COMPILER CONSTRUCTION

# LEXICAL ANALYZER

Qazi Arsalan Shah

**Reg No :** 1802030 **UET :** 2018-UET-NML-CS-30

**Submitted to :** Mr Sadiqullah

## Background / Project Requirements

A program that performs lexical analysis and outputs a set of tokens. The input to the program will be a program written using the following specifications.

- Keywords: int, float, String, for, while, if, else
- Symbols & Punctuation: ;, (, ), {, }, ., -, +, *, /, <, >, <=, >=, == , &, !, |,

  The output of the program should be in the form <class, pointer>
- For example, for keywords: <int, >, <for,>, <id, pointer to symbol table>, <num, pointer to number table>
- For Symbols & Punctuation: <arop, ADD>, <relop, LE >, <SEMC, >. LE means less or equal, arop means arithmetic operation, relop means relational operation. You can also use your own naming convention
- The program should maintain a table for identifiers and a separate table for numbers. The program should be able to display the tables.
- The program should be able to report any lexical errors when the program run completely. It should also be able to continue lexical analysis even if an error occurred.

You will start by specifying regular definitions for the classes of tokens. After that you have to convert them into their corresponding NFAs and then from the NFAs you have to build the corresponding DFAs, all on paper. Once this process is completed, you have to represent all of the DFAs programmatically. Then you have to simulate the DFAs on real input program and the DFAs should be able to tell you whether the strings forming the program conform to the regular definitions or not. If yes, form the corresponding tokens as mentioned above.

When the program runs completely, display the list of all tokens, the symbol table, the number table and any error that has occurred.

## READ-ME FILE ( Related to Python Code ) :

Please refer to the read me file in the main project directory.

*Directory Structure : qazi_1802030_CC / CC_LX / readme.txt*

## Paper Work :

Below are pictures of all paperwork done.

CC Assignment

(*) Regular Expressions:

① Class keywords:

(*) keywords → int | float | string | for | while | if | else

int → int
float → float
string → string
for → for
while → while
if → if
else → else

"These are the reserved words and can't be used as identifiers."

"A separate finite automaton will be constructed for each" i.e. language recognizer"

② Class identifiers

id → letter (letter | digit)*

letter → [A-Z | a-z]
digit → [0-9]

③ Class numbers

digit → [0-9]

digits → digit+

number → digits

Accepting integer numbers

④ Class relational operator (relop)

relop → < | > | <= | >= | ==

⑤ Class Arithematic Operators

aprop → + | - | / | *

⑥ Class logical operators

logi → && | ! | ||

⑦ Punctuation Class

punct → ( | ) | { | } | . |

⑧ Class Semicolon

Semc → ;

(Note): Symbol table will be maintained for identifiers and number lable for numbers.

eg.

"Symbol Table"

| Reference | lexeme | Line # |
|---|---|---|
| 01 | abc | 2 |
| 02 | text | 8 |
| | | |

"Number Table"

| Reference | number | may be Line # |
|---|---|---|
| 01 | 20 | 2 |
| : | | |
| 07 | 161 | 21 |

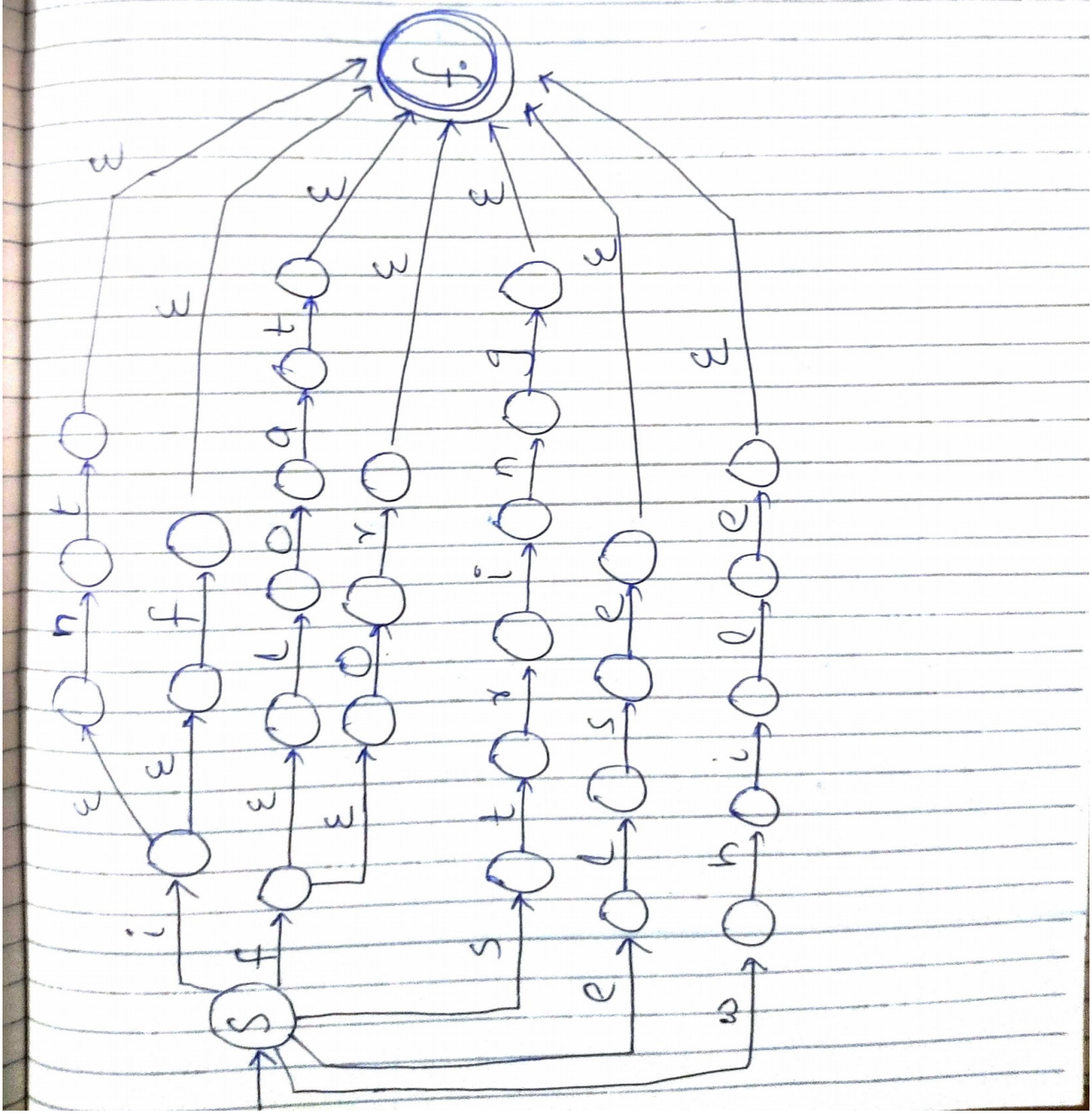
# Conversion of RE to NFA:

