**Assignment#1**

**Phase 1 - Lexical Analysis**

**Group Members**:

17L-4358 Muhammad Shehriyar

17L-4025 Saif-ul-Haq

17L-4441 Faizan Tanveer

**Lexical Analyzer:**

Description of Language Tokens and Lexemes

**DATA TYPES:**

(int , ^ )

(char , ^)

**KEYWORDS:**

1. **Decision Statements:  if, elif, else**

(if ,      ^)

(elif ,   ^)

(else , ^)

1. **Looping Statements: while**

(while , ^)

1. **Standard input statement: input**

(input , ^)

1. **Standard output statement: print and println**

(print , ^)

(println , ^)

**ARITHMETIC**

(AO , ‘+’)

( AO , ‘-’)

(AO , ‘\*’)

(AO , ‘/’)

**RELATIONAL OPERATORS: < <= > >= == ∼=**

( RO,   “<”)  // Lesser Than

( RO,   “<=” ) // Lesser Than Equal to

( RO,  “>”) // Greater Than

(RO , “>=”) // Greater Than Equal To

(RO , “==”) // Equal To Equal To

(RO , “!=”) // Not Equal To

**Identifiers start with a character followed by any number of characters, digits or underscore**

(ID , “num” )

(ID , “num1”)

(ID , “num\_one”)

**Numeric Constants**

(NC , ‘0’ )

**Literal Constant**

(LC , ‘d’ )

**Strings**

(String , “enter a number” )

**Separators**

(‘ ,’ , ^ )

( ‘;’ , ^ )

(‘:’ , ^)

(‘’’ , ^ )

(‘”’ , ^ )

**Input Operator**

(IO  ,    “->”)

**Parenthesis**

( ‘(‘ , ^ ) (  ‘)’ , ^ )

( ‘[‘ , ^)  (  ‘]’ , ^)

( ‘{‘ , ^)  ( ‘}’ , ^)

**Assignment Operator**

(AsO , ‘=’ )

**COMMENTS**

( “\\”  , ^)

(“/\*” , ^)

(“\*/” , ^)

Language will Ignore:

White Spaces

Tabs

Line Breaks(\n)

Single Line Comments(//)

Multi Line Comments(/\*\*/)

Regular Expression/Definition

**IDENTIFIER**  --->   letter(letter|digit | underscorel)\*

Digit->0|1|2|3|....|9

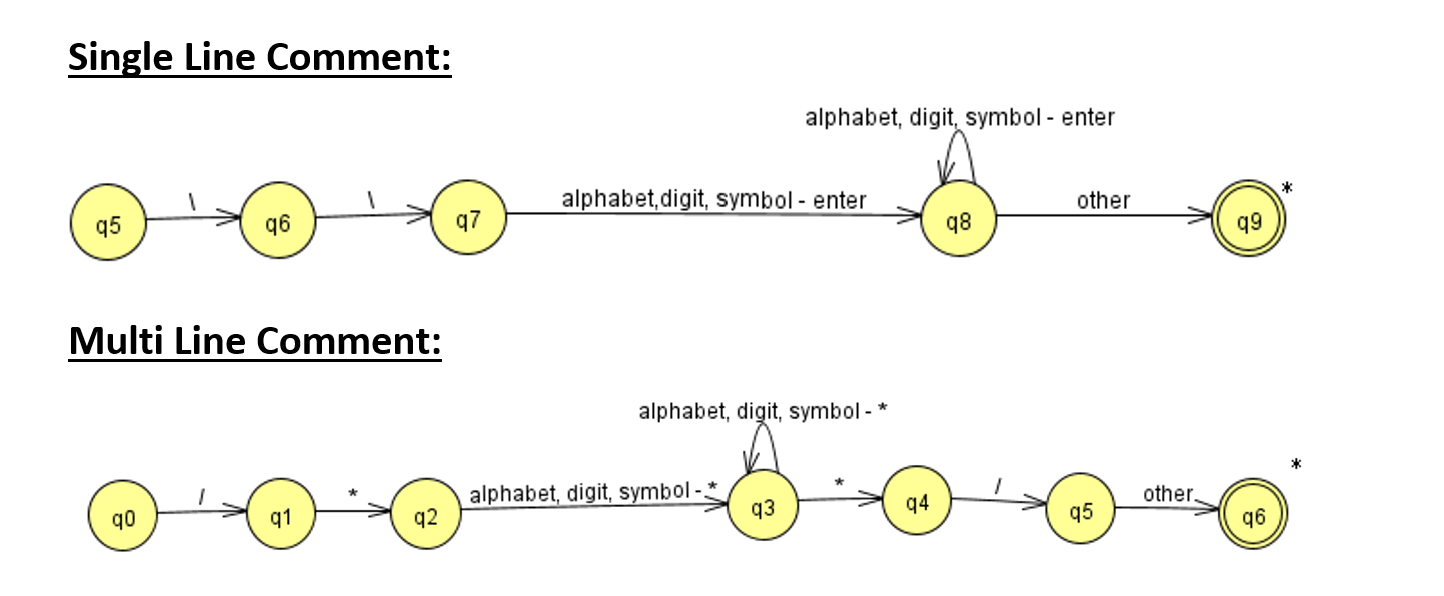
letter ->(A|B|.....|Z)(a|b|.....|z)

