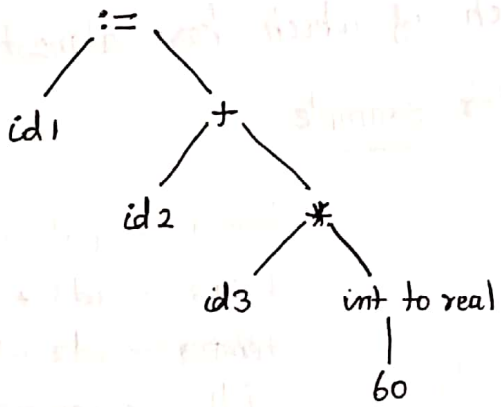




Semantic Analysis

* It is the third phase of Compiler.
* This phase takes the input as a syntax tree and identifies its meaning and check whether the given Syntax is correct or not.
* It generally performs type checking of the identifiers and constants and creates the Semantic tree

Example - Semantic tree



Intermediate Code Generation

* It is the 4th phase of Compiler. It gets the input from the semantic analysis and converts the input into intermediate code such as three address code (TAC)
* The intermediate representation should have 2 properties
  - → It should be easy to produce
  - → It should be easy to translate into the target program.

→ The intermediate code can be represented in 3 ways.

1. Three - address Code
2. Syntax Tree notation
3. Postfix notation.

* Most of the compiler generates three-address code

The three address code looks like the assembly language in which every memory location can act like a register. It has sequence of instructions each of which has atmost three operands.

For example

```
temp1 := int to real(60)
temp2 := id3 * temp1
temp3 := id2 + temp2
id1   := temp3
```

This is the three-address code generated for syntax tree

Code optimization

* This phase get the intermediate code as input and produce optimized intermediate code as output.
* This phase reduces the redundant code and attempt to improve the intermediate code to form the faster running machine code.
* During optimization the result of the program is not affected. To improve code generation, the optimization involves.

* Deduction and removal of dead Code
* Redundant Code elimination
* removal of unwanted temporary variable
* Calculation of constants in expression and terms.
* loop unrolling
* Moving Code outside the loop

for example

Optimized Code for the intermediate Code is

```
temp1 := id3 * 60
id1 := id2 + temp1
```

Code Generation

* The final phase of the compiler is the generation of target Code.
* The target Code may be either "relocatable machine Code" or "assembly Code".
* Then the intermediate instructions are translated into sequence of machine instructions that perform the same task.
* The Code generation phase involves
  * allocation of registers and memory
  * generation of correct references

* generation of Correct datatype
* generation of machine Code.

Example - target Code generated from the intermediate instructions.

```
MOVF   id3, R2
MULF   #60.0, R2
MOVF   id2, R1
ADDF   R2, R1
MOVF   R1, id1
```

Symbol table Management

* The main function of the compiler is to record the identifiers used in the Source program and Collect the information about the identifier.
* The attributes (information) about the identifier.
* The symbol table is a datastructure containing a record for each each identifier with its attributes
* In Lexical analysis phase the identifier is recorded in the symbol table.

Error Detection and Reporting (Error handler)

* Each phase can detect errors. After detecting an error, each error is handled by error handling mechanism & the phase deals with that errors.

The example statement that is performed by each phase of the compiler is shown below

Position := initial + rate * 60

↓

Lexical analyzer

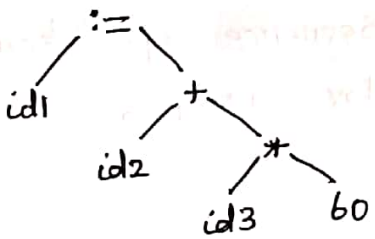
↓

id1 := id2 + id3 * 60 (tokens)

↓

Syntax analyzer

↓



(Syntax tree)

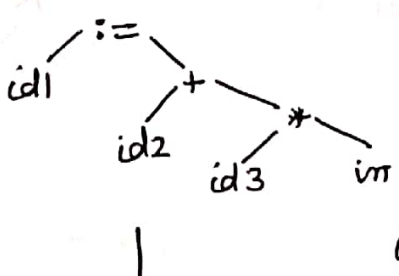
Symbol Table

1	Position	- - -
2	initial	- -
3	rate	- -

↓

Semantic analyzer

↓



(Semantic tree)

↓

Intermediate Code generator

↓

temp1 := int to real (60)
temp2 := id3 * temp1
temp3 := id2 + temp2
id1 := temp3

(Intermediate Code)

↓

Code Optimizer

↓

temp1 := id3 * 60.0
id1 := id2 + temp1

(Optimized Code)

↓

Code Generator

↓

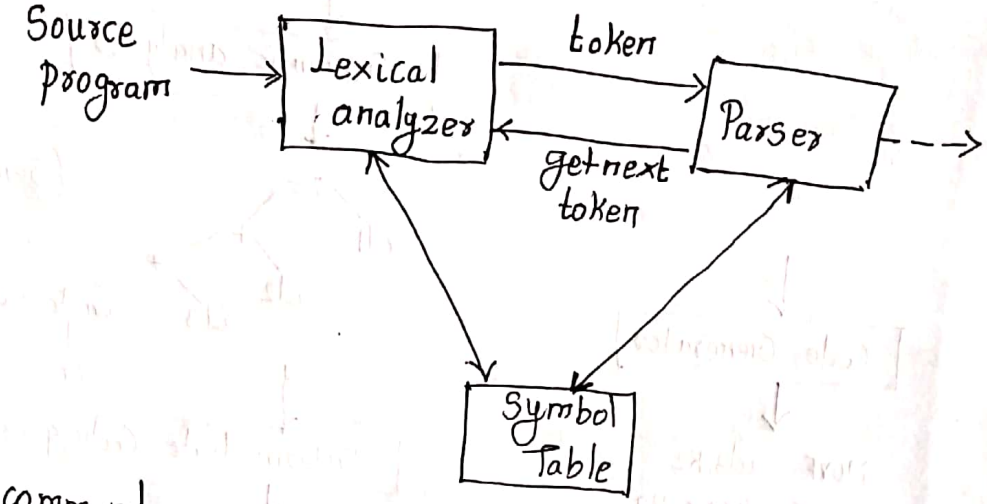
```
MOVF   id3, R2
MULF   #60.0, R2
MOVF   id2, R1
ADDF   R2, R1
```

Syntax Analysis

Eg

The Role of the lexical analyzer

* The lexical analyzer is the first phase of a compiler. It is also called as Scanner
* The main task of the lexical analyzer is to read the input characters and produce the output as a sequence of tokens that the parser uses for Syntax analysis.
*



* The command get next token is used to read the input characters until it can identify the next token.

ii) Enhancing Compiler I

Tokens, Patterns and Lexemes

i) Tokens

* A token is an atomic unit represents a logically cohesive sequence of characters.
* The process of forming tokens from an input stream of characters is called tokenization.
* Consider the expression Sum=3+2

Lexeme	Token Type
Sum	identifier
=	Assignment operator
3	Number
+	Addition operator
2	Number
#	End of Statement

ii) Patterns

* The pattern is a rule describing the set of lexemes that can represent a particular token in the source program.