Used "Mc Naughton Yamada-Thompson algorithm"

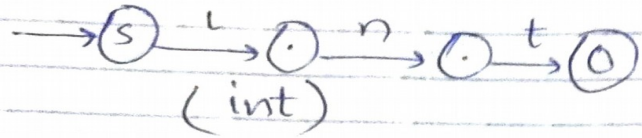
Basic idea, constred NFA for smaller parts then combine them

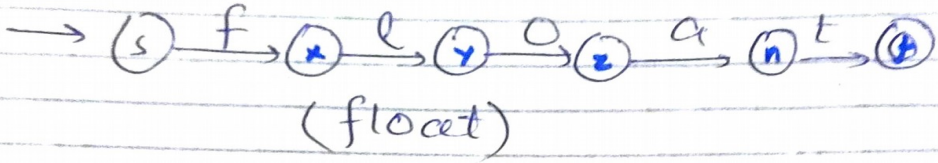
## *NFA for keywords Class:

So for general keyword class

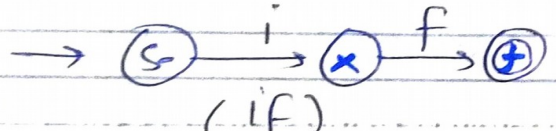


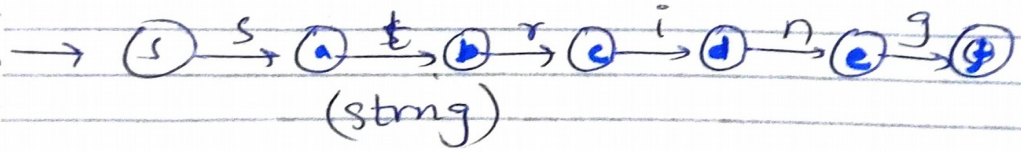
# "KEYWORDS"

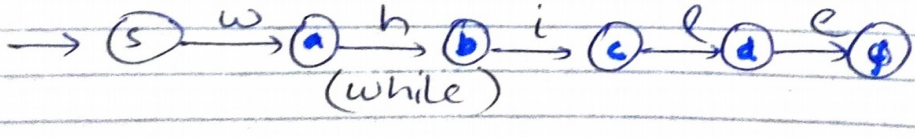

s
i
n
t
(int)

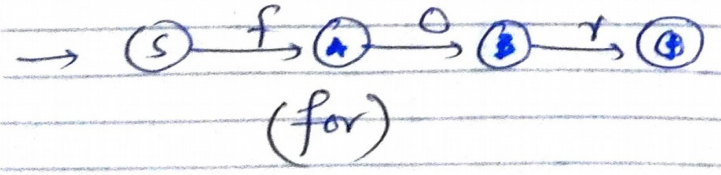

s
f
l
o
a
t
(float)


s
e
l
s
e
(else)


s
i
f
(if)


s
s
t
r
i
n
g
(strng)

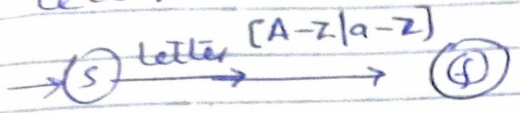

s
w
h
i
l
e
(while)


s
f
o
r
(for)

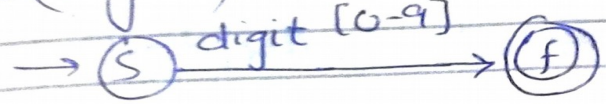
# NFA for identifiers :

id → letter (letter|digit)*

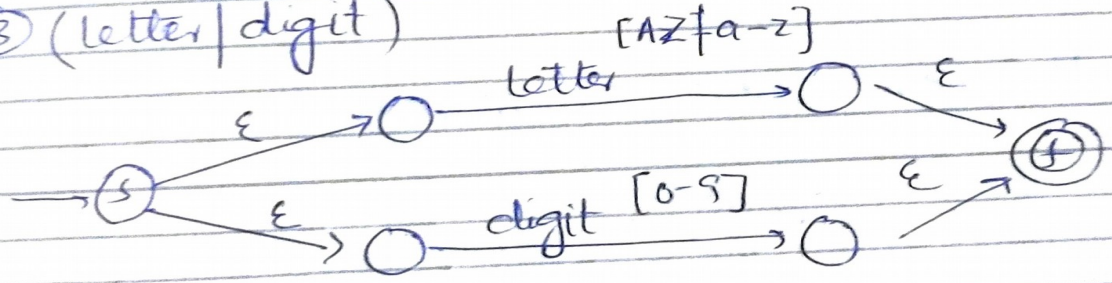
① 'letter'



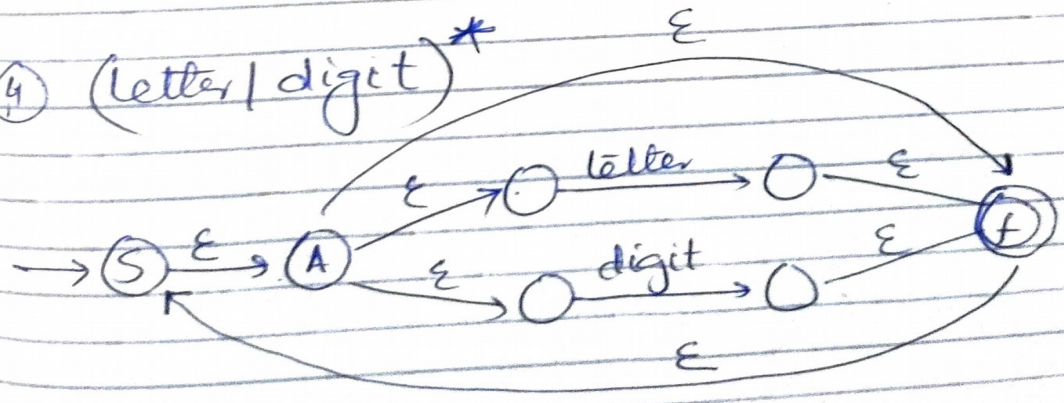

② (digit)