underscore -> \_

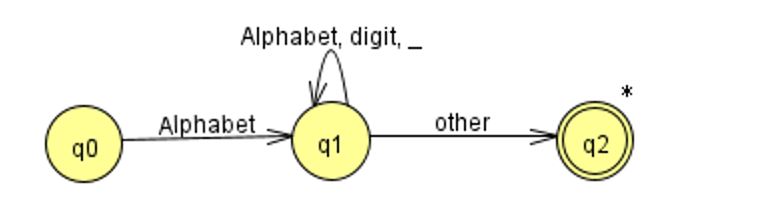
**RO** - >   <(= | ^)  |   >(= | ^)   |   =(= | ^)   |  ~=

Finite State Machines for complex tokens

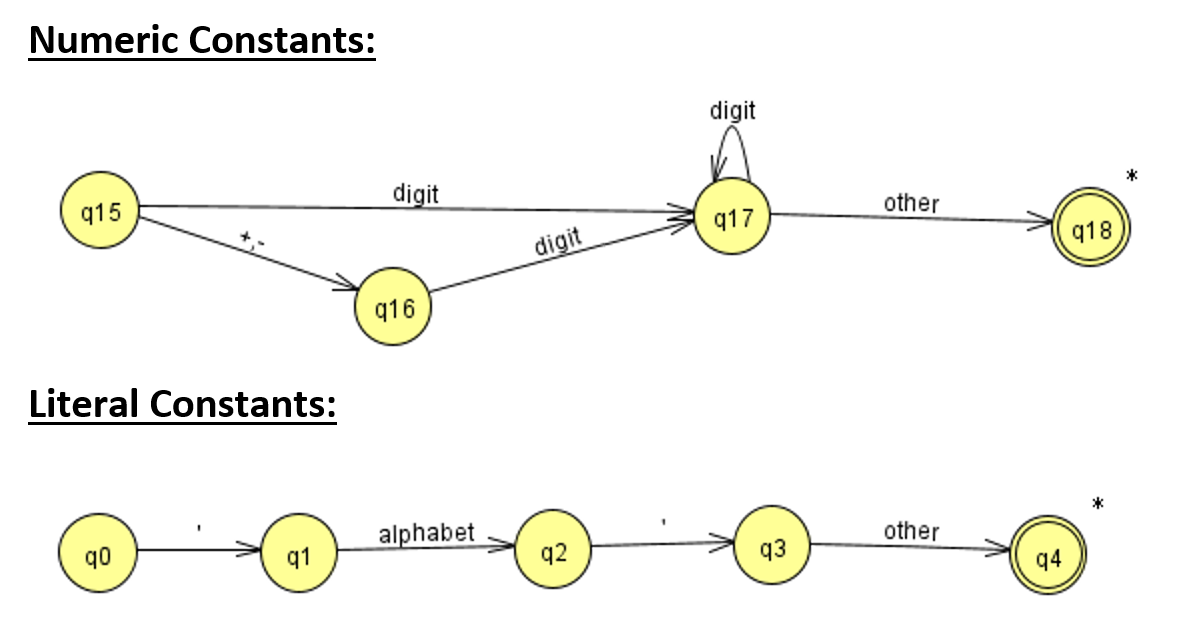
1. **Comments**

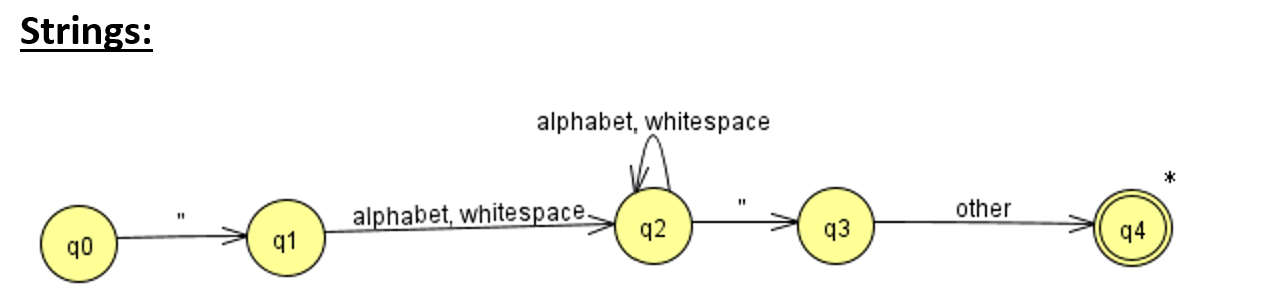