For example

Pattern for the token float is the sequence of characters starts with 'f' followed by 'l' 'o' 'a' 't'

* The Lexical analyzer also perform secondary task at the user interface
  ie) The lexical analyzer read the source program and removes Comments, white spaces in the form of blank.
* The lexical analyzers are divided into two phases
  i) Scanning – The scanner is responsible for doing simple task
  ii) Lexical analysis – It is responsible for doing complex task.

Issues in Lexical Analysis

* There are several reasons for seperating the analysis phase into Lexical analysis and Syntax analysis.

i) Simpler Design

* To make the design simpler. The seperation of lexical & Syntax analysis allows the other phases to be simpler.

ii) Improving Compiler efficiency

A seperate lexical analyzer allows us to construct a specialized and efficient processor for the task.

iii) Enhancing compiler portability

Lexeme

* Collection or group of characters forming tokens is called lexeme.

ie) It is the sequence of character in the source program which is matched by the pattern for a token.

Example

Token	Sample Lexemes	Informal description of patterns.
const	const	const
if	if	if
relation	<, <=, =, <>, >, >=	< or <= or = or <> or >= or >
id	Pi, count, D2	letter followed by letters and digits.
num	3.1416, 0, 6.02E23	any numeric constant
literal	"Core dumped"	any character between and except.

Attributes for Tokens

* Some tokens have attributes that can be passed back to the parser
* When more than one pattern matches a lexeme, the lexical analyzer must provide additional information about the particular lexeme that matched to the subsequent phases of the compiler.
* The lexical analyzer collects the information about tokens into thier associated attributes.
  A token has single attribute, which holds the pointer to the symbol table entry, in which the information about the token is kept.

For example

⇒ The token and attribute values for the given statement is

$x = y \times 10$

<id, pointer to symbol table entry for x>
<assign_op>
<id, pointer to symbol table entry for y>
<mult_op>
<num, integer value 10>

Lexical Errors

* Few error are recognized at lexical level, because lexical analyzer has a very localized view of the source program.
* The error recovery strategy for solving the problems by the lexical analyzer is "Panic mode Recovery"
* In panic mode recovery, we delete sucessive characters from the remaining input, until the lexical analyzer can find a well-formed token.

⇒ Other possible error recovery methods are

* Deleting an extraneous character
* Inserting a missing character.
* Replacing an incorrect character by correct character
* Transposing two adjacent characters

* These are about the role of a lexical analyzer.

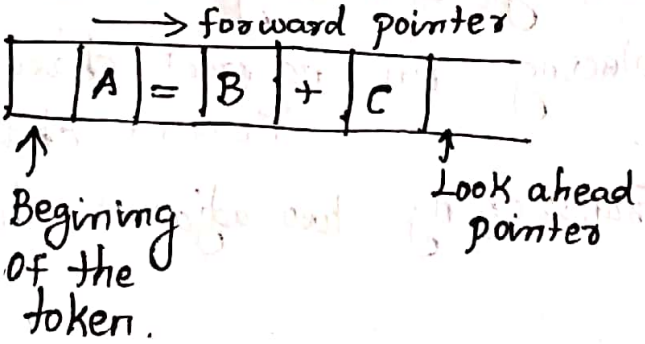
Input Buffering

* This method is used to read the source program and to identify the tokens efficiently.

* The speed of lexical analyzer is a main concern in Compiler design. Since the lexical analyzer is the only phase of the compiler, which takes more time in reading the program character by character

* The speed of lexical analysis has to be improved using proper buffering technique.

* As characters are read from left to right, each charact is stored in the buffer ~~from a~~ to form a meaningful token.



* We introduce a two-buffer scheme that handles large look aheads safely.

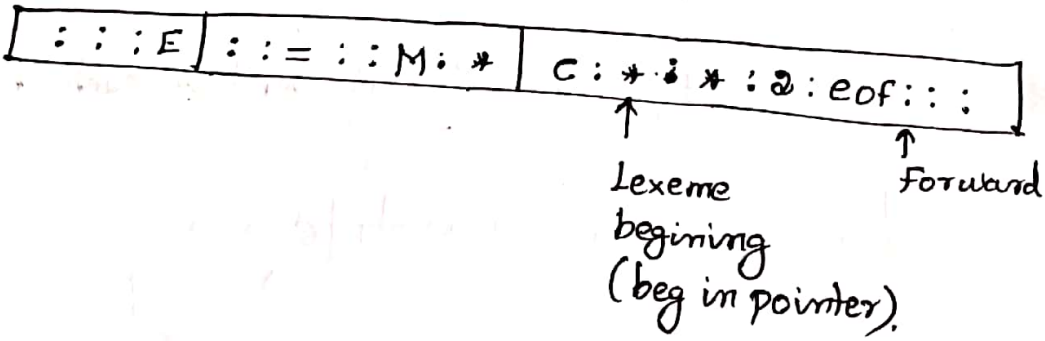
ie) Buffer pair method
Sentinel method.

Buffer pair method

* The lexical analyzer needs to look ahead many characters beyond the lexeme for finding the pattern.
* In order to reduce the amount of overhead required to process an input character, specialized buffering technique have been developed.
* A buffer is divided into two N-character halves, where N is the number of characters on one disk block.

eg: 1024 or 4096.

Fig: An input buffer in two halves.



The processing of buffer pair is as follows

i) Read N input character into each half of the buffer.
ii) If fewer than N characters in the input, then read eof (end of file) marker.
iii) To pointers to the input buffers are maintained
   i) Begin pointer – points the start of the lexeme
   ii) forward pointer – it set to the character as its right end.