③ (letter|digit)




④ (letter|digit)*



Python Regex: "^[A-Za-z_]*$" → letter
"^[0-9]*$" → numbers

(5) letter (letter/digit)*

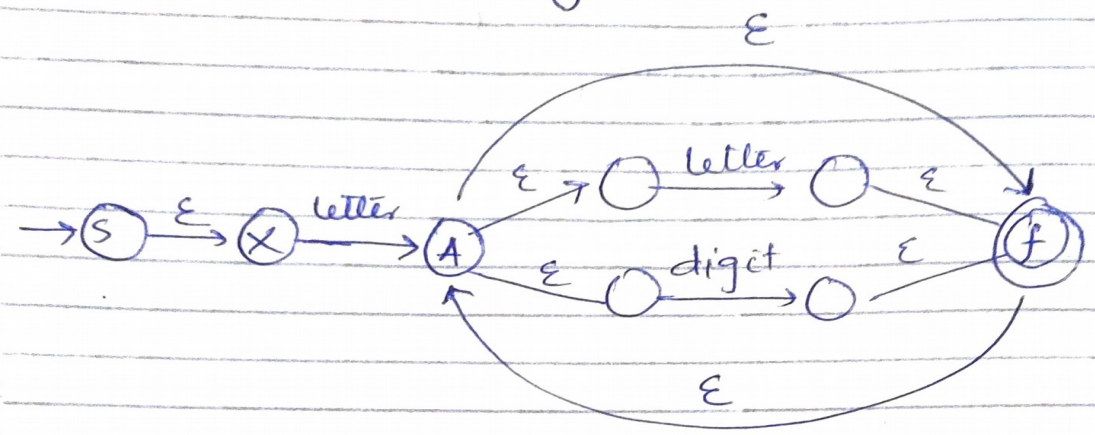



Fig: id → letter (letter/digit)*

---

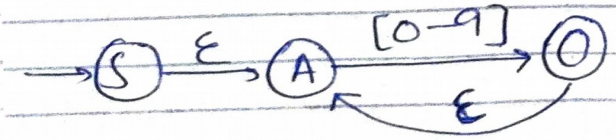
## NFA for Class Numbers:

number → digits

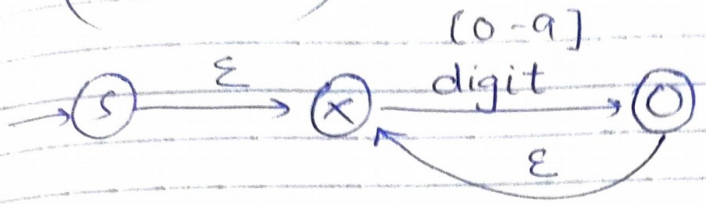
digit → [0-9]

digits → digit+

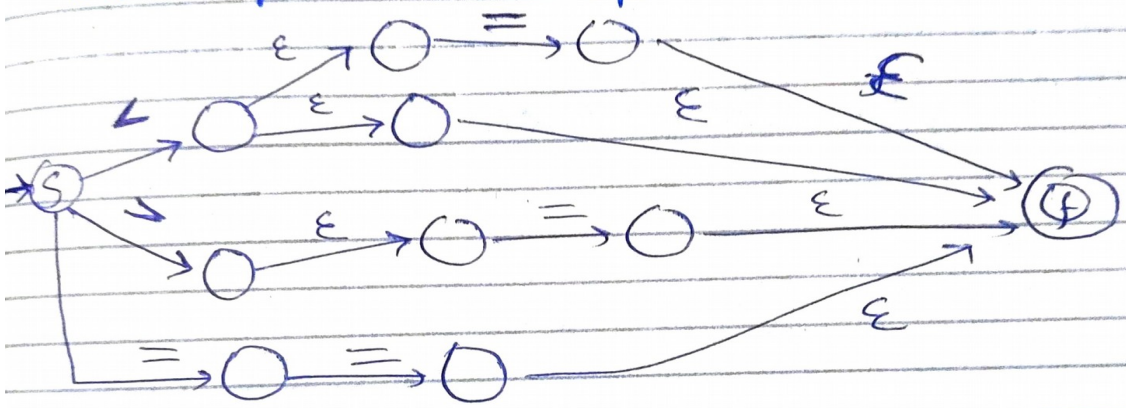
(digits)



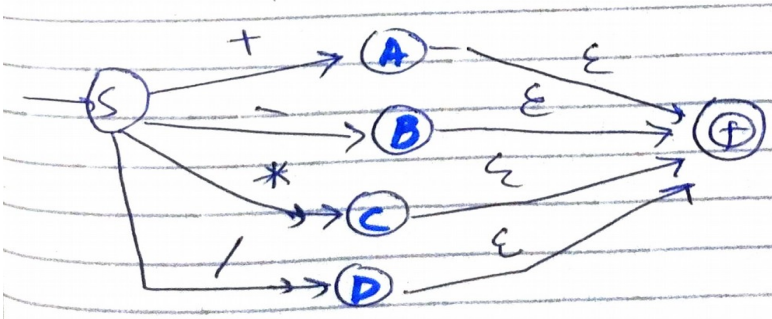
(number)



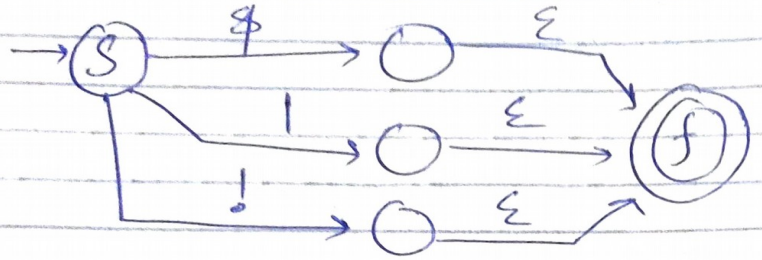

## NFA for Relop:



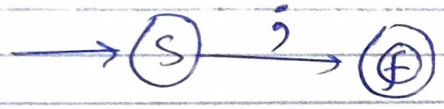

## NFA for Arop



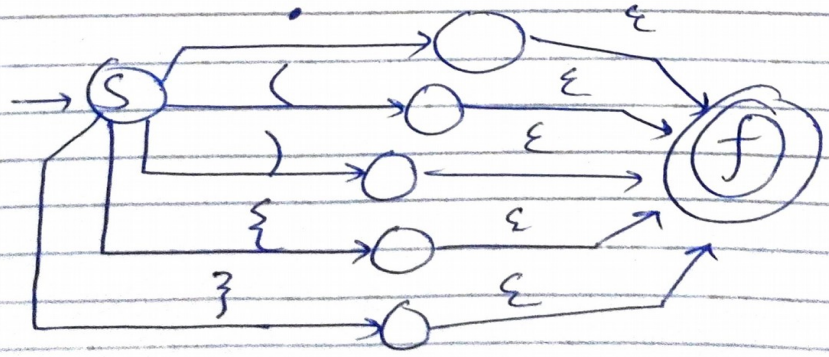
## CYFA for class logical operators :




## NFA for Semicolon




## NFA for Punctuations



# Conversion of NFA to DFA:

As lexical analyzer use DFA as a language recognizer | token recognizer.

we can convert an NFA → to DFA using subset construction method.