1. **Identifiers**

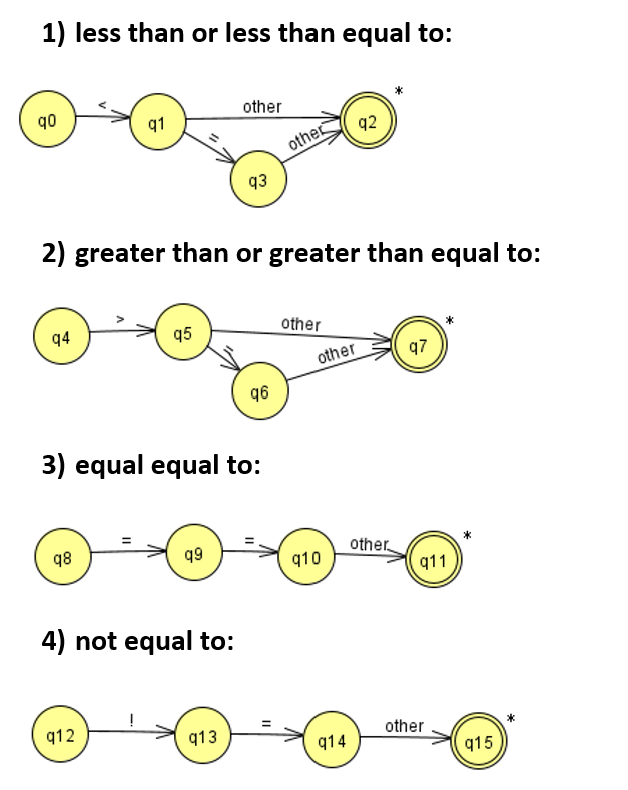


1. **Constants**





4. **Relational Operators**



**Parser:**

Context Free Grammar:

Arithematic Operators:

AO -> + | - | \* | /

Relational Operators:

RO -> < | <= | > | >= | == | !=

Input Operator:

IO -> ->

Increment / Decrement:

Inc/Dec -> ++ | --

Literal Constant:

LC -> Sigma //Sigma = any character from 0 – 255 ASCII

String:

String -> A String | ^

A -> LC

Numeric Constant:

NC -> B A

A -> NC | ^

B -> 0|1|....|9

IDENTIFIER:

ID -> B’ A’

A’ -> B’ A’ | C’ A’ | D’ A’ | ^

B’ -> A|B|....|Z|a|b|....|z

C’ -> 0|1|.....|9

D’ -> \_

AKW -> KWint AKW | KWchar AKW | KWif AKW | KWwhile AKW | KWinput AKW | KWprint AKW | KWprintln AKW | KWintegeridentifier AKW | KWcharidentifier AKW | ^

-------------------------------------------------------------------------------------------------------------

Int:

KWint -> int : A’;

A’ -> ID B’

B’ -> , A’ | = D’ C’ | ^

C’ -> ,A’ | ^

D’ -> ID | - ID | NC | - NC

Char:

KWchar -> char : A’;

A’ -> ID B’

B’ -> , A’ | = D’ C’ | ^

C’ -> ,A’ | ^

D’ -> ID | ‘ LC ’

Identifier:

KWintidentifier -> ID A’;

A’ -> AsO B’ |Inc/Dec

B’ -> ( B’ ) E’ | C’

C’ -> X’ D’

D’ -> AO F’ | ^

E’ -> AO G’ | ^

F’ -> Y’ D’ | B’

G’ -> Y’ E’ | B’

X’ -> ID | - ID | NC | -NC

Y’ ->ID | NC

KWcharidentifier -> ID AsO A’;

A’ -> ‘ LC ’ | ID

While:

KWwhile -> while A’ : { **AKW** }

A’ -> ID RO NC | NC RO ID

If:

KWif -> if A’ : { AKW } B’

A’ -> ID RO NC | NC RO ID

B’ -> KWelif B’ | KWelse | ^

elif:

KWelif -> elif A’ :{ AKW }

A’ -> ID RO NC | NC RO ID

else:

KWelse -> else:{ AKW }

Print:

KWprint -> print( A’ );

A’ -> “ String ” B’ | ID B’

B’ -> + A’ | ^

Println:

KWprintln -> println( A’ );

A’ -> “ String ” B’ | ID B’

B’ -> + A’ | ^

Input:

KWinput -> input A’;

A’ -> IO ID B’

B’ -> A’ | ^

**Translation Scheme:**

Input:

KWinput -> input A’;

A’ -> IO ID [print(“in” + ID.lex + “\n”)] B’

B’ -> A’ | ^

While:

KWwhile -> while A’ : { **AKW [emit(“goto”, BE.t]**} [backpatch(BE.f, n)]

A’ -> ID RO NC [BE.t = n; emit(“if ID.lex RO.lex NC.lex goto”, n + 2)

BE.f = n; emit(“goto”)]

Identifier:

KWintidentifier -> ID A’; [emit(Id.lex, A’.s)]

A’ -> AsO B’ [A’.s = AsO.lex + B’.s]

A’ ->Inc/Dec [A’.s = Inc/Dec.lex]

B’ -> ( B’ ) E’

B’ -> C’ [B’.s = C’.s]

C’ -> X’ D’ [C’.s = newTmp(); emit(C’.s, “=”,X’.s,D’.s)]

D’ -> AO F’ [D’.s = AO.lex + F’.s]

D’ -> ^ [D’.s = null]

E’ -> AO G’ | ^

F’ -> Y’ D’ [F’.s = newTmp(); emit(F’.s, “=”,Y’.s,D’.s)]

F’ -> B’ [F’.s = B’.s]

G’ -> Y’ E’ | B’

X’ -> ID | - ID | NC | -NC [X’.s = ID.lex]

Y’ ->ID | NC [Y’.s = ID.lex]