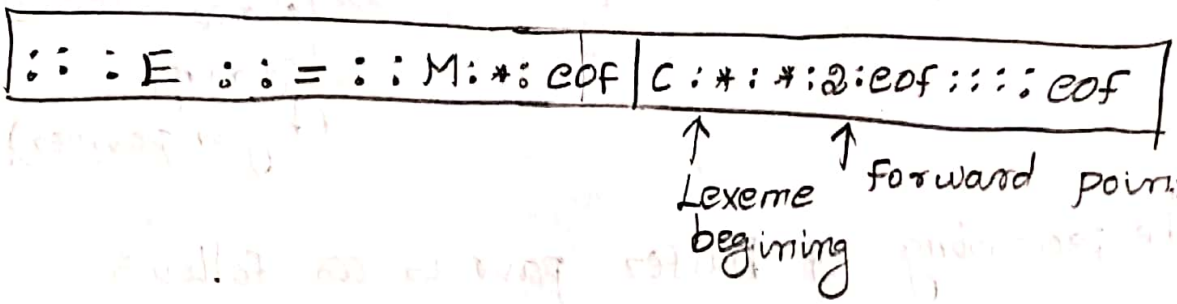
Sentinel method

→ In the buffer method we should make a check each time we move the forward pointer that we have not moved off one half of the buffer.

→ Instead of testing the forward pointer each time by two tests, we extend each buffer half to hold a sentinel character at the end and reduce the number of tests to one.

* A Sentinel is a special character which is not a part of the source program, used to represent the end of file (eof)

Fig: Sentinels at end of each buffer half



Specification of Tokens

* Regular expressions are an important notations, which is used to specify ~~take~~ patterns.
* Each pattern matches a set of strings, so regular expressions serve as names for set of strings.

→ There are 3 specification of tokens

1) Strings 2) Languages 3) Regular expressions

1. String and Languages

i) Alphabet – An alphabet or character class denotes any finite set of symbols.

eg: Letters, characters, ASCII characters, EBCDIC characte

* Symbols → Collection of letters and characters

ii) String → A String over some alphabet is a finite sequence of symbols.

eg: 101101 is a string over $\{0,1\}^*$

* The length of the string is denoted as $|s|$, that is the number of occurance of the symbol in s.

$$|101| = 3$$

* Empty string is refered as ϵ that has the length zero.

iii) Language → A language denotes any set of strings over some fixed alphabet Σ

Language $L = \{0^n 1^n / n > 0\}$

Some common terms associated with part of a string are as follows

Let s be the string where s = "regular"

a) prefix of s : a string obtained by removing 0 or more trailing symbol of string s.
eg → 'reg'

b) Suffix of s : A string formed by deleting 0 (or) more of the leading symbols of String s.
eg: = "lar"

c) Substring of s : A string obtained by deleting a Prefix and suffix from s.
eg: "gul"

d) Proper prefix, Sufix, Substring of s
Any non-empty string s, that is respectively a prefix, suffix, substring of 's' such that $s \neq x$

e) subsequence of s
Any string formed by deleting 0 or more, not necessarily contiguous symbol from s
eg:- rgl

Operations on languages

→ The operations of languages are union, closure, and concatenation etc

i) Union of L and M (LUM)

ii) $LUM = \{ s / s \text{ in } L \text{ or } s \text{ is in } M \}$

ii) Concatenation of L and M ⇒ LM

$LM = \{ st / s \text{ is in } L \text{ \& } t \text{ is in } M \}$

iii) Kleene closure of L ⇒ L^*

ie) $L^* = \bigcup_{i=0}^{\infty} L^i$ (L^* denotes zero or more concatinations of L)

iv) Positive closure of L ⇒ L^+

$L^+ = \bigcup_{i=1}^{\infty} L^i$ (L^+ denotes one or more occurance of L)

Example

consider $L = \{0, 1\}$ and $M = \{a, b, c\}$

i) $LUM = \{0, 1, a, b, c\}$

ii) $LM = \{0a, 0b, 0c, 1a, 1b, 1c\}$

iii) $L^* = \{\epsilon, 0, 1, 01, 10 \ldots\}$

iv) $L^+ = \{0, 1, 00, 01, 10, 11 \ldots\}$

Regular Expressions

⇒ It is a set of rules that defines the identifier as

letter (letter/digit)*

Regular expression is used to describe the tokens of a programming language.

* Each regular expression denotes the language $L(r)$

* The rules of regular expression 'r'

1. 'ε' is a regular expression that denotes {ε}
   i.e) Set containing empty
2. If 'a' is a symbol in Σ, then 'a' is the RE that denotes {a}
3. If r & s are regular expressions denoting the language $L(r)$ & $L(s)$
   Then

$(r)\|(s)$	denotes	$L(r) \cup L(s)$
$(r)(s)$	denotes	$L(r) \cdot L(s)$
$(r)^*$	denotes	$(L(r))^*$
(r)	denotes	$L(r)$

→ The precedence and associativity of operators are as follows

i) Unary operator '*' has highest precedence & left associative

ii) Concatenation '.' has second highest precedence and left associative

iii) '|' has lowest precedence & left associative.

Algebric properties of regular expressions

i) $r|s = s|r$ ⇒ | is Commutative

ii) $r|(s|t) = (r|s)|t$ ⇒ Concatenation is associative

iii) $(rs)t = r(st)$ ⇒ Concatenation is associative

iv) $r(s|t) = rs|rt$
$(s|t)r = sr|tr$
} - Concatination distributes over |

v) $\epsilon r = r$ &
$r\epsilon = r$ ϵ is the identity element for Concatenation

vi) $r^* = (r|\epsilon)^*$ → Relation between * and ϵ

vii) $r^{**} = r^*$ → * is idempotent

Regular Definition

* Regular Definition is the sequence of definitions over Σ and its form is

$d_1 \rightarrow r_1$ where d_i is distinct name
$d_2 \rightarrow r_2$ r_i is regular expression
$\vdots$
$d_n \rightarrow r_n$

Eg

Regular Definition of identifier is

letter → A|B|...|z|a|b|...|z|
digit → 0|1|2...|9|
id → letter (letter|digit)*

Notational Shorthands

1. One or more instances (+)
2. Zero or more instance (*)
3. Zero or one instance (?)
4. Characters classes

d) Non-Regular set

→ Some languages cannot be described by any regular expression.

→ This non-regular set can be specified by context free Grammar (CFG)

→ These are about specification of Token.

Recognition of Tokens

* Tokens are recognized by following grammatic specifica of tokens.

Eg Consider the following grammar

stmt → if expr then stmt /
if expr then stmt else stmt / ε

expr → term relop term / term

term → id / num

Here if, then, else, relop, id and num are terminals. These terminals are generate the following regular definition

ie)

if → if

then → then

else → else

relop → < / <= / <> / > / >=

id → letter (letter/digit)*

num → digit$^+$ (.digit$^+$)? (E(+/-)?digit$^+$)?