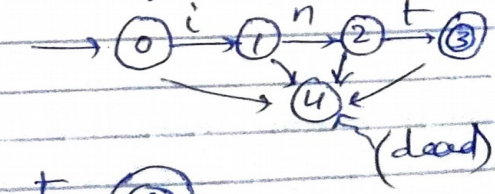
Idea, a Dfa state can be set of NFA states, and start state of DFA = $\epsilon$-closure (So Nfa).
(∵ reference book)

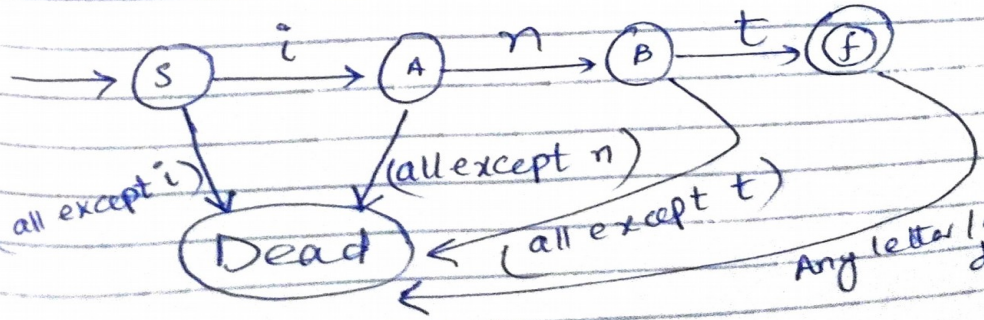
"All NFA ares converted to DFA keeping all required steps and rules in mind."

## DFA for class Keywords:

There will be seperate Dfa for every keyword (reserved).

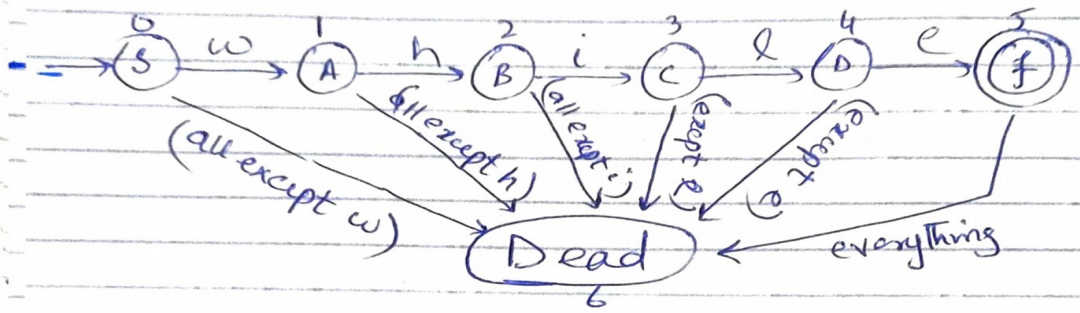
"int"



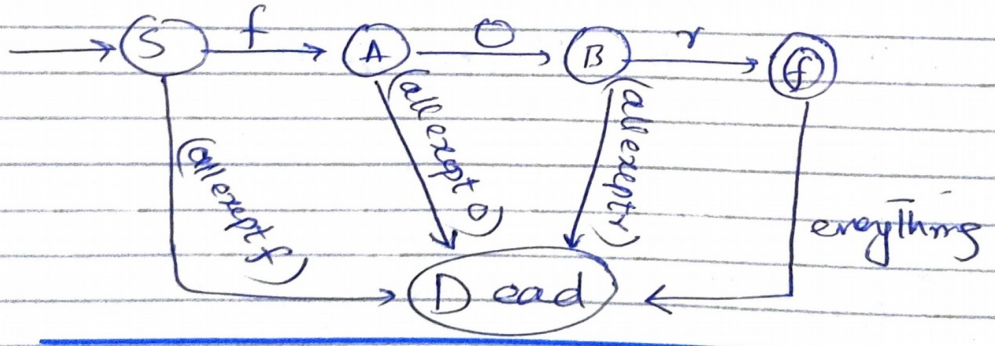



it will only except i → n → t.
And will go to regeet state if ~~this~~ this order is ~~disturbed~~ disturbed.
Dead state is placed in DFA.

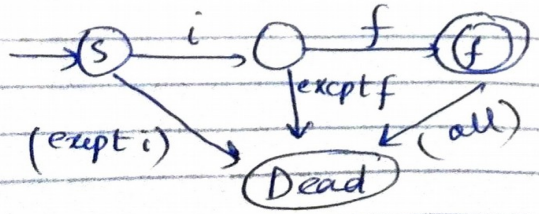
"while"



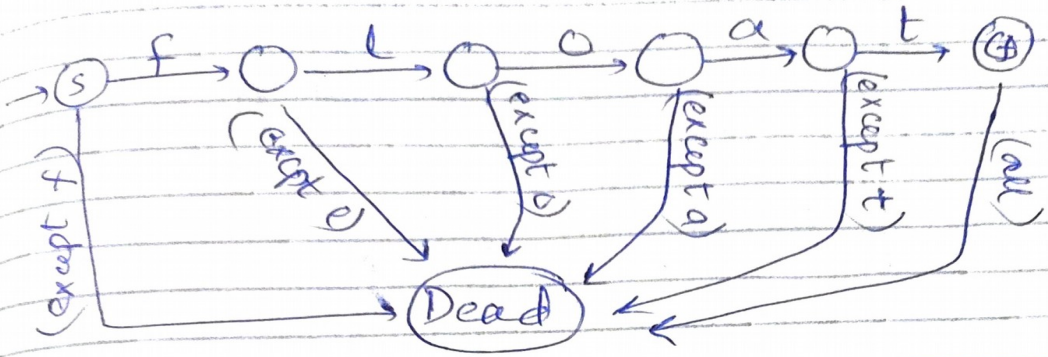

"for"



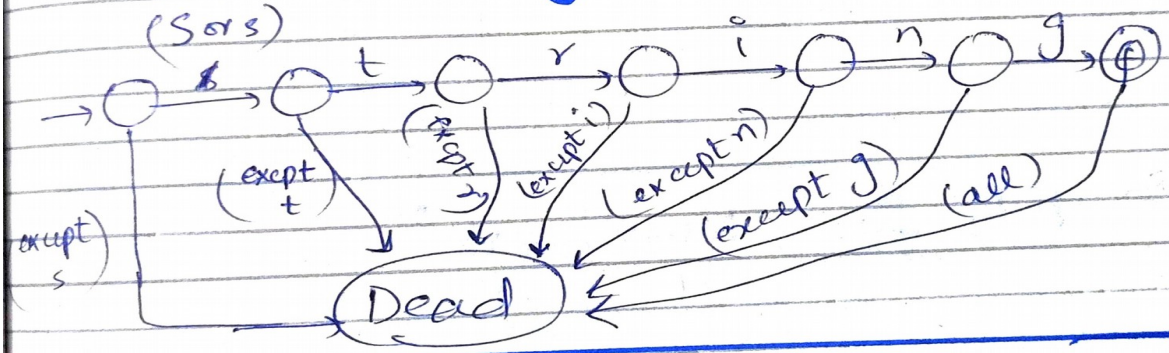

"if"



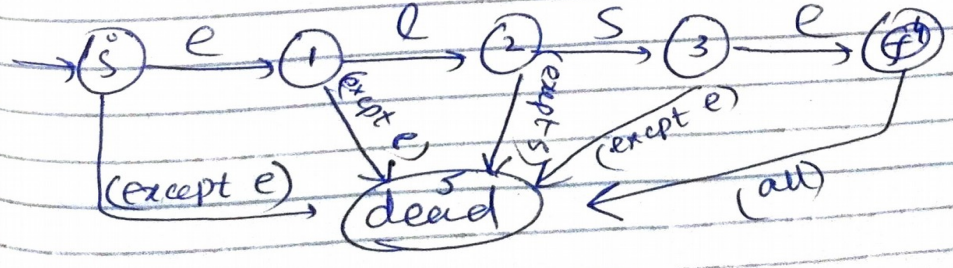
## "float"




## "String"



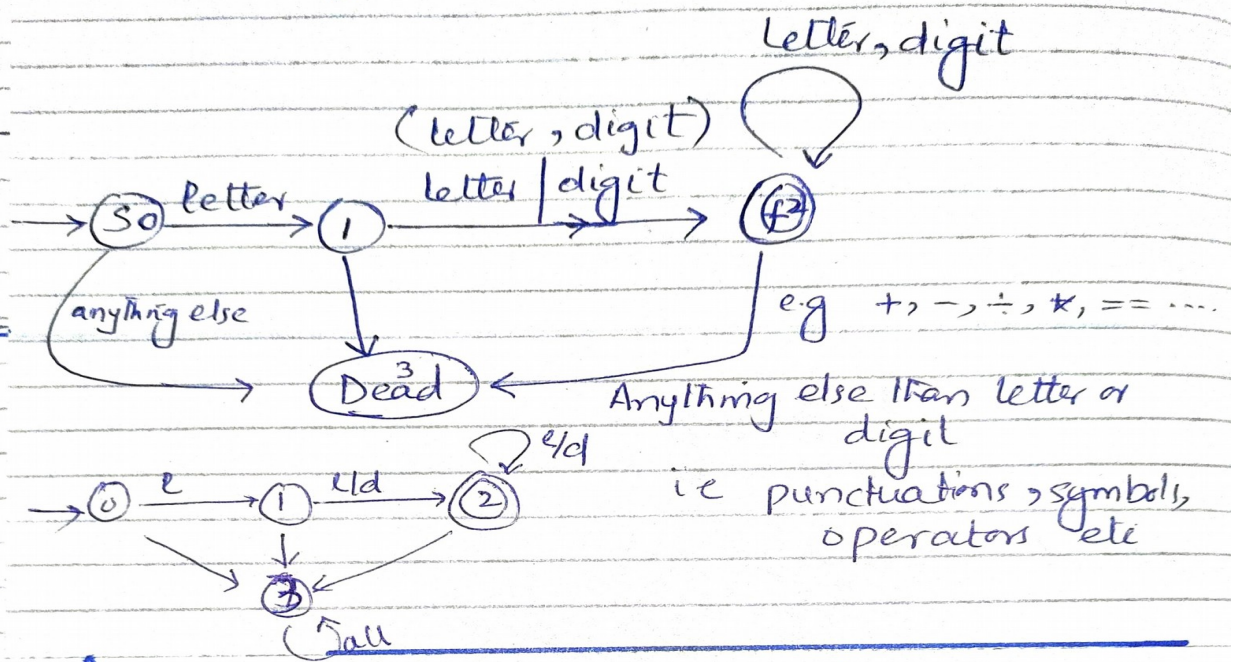

## "else"



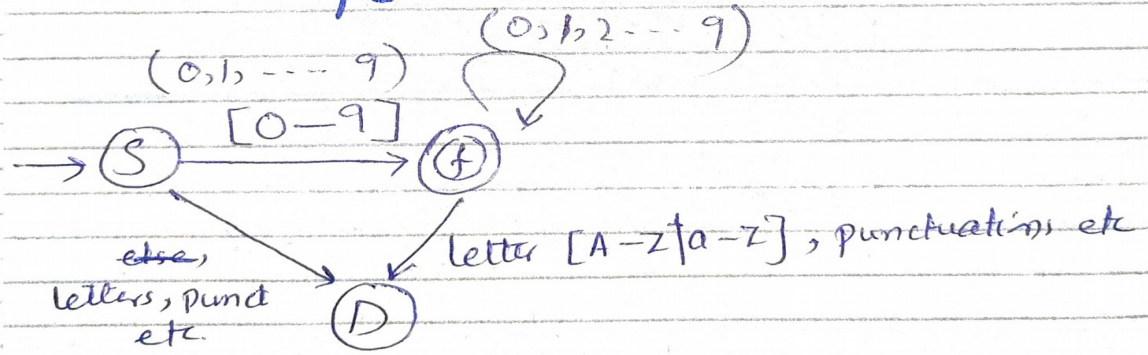
for Letters : [A-Za-z_] & Letter|digit : [A-Za-z0-9_]
digits : [0-9]