* In the above statement, the lexical analyzer will recognise keywords as if, then, else as well as lexemes denoted by relop, id and num.

* The goal of the lexical analyzer is to isolate the lexeme for the next token in the input buffer and produce as output.

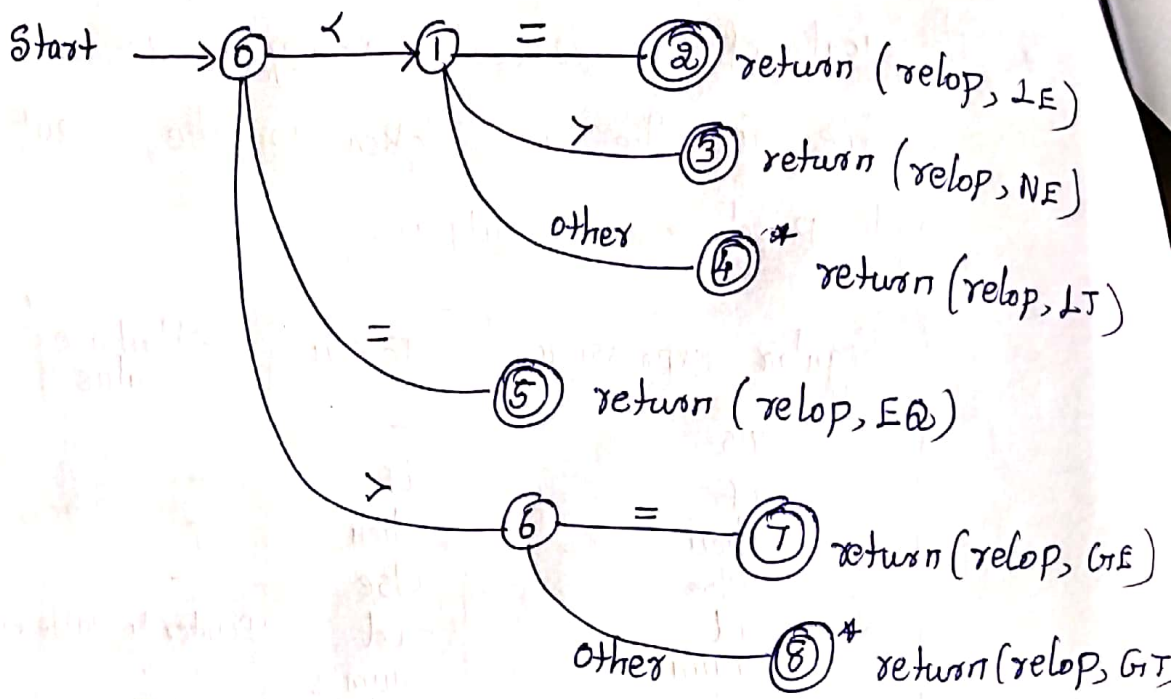
Regular expression	Token	Attribute value
ws	–	–
if	if	–
then	then	–
else	else	–
id	id	Pointer to table entry
num	num	"
<	relop	LT
<=	relop	LE
<>	relop	NE
>	relop	GT
>=	relop	GE
	relop.	EQ

Transition Diagram

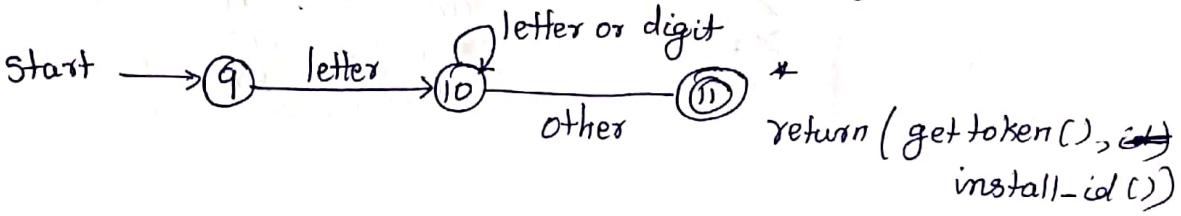
* It is a pictorial representation denotes the actions that take place when a lexical analyzer is called by the parser to get the next token.
* It is used to keep track of information about characters that are seen as the forward pointer scans the input.
* Each state is represented as nodes and transitions are represented by edges.

* The transition diagram for the relational operator.

(a)



Transition diagram for identifier and keyword



→ The symbol table is examined, if the lexeme found install_id() return 0 & it returns a pointer to the symbol table entry.

→ If lexeme is not found, it is installed as a variable & pointer to the newly created entry is returned.

→ gettoken() looks for lexeme in the symbol table & the token is returned.

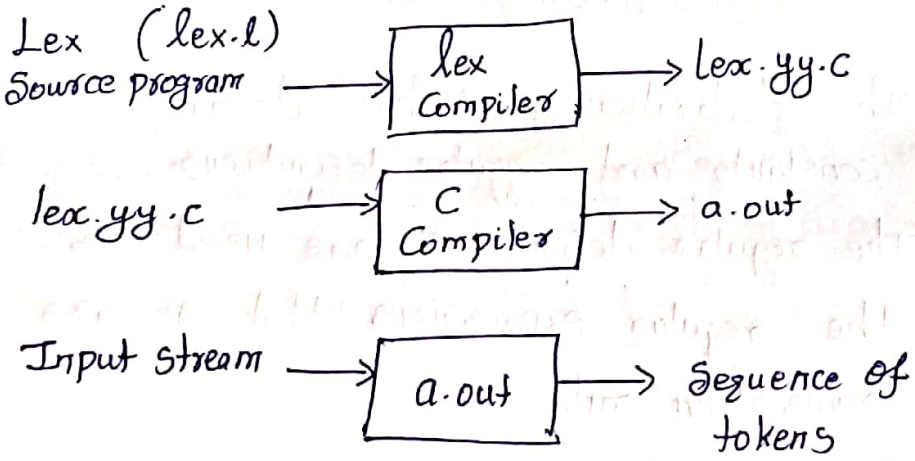
Language for Specifying Lexical Analyzer

* There are several tools for constructing lexical analyzer

  ie) LEX – A Lexical analyzer generator
  YACC – Yet another compiler compiler
  FLEX – A fast Scanner Generator
  BISON – YACC – Compatible parser generator.

* LEX is used to specify the lexical analyzer for a variety of languages.