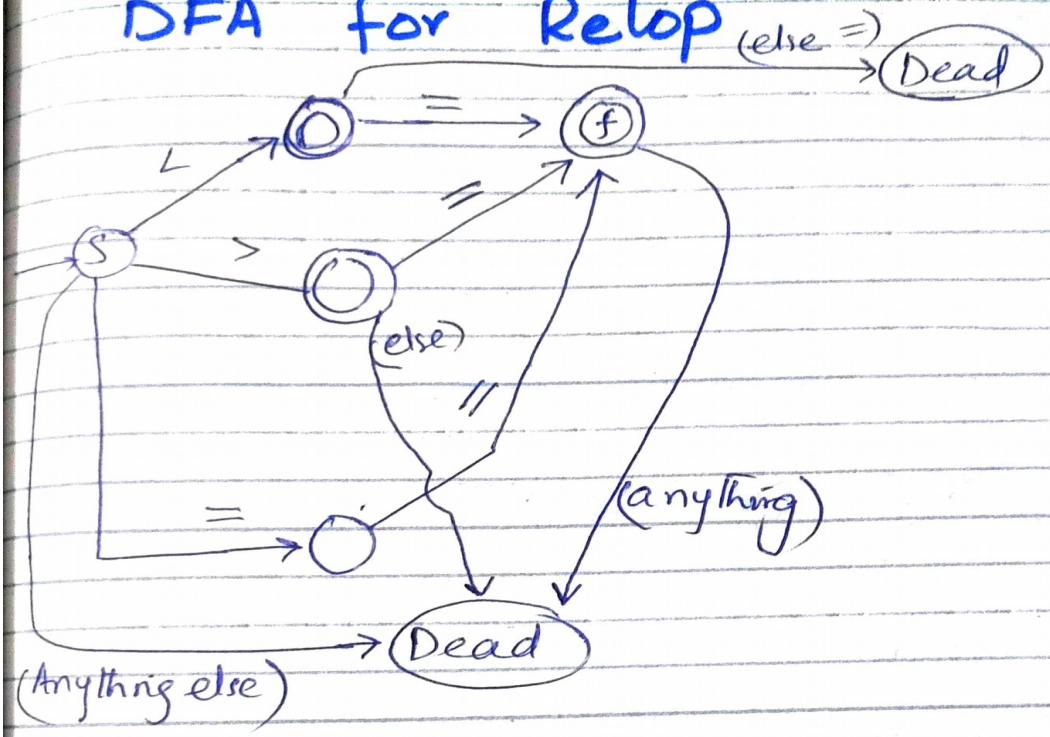
## "DFA for identifier"



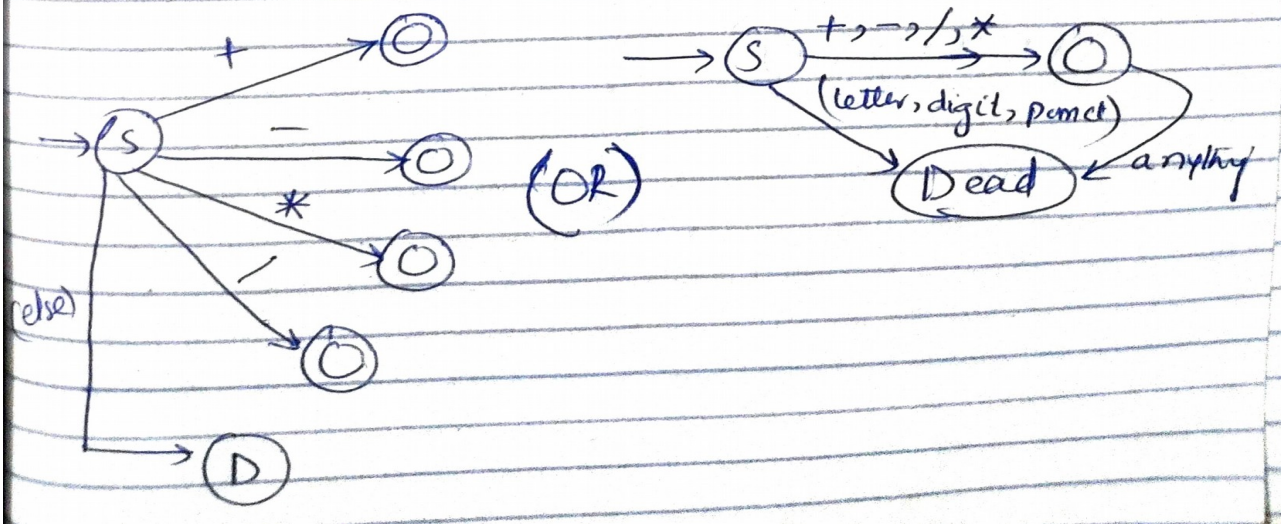

## "DFA for Numbers"



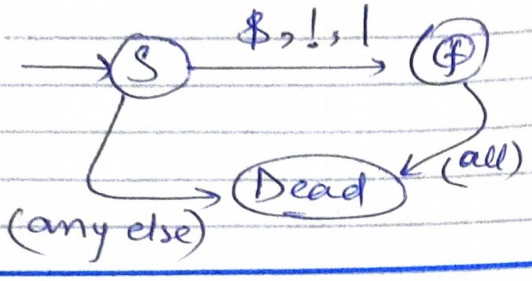
## DFA for Relop



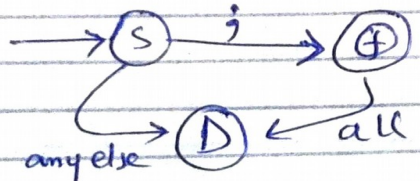

## DFA for Arop (+, -, /, *)



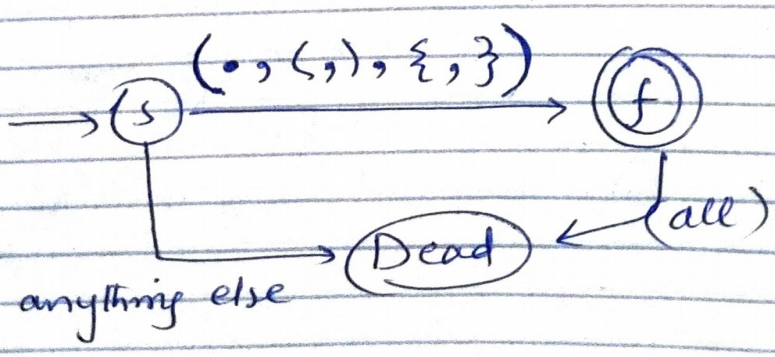
# DFA for Logical Operators :




# DFA Semicolon




# (DFA Punctuations)



(Note:)

for the sake of keeping the transition diagrams simple in case of DFA's, i used shorthand terms like,

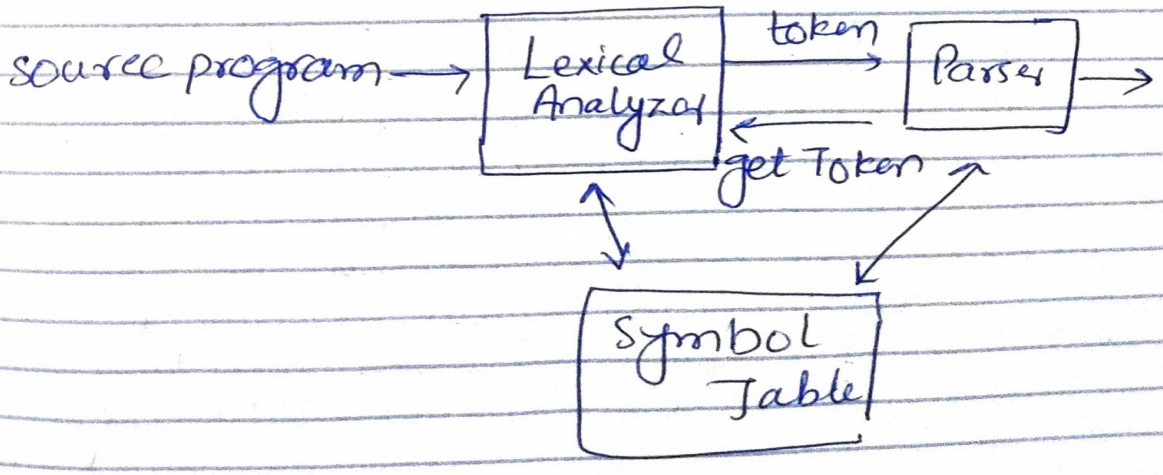
all → it means, in said context all letters, digits, punctuations.

&

except a → it means all, except a. ie a is excluded from it.

letter → [A, B, C --- Z | a, b, c --- z]

digit → [0, 1, 2 --- 9]



* Lexical Analysis
  (1) Token specification (patterns)
  (2) Recognition (matching)

## DFA Representation in Code:

(*) DFA can be coded using graph theory easily.

e.g dfa = {0: [1, 3], -- }

it means, if we are on '0' by reading certain input according to transition table, we can go to either '1' or '3'.

(*) But in this, I have implemented them in simple programatic way.

(*) All DFA's are coded as seperate modules and are placed in "modules/" directory

# PYTHON CODE DETAILS:

I have made three dictionaries

1) SymbolTable = {}
2) ErrorTable = {}
3) NumberTable = {}

Python code reads from file string of characters.

(*) It first reads a complete line to keep track of Line number

(*) Then it splits the line into words and then each word is checked will all legal DFAs.

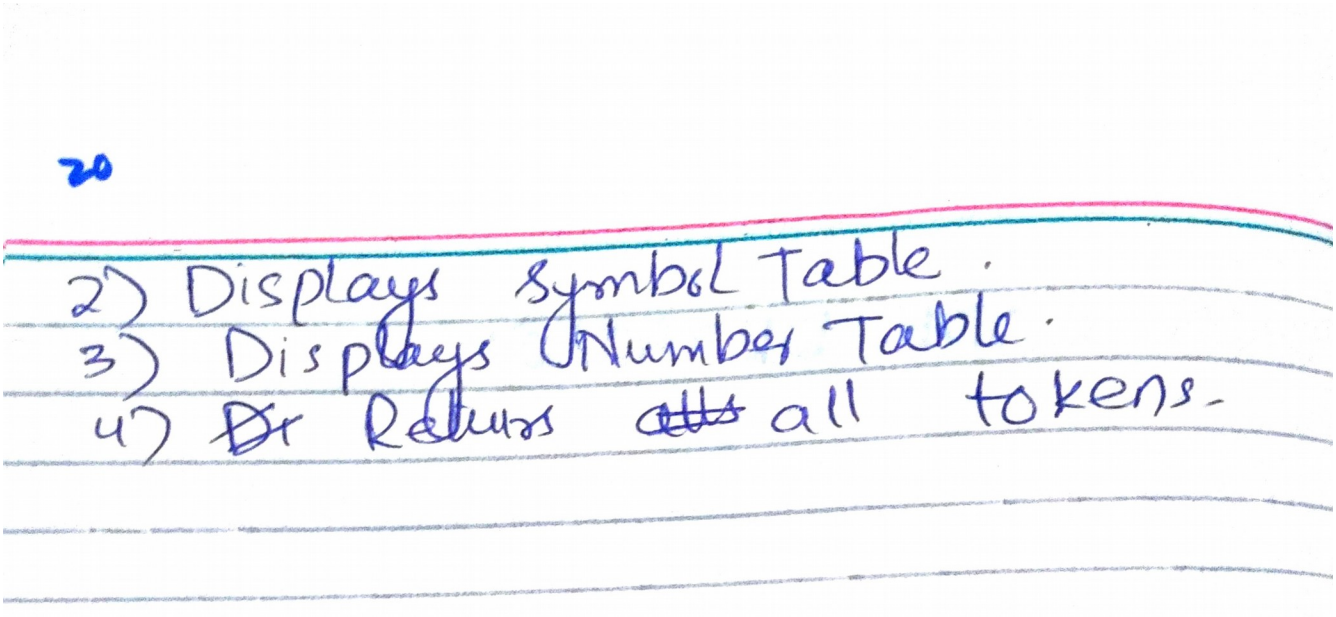
(*) if isdigit → no need to check for keywords or identifier etc.

(*) Else check for keywords, then punctuations and etc.

(*) Then check for identifiers.

(*) **OUTPUT:**

1) Displays all Errors occured.




## PYTHON CODE:

Main code file is in the following directory.
“ qazi_1802030_CC / CC_LX / lexical_Analyzer.py “
Place test code ( Source program ) in test.txt file. Python program will read the text file and display its result.