* Input Specification



* The specification of lexical analyzer is prepared by creating a program lex.l in the lex language.
* The lex.l is run through Lex compiler to create C program lex.yy.c
* The lex.yy.c consists of a tabular representation of a transition diagram constructed from the regular expression of lex.l

* Then the program runs through a compiler and produce the object program a.out, which is the lexical analyzer that transforms an input stream into sequence of tokens.

Lex Specification

→ The lex program consists of three parts

declarations
%%
Translation Rules
%%
Auxiliary procedures.

* The declaration part has declarations of variables, constants and regular definitions.
* The regular definitions are used as components of the regular expressions that appears in the translation rule
* The translation rules are

P1 {action 1}
P2 {action 2}
:
Pn {action n}

where P_i - regular expression

* The third part hold the auxiliary procedures needed by the actions.

Finite Automata

* A better way to convert a regular expression into a recognizer is to construct a generalized transition diagram from the expression. This diagram is called a finite automata.

* It can be
  - → Deterministic finite automata (DFA)
  - → Non Deterministic Finite automata (NFA)

Non Deterministic Finite Automata (NFA)

* NFA is a mathematical model that consist of five tuples denoted by.

$$M = \{ S, \Sigma, \delta, S_0, F \}$$

where

S - Finite set of states

Σ - Finite set of input symbols

δ - a transition function that maps State-symbol pairs to set of states.

S_0 - Starting state

F - Final or Accepting state.

Deterministic Finite Automata (DFA)

→ DFA is a special case of NFA in which

* no states has ϵ - transition
* There is atmost one edge labeled 'a' leaving S.

* DFA has 5 tuples denoted by

$$M = \{ S, \Sigma, \delta, S_0, F \}$$

Where

S - finite set of states

Σ - finite set of input symbols

δ - a transition function that maps state symbol pair to another state

S_0 - Starting state

F - Final (or) accepting state.

Converting NFA to DFA

* The algorithm for converting NFA to DFA is often called subset-construction.

* In the transition table of an NFA, each entry is a set of states, where as in DFA it is a single state.

Subset construction algorithm

* Initially ε-closure (S_0) is the only state in D_{states} and it is unmarked.

* While there is an unmarked state T in D_{states} do begin

mark T

```
For each input symbol a do begin
        U := ε-closure (move(T,a))
        if U is not in Dstates then
         add U as an unmarked state to
         Dstates;
         Dtrans [T,a] := U
    end
end
```

Computation of ε-closure

```
push all states T onto stack;
initialize ε-closure (T) to T;
While stack is not empty do begin
    Pop t, the top element of the stack
    for each state u with an edge from t
        to u labeled ε do
    if u is not in ε-closure (T) do begin
        add u to ε-closure (T)
        push u onto the stack;
    end
end.
```

* DFA has 5 tuples denoted by

$$M = \{ S, \Sigma, \delta, S_0, F \}$$

Where

S - finite set of states

Σ - finite set of input symbols

δ - a transition function that maps state symbol pair to another state

S_0 - Starting state

F - Final (or) accepting state.

Converting NFA to DFA

* The algorithm for converting NFA to DFA is often called subset - construction.
* In the transition table of an NFA, each entry is a set of states, where as in DFA it is a single state.

Subset construction algorithm

* Initially ε-closure (S_0) is the only state in D_{states} and it is unmarked.
* While there is an unmarked state T in D_{states} do begin

mark T

```
For each input symbol a do begin
        U := ε-closure (move(T,a))
        if U is not in Dstates then
            add U as an unmarked state to
            Dstates;
            Dtrans [T,a] := U
    end
end
```

Computation of ε-closure

```
push all states T onto stack;
initialize ε-closure (T) to T;
While stack is not empty do begin
    Pop t, the top element of the stack
    for each state u with an edge from t
        to u labeled ε do
        if u is not in ε-closure (T) do begin
            add u to ε-closure (T)
            push u onto the stack;
        end
    end.
```

* The following steps involved in the construction of DFA from regular expression.

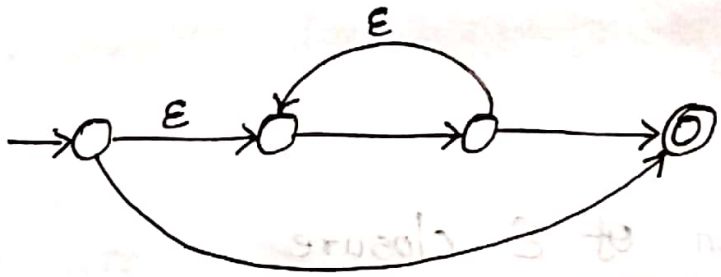
→ Convert regular expression to NFA using Thomson's rules.

→ Convert NFA to DFA

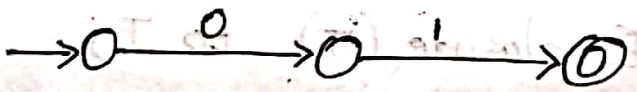
→ Construct minimized DFA

$0^*(01)(0|111)^*$

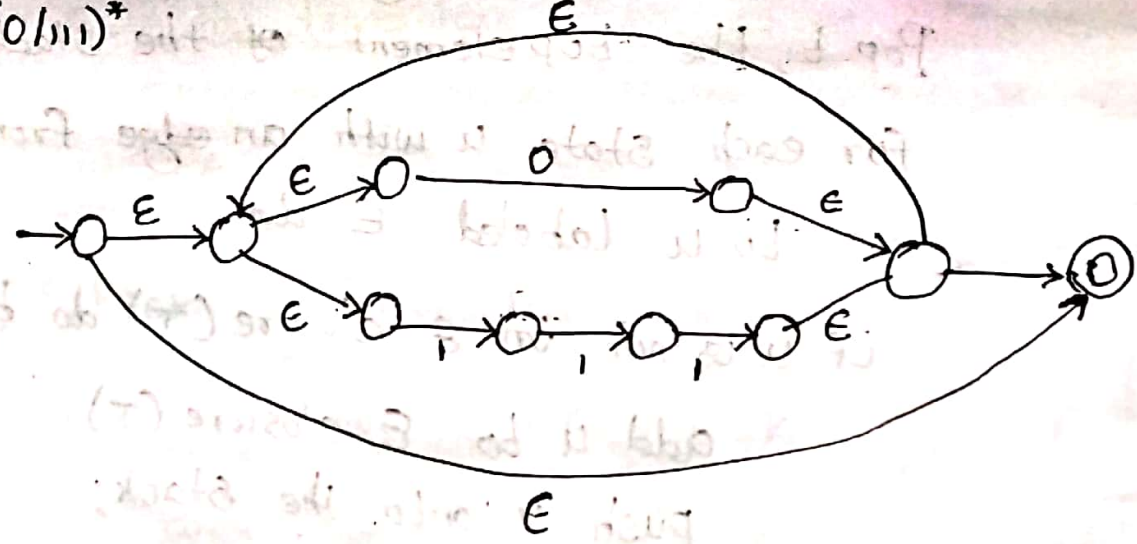
0^*



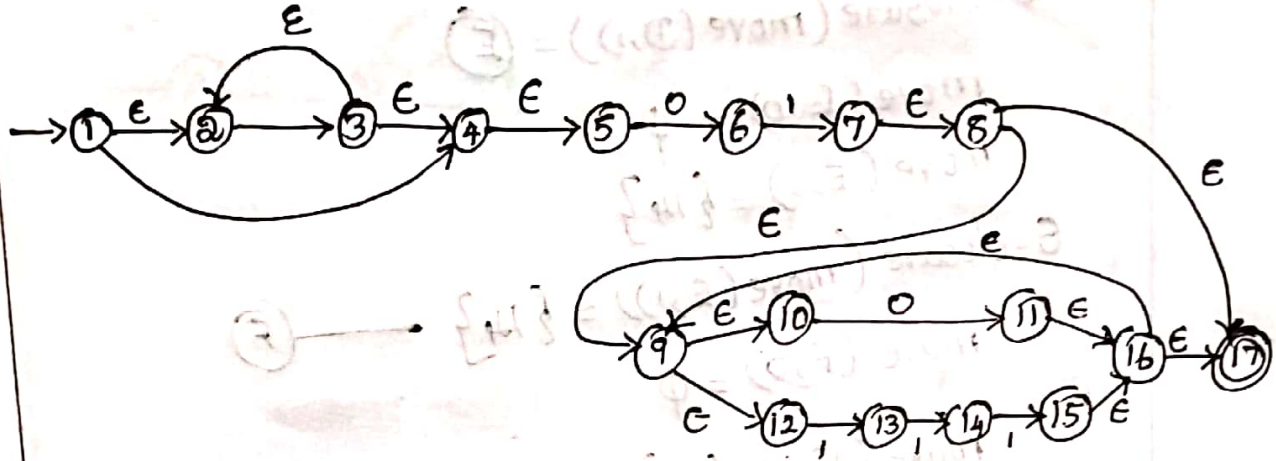
01



$(0|111)^*$



$0^*(01)(0/111)^*$



ϵ-closure (1) = {1, 2, 4, 5} ——— (A)

Move (A, 0) = {3, 6}

ϵ-closure (move (A, 0)) = {3, 4, 5, 2, 6}

= {2, 3, 4, 5, 6} ——— (B)

move (A, 1) = ϕ

move (B, 0) = {3, 6}

ϵ-closure (move (B, 0)) = B

move (B, 1) = {7}

ϵ-closure (move (B, 1)) = {7, 8, 9, 10, 12, 17} ——— (C)

move (C, 0) = {11}

ϵ-closure (move (C, 0)) = {9, 10, 11, 12, 16, 17} ——— (D)

move (C, 1) = {13}

ϵ-closure (move (C, 1)) = {13} ——— (E)

move (D, 0) = {11}

ϵ-closure (move (D,0)) = (D)

move (D,1) = {13}

ϵ-closure (move (D,1)) = (E)

move (E,0) = ϕ

move (E,1) = {14}

ϵ-closure (move (E,1)) = {14} ——— (F)

move (F,0) = ϕ

move (F,1) = {15}

ϵ-closure (move, 1))

= {15, 16, 17, 9, 10, 12} ——— (G)

move (G,0) = {11}

ϵ-closure (move (G,0)) = (D)

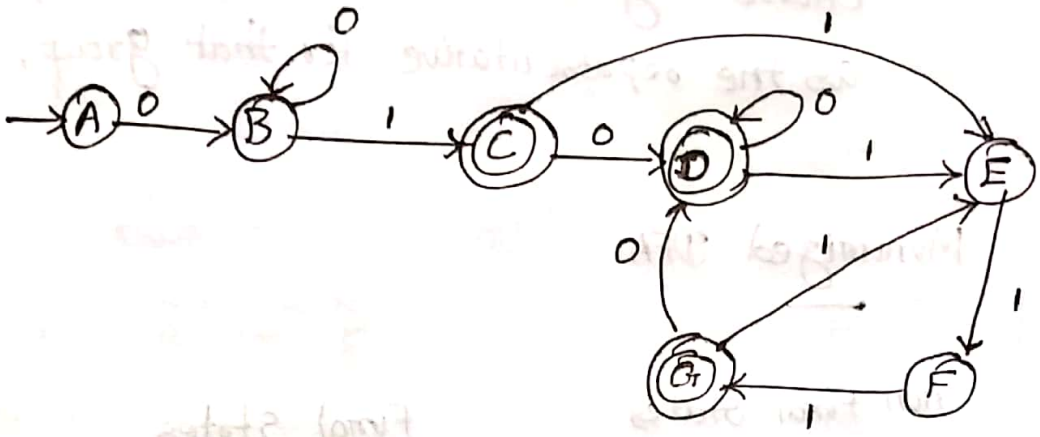
move (G,1) = {13}

ϵ-closure (move (G,1)) = (E)

Transition table

State / ip	0	1
→ A	B	ϕ
B	B	C
* C	D	E
* D	D	E
E	ϕ	F
F	ϕ	G
* G	D	E

DFA



Minimizing the number of states in DFA

* Construct an initial partition Π of 2 groups: the accepting state F and non-accepting states S-F for each group G of Π do begin

* Partition G into subgroups such that 2 states s and t of G are in the same subgroup if any only if for all input symbols 'a', states s and t have transitions on 'a' to states in the same group of Π

* replace G in Π_{new} by the set of all subgroups formed

end

* If $\Pi_{new} = \Pi$, then $\Pi_{final} = \Pi$ and choose any one state in each group as the representative for that group.

Minimized DFA

non final states

ABEF

	0	1
→ A	B	ϕ
B	B	C
E	ϕ	F
F	ϕ	G

Final States

CDG

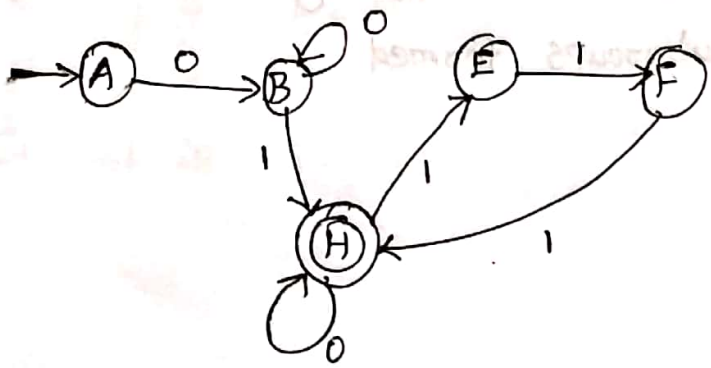
	0	1
C	D	E
D	D	E
G	D	E

ABEF

	0	1
→ A	B	ϕ
B	B	C
E	ϕ	F
F	ϕ	H
H	H	E

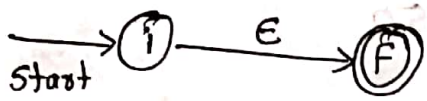
CDG

H

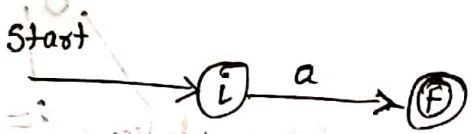


Construction of an NFA from Regular Expression
(Thompson's Construction)

1. For ε

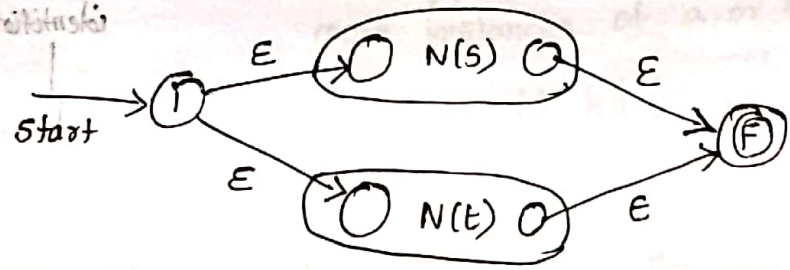


2. For a in ε

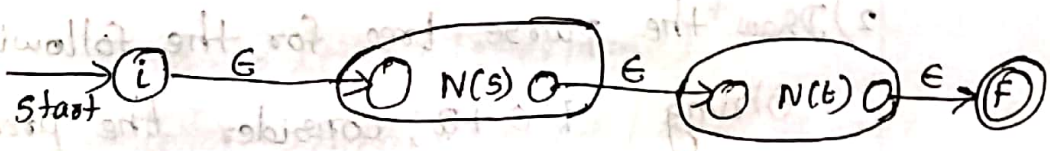


3. Suppose N(s) and N(t) are NFA for regular expression s and t

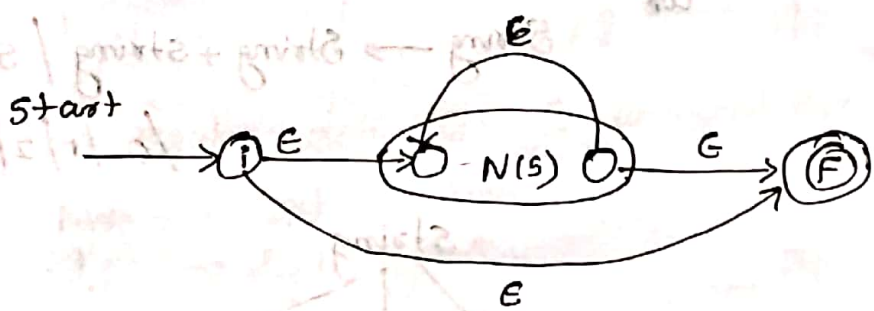
a) s/t



b) st



c) s^*



d) (s)

N(s) itself.

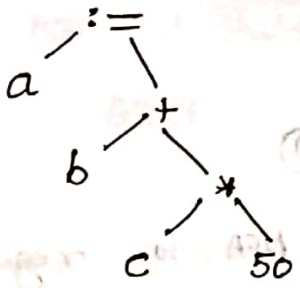
Unit-I

Additional problems ..

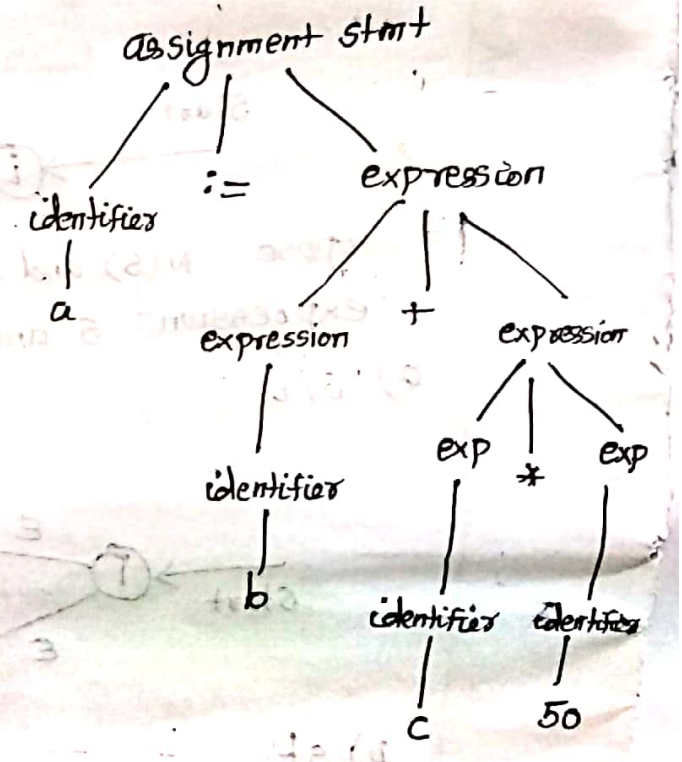
1) Draw the syntax tree and parse tree for

$a := b + c * 50$

Syntax tree



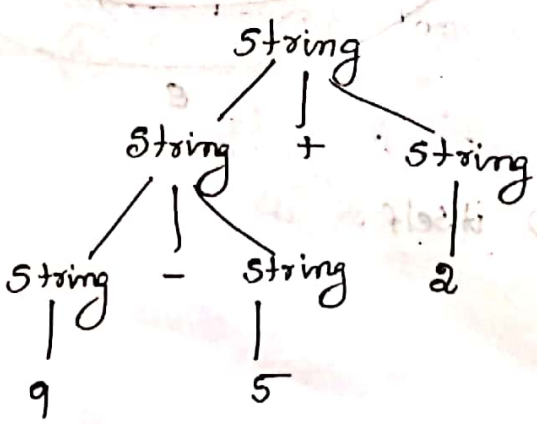
Parse tree



2) Draw the parse tree for the following string $9-5+2$, consider the productions as

String → String + string / string − String

/0 /1 /2 /3 ... /9



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

write down the language generated by the following

i) a/b

ii) a^*

iii) $(a/b)^*$

iv) (a/b) (a/b)

Answer

i) $a/b \Rightarrow L = \{a, b\}$

ii) $a^* \Rightarrow L = \{\varepsilon, a, aa, aaa, \ldots\}$

iii) $(a/b)^* \Rightarrow L = \{\varepsilon, a, b, ab, aa, ba, bb \ldots\}$

= all string contain zero or more instances of a or b

iv) $(a/b)\,(a/b) \Rightarrow L = \{aa, ab, bb, ba\}$

4) Write the regular definition for identifier

id → letter (letter/digit)*

letter → A /B/ . . . /Z/ a/b/ . . /z/

digit → 0/1/2 . . . /9

5) Write the regular definition for unsigned number

num → digit fraction exp

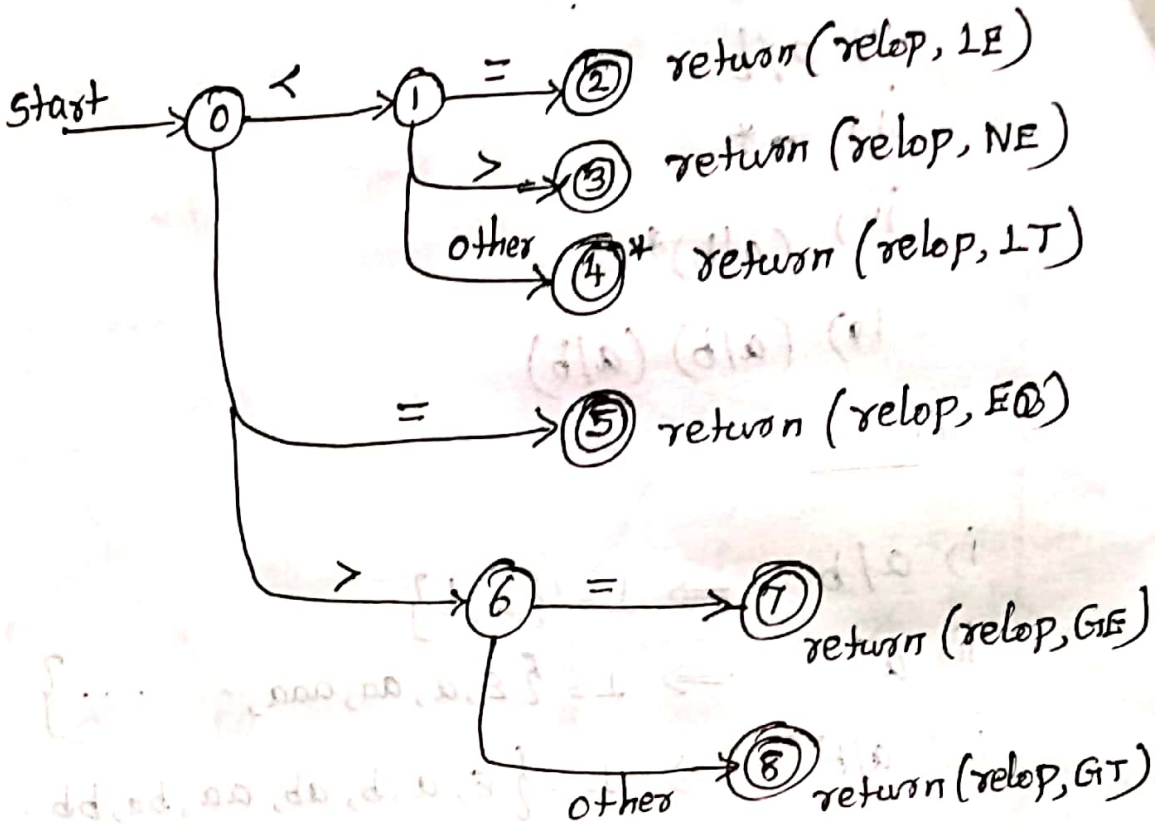
digit → digit digit *

digit → 0/1/2 . . /9

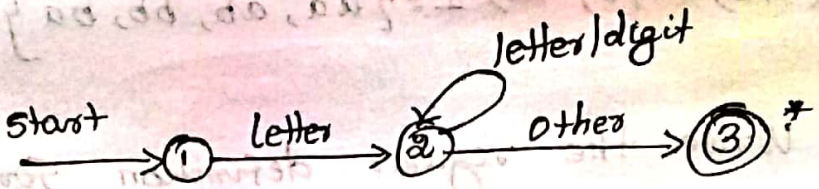
Fraction → . digits /ε

exp → (E (+/- /ε) digits /ε

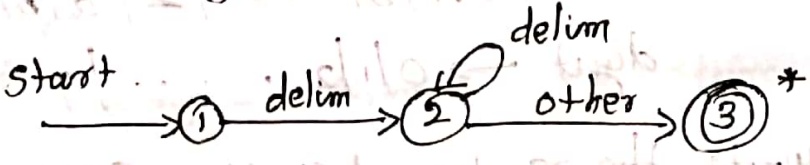
6) Transition diagram for relational operator



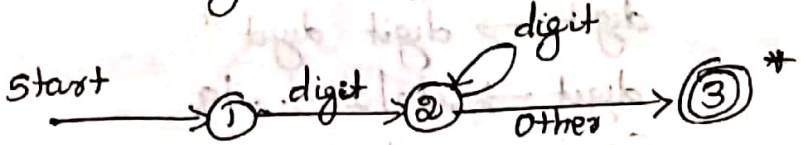
7) Transition diagram for identifier and keywords



8.) Transition diagram for white space



9) Transition diagram for constant



Identify the lexeme that make up tokens.

```
begin
    if  i>j  then   max:=i
    else  max:=j
end.
```

Lexeme	Token	Attribute value.
begin	keyword	
if	keyword	
i	id	pointer to symbol table entry for i
j	id	pointer to symbol table entry for j
>	operator	—
max	id	pointer to symbol table entry for max
:=	operator	—
else	keyword	—
end	keyword	—
;	;	—

11) Give the regular expressions for the following

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

i) all strings containing atleast one a

$(a/b)^* a (a/b)^*$