## Important Code Explanation:

1 . All DFAs are coded as separate modules and are imported as modules.

lexical_Analyzer.py

```
1   # DFAs FOR ALL CLASSES OF TOKENS
2   # DIRECTORY modules/
3   # Return True ( if accepeted ) else False
4   from modules.semicolon import semicolonDFA
5   from modules.punctuations import punctuationDFA
6   from modules.logicalOp import logicalOpDFA
7   from modules.arithematic import aropDFA
8   from modules.relop import relopDFA
9   from modules.numbers import numDFA
10  from modules.identifiers import idDFA
11  from modules.int import intDFA
12  from modules.elseD import elseDFA
13  from modules.ifD import ifDFA
14  from modules.forD import forDFA
15  from modules.whileD import whileDFA
16  from modules.stringD import stringDFA
17  from modules.floatD import floatDFA
18  # ============================================================
```

2. Symbol Table , Error Table and Number Table is implemented as a python dictionary.

```
20  # SYMBOL TABLE
21  """
22      no. 1 | arsalan | line 1
23
24  """
25  symbolTable = {}
26
27  # ================================
28
29  # NUMBER TABLE
30  """
31      no. 1 | 20 | line 2
32
33  """
34  numberTable = {}
35
```

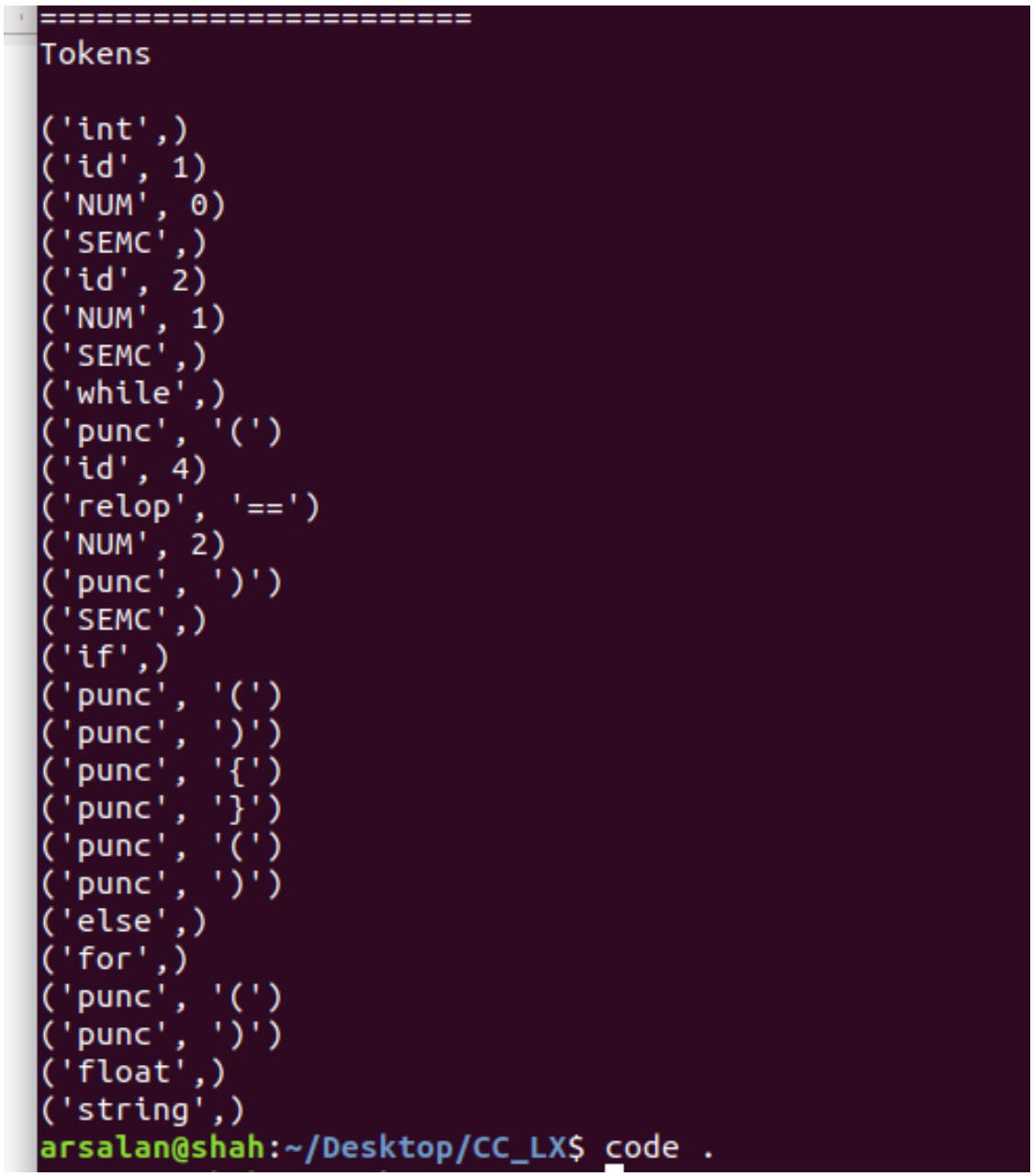
```
37  # ERROR TABLE
38  """
39      no. 1 | NAME | line 2
40
41  """
42  errorTable = {}
43  errorno = 0
```

3. Tokens are returned in python list. As list of tuples.
   e.g Tokens = [(int,),(id,01),…..]
4. For Further code reading please refer to code file.

5. Here is a sample output of code.

```
arsalan@shah:~/Desktop/CC_LX$ python3 lexical_Analyzer.py
============================================
Errors Occured In Program 2

Here is the Error List:
{0: ['=', 'Error in Line : 1'], 1: ['=', 'Error in Line : 2']}
============================
Numbers Table is  Below
{0: [20, 'Line No :1'], 1: [30, 'Line No :2'], 2: [3, 'Line No :3']}
=========================
Here is The Symbol Table
{0: ['int', 'Line No : 1, RESERVED WORD'], 1: ['number', 'Line No :1'], 2: ['ide
ntifier', 'Line No :2'], 3: ['while', 'Line No : 3, RESERVED WORD'], 4: ['num',
'Line No :3'], 5: ['if', 'Line No : 4, RESERVED WORD'], 6: ['else', 'Line No : 7
, RESERVED WORD'], 7: ['for', 'Line No : 8, RESERVED WORD'], 8: ['float', 'Line
No : 9, RESERVED WORD'], 9: ['string', 'Line No : 10, RESERVED WORD']}
```




SAMPLE INPUT: