**COVER PAGE**

# **CS323 Programming Assignments**

**Fill out all entries 1 - 7. If not, there will be deductions!**

**Check one**

1. Names [ 1. Abhishek Mhatre], (4pm class [ ] or 5:30pm class [ X ] )

[ 2. Aishwarya Iyer ], (4pm class [ X ] or 5:30pm class [ ] )

2. Assignment Number [ 1 ]

3. Due Dates **Softcopy**  [ 3/5 ], **Hardcopy** [ 3/6 ]

4. Turn-In Dates **Softcopy** [ 3/5 ], **Hardcopy** [ 3/6 ]

5. Executable FileName [ ]

(**A file that can be executed without compilation by the instructor**)

6. LabRoom [ ]

**(Execute your program in a lab in the CS building before submission)**

7. Operating System [ ]

**To be filled out by the Instructor:**

GRADE:

COMMENTS:

**Assignment 1 Documentation**

1. **Problem Statement**

Write a lexical analyzer for RAT18S language.

2. **How to use program**

To use this program, open the directory containing the entire program file. In order for the lexer to generate an output, open the text file and input your code in the RAT18S language. The default input is the sample input provided by the assignment instructions. After inputting the sample code, save and close the file. Next, open the file named “**lexical\_analysis.py**”. Run this python file. After running, an output file containing a list of all the tokens and their values should be listed. If a word or character does not match the lexical rules, it will not be listed in “**lexical\_analysis.py**”. To repeat this process, simply change the content of the text file and run the python file to get an output.

3. **Design of your program**

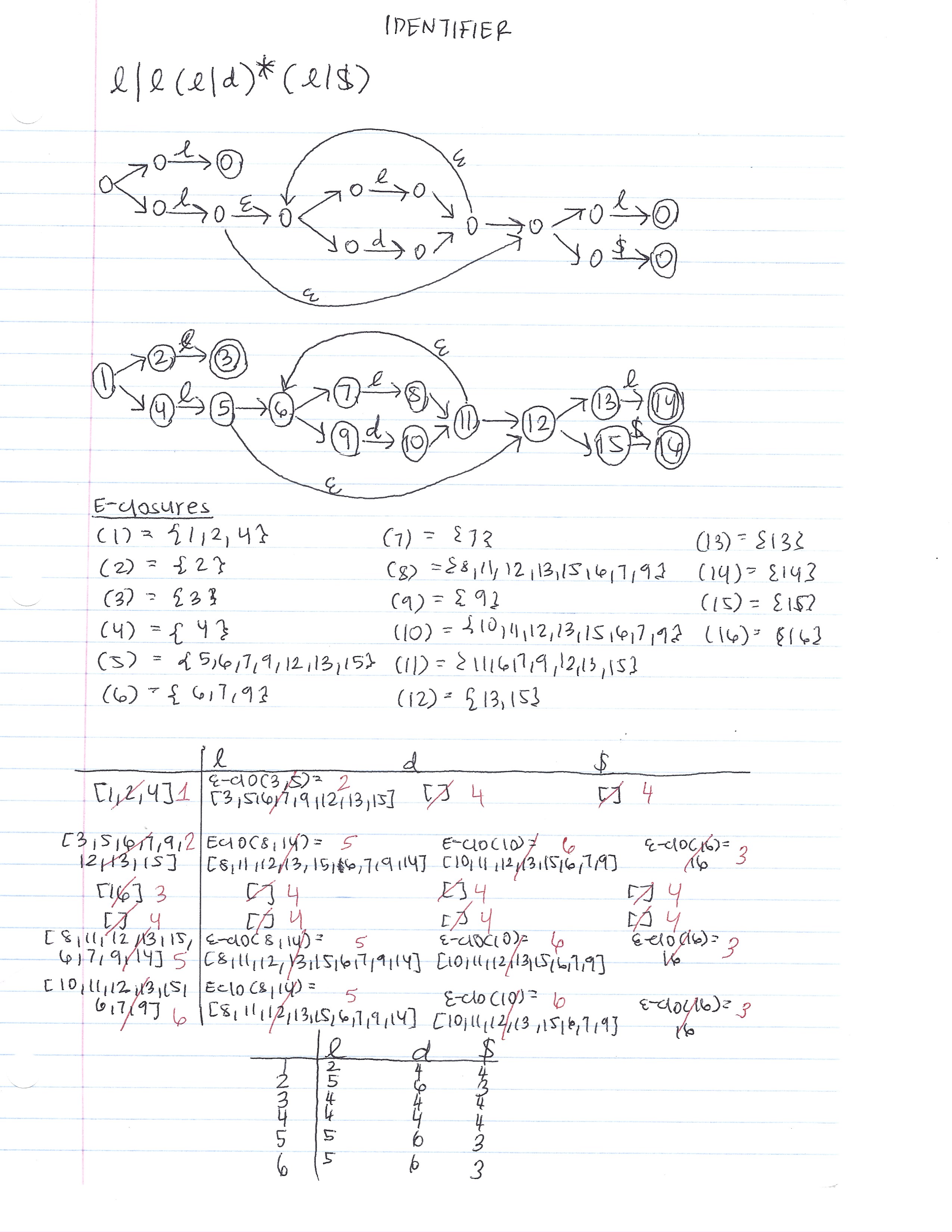
The program is compatible with Python 3.6+ and consists of a buffer function and a lexer function in order to create the lexical analyzer. As a group, we used a buffer function called “createBuffer” to break down each line into separate words (tokens). Once this is done, we created a function called “lexer” which passes createBuffer. In this function we take the broken down lines and determine whether or not the token is a separator, identifier, a comment, integer or real.

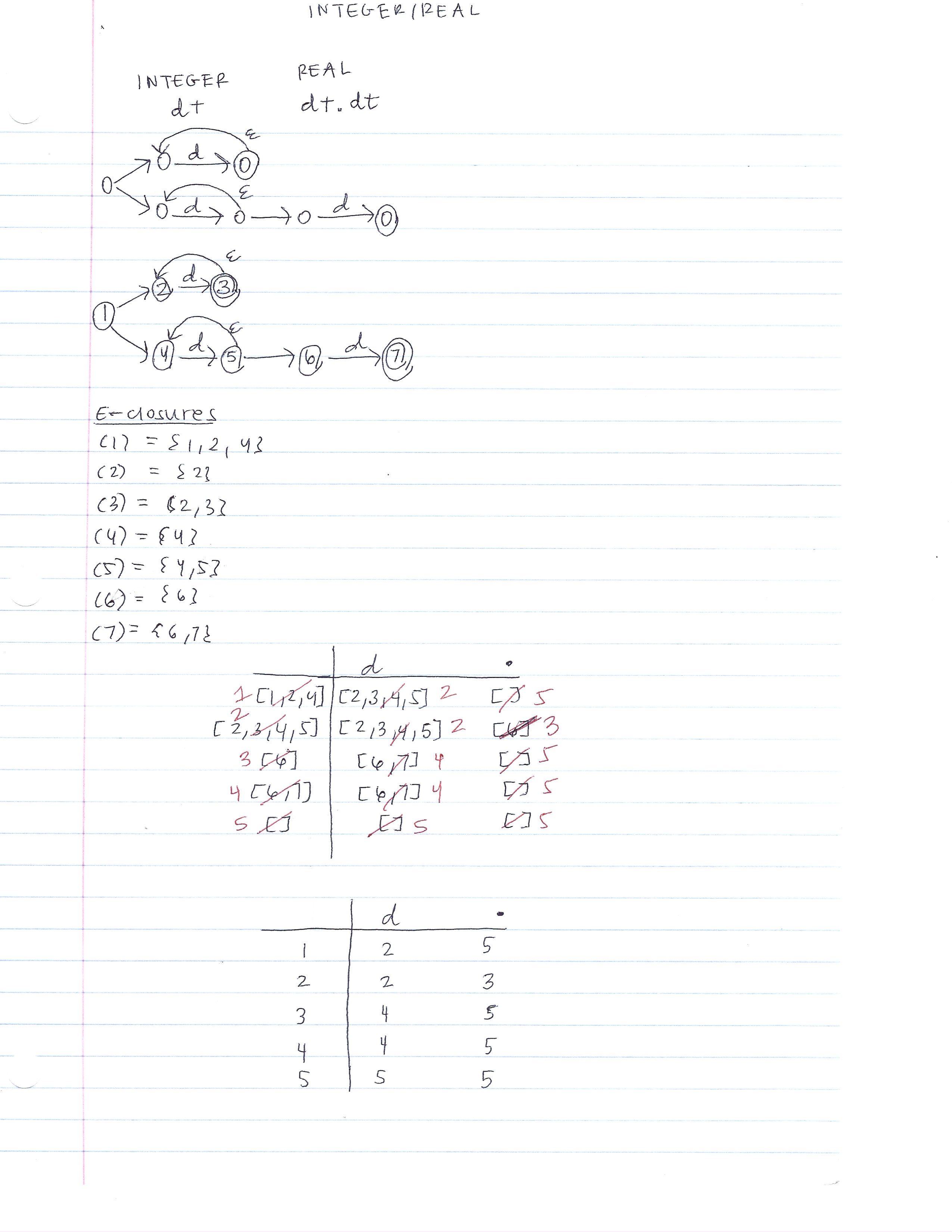
4. **Limitation**

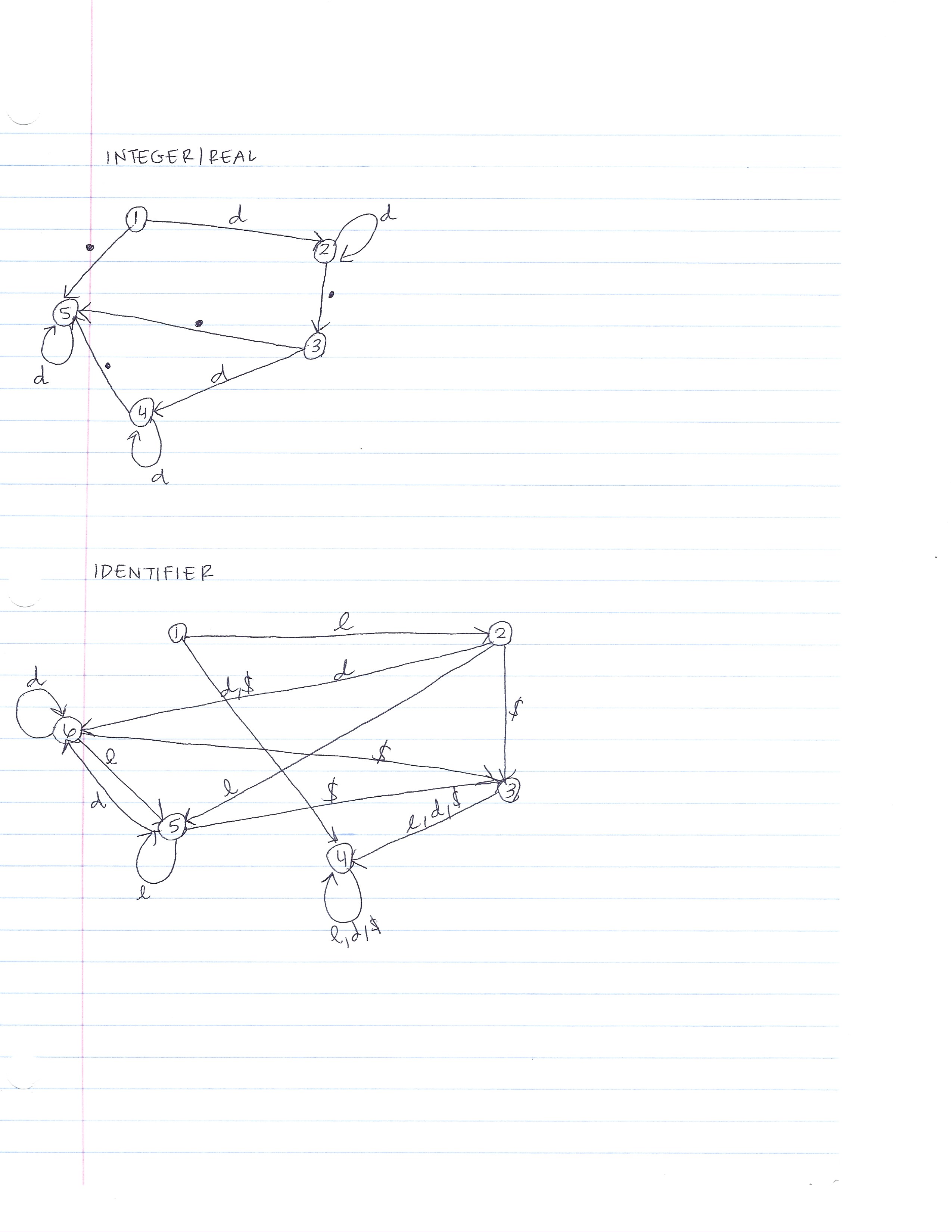
None

5. **Shortcomings**

None







**Source Code**

|  |  |
| --- | --- |
|  | ##################################################################### |
|  | ## CPSC323 Assignment 1 (Lexical Analysis) ## |
|  | ## Team Members: ## |
|  | ## 1. Aishwarya Iyer ## |
|  | ## 2. Abhishek Mhatre ## |
|  | ## ## |
|  | ##################################################################### |
|  |  |
|  |  |
|  | #Keywords |
|  | Keywords = ["int", "float", "boolean" ,"if", "else", "endif", "while", "return", "get", "put","function","for"] |
|  | #Separators |
|  | Separators = ["{", "}", "[", "]", "(" ,")", ";", ",",":"] |
|  | #Operators |
|  | Operators = ["+", "-", "\*", "/", ">", "<", "=","!","&" ] |
|  | #Double Operators |
|  | Double\_Operators = ["+=","-=","\*=","/=","%=","==","<=",">=","!="] |
|  |  |
|  | # File name without extension |
|  | source\_filename = "" |
|  | # Machine States |
|  | buffer\_token = "" |
|  | Token\_Array = [] |
|  | Lexeme\_Array = [] |
|  | Buffer\_Array =[] |
|  |  |
|  | def crateBuffer(char): |
|  | global buffer\_token |
|  | #identifier/ keyword/ integer/ real |
|  | if char.isalpha() or char.isdigit(): |
|  | if buffer\_token !="" and buffer\_token[-1] in Operators: |
|  | Buffer\_Array.append(buffer\_token) |
|  | buffer\_token = "" |
|  | buffer\_token +=char |
|  | #identifier (last letter of the identifier should be a letter or a $ sign) |
|  | elif char =="$": |
|  | buffer\_token+=char |
|  | #all characters in the buffer are digit and '.' (dot) appeared--> possible real |
|  | elif buffer\_token.isdigit() and char ==".": |
|  | buffer\_token+=char |
|  | #Separator |
|  | elif char in Separators: |
|  | Buffer\_Array.append(buffer\_token) |
|  | Buffer\_Array.append(char) |
|  | buffer\_token = "" |
|  | #Operator |
|  | elif char in Operators: |
|  | if buffer\_token =="": |
|  | buffer\_token+=char |
|  | # Buffer\_Array.append(buffer\_token) |
|  | #already an operator present in the buffer --> could be a double operator |
|  | elif buffer\_token != "" and buffer\_token[-1] in Operators: |
|  | buffer\_token+=char |
|  | elif buffer\_token[-1].isalpha() or buffer\_token[-1].isdigit(): |
|  | Buffer\_Array.append(buffer\_token) |
|  | buffer\_token = char |
|  | else: |
|  | buffer\_token+=char |
|  | Buffer\_Array.append(buffer\_token) |
|  | buffer\_token = "" |
|  | # Unknown |
|  | elif char==" " or char=="\n" or char =="\t": |
|  | if char =="!": |
|  | buffer\_token+= char |
|  | Buffer\_Array.append(buffer\_token) |
|  | buffer\_token ="" |
|  |  |
|  | ###Function to # |
|  | def lexer(Buffer\_Array): |
|  | token = "" |
|  | lexeme = "" |
|  |  |
|  | for char in Buffer\_Array: |
|  | #If its an empty space, just ignore it |
|  | if len(char) > 0: |
|  | #Process |
|  | for c in char: |
|  | #Beginning of comment section |
|  | if c =="!" and (token =="Unknown" or token ==""): |
|  | token = "Comment" |
|  | #End of the comment section |
|  | elif c =="!" and token =="Comment": |
|  | token ="" |
|  | #Identifier/ Keyword |
|  | elif c.isalpha(): |
|  | #Its a part of comment section, ignore it |
|  | if token =="Comment": |
|  | break |
|  | elif token=="" or token=="Identifier": |
|  | lexeme +=c |
|  | token = "Identifier" |
|  | #Identifier |
|  | elif c =="$" and token =="Identifier": |
|  | #Its a part of comment section, ignore it |
|  | if token =="Comment": |
|  | break |
|  | else: |
|  | lexeme +=c |
|  | token = "Identifier" |
|  | #Integer/ Real |
|  | elif c.isdigit(): |
|  | #Its a part of comment section, ignore it |
|  | if token =="Comment": |
|  | break |
|  | elif token =="" or token =="Integer": |
|  | lexeme+=c |
|  | token = "Integer" |
|  | elif token =="Real": |
|  | lexeme +=c |
|  | token = "Real" |
|  | else: |
|  | lexeme +=c |
|  | token = "Unknown" |
|  | # Real |
|  | elif c ==".": |
|  | #Its a part of comment section, ignore it |
|  | if token =="Comment": |
|  | break |
|  | #Its a separator or an unknown |
|  | elif token == "Identifier" or token =="Unknown": |
|  | Token\_Array.append(token) |
|  | Lexeme\_Array.append(lexeme) |
|  | token = "" |
|  | lexeme= "" |
|  | #its a real value |
|  | elif token == "Integer": |
|  | lexeme +=c |
|  | token = "Real" |
|  | else: |
|  | Token\_Array.append(token) |
|  | Lexeme\_Array.append(lexeme) |
|  | token = "" |
|  | lexeme= "" |
|  | elif c in Separators: |
|  | #Its a part of comment section, ignore it |
|  | if token =="Comment": |
|  | break |
|  | else: |
|  | token = "Separator" |
|  | Token\_Array.append(token) |
|  | token = "" |
|  | lexeme += c |
|  | Lexeme\_Array.append(lexeme) |
|  | lexeme = "" |
|  | elif c in Operators: |
|  | #Its a part of comment section, ignore it |
|  | if token =="Comment": |
|  | break |
|  | elif token=="": |
|  | token = "Operator" |
|  | lexeme += c |
|  | else: |
|  | token = "Operator" |
|  | Token\_Array.append(token) |
|  | token = "" |
|  | lexeme += c |
|  | Lexeme\_Array.append(lexeme) |
|  | lexeme = "" |
|  | if token =="Identifier" and lexeme in Keywords: |
|  | token = "Keyword" |
|  | Token\_Array.append(token) |
|  | token = "" |
|  | Lexeme\_Array.append(lexeme) |
|  | lexeme = "" |
|  | elif token =="Identifier" or token=="Integer" or token=="Real" or token =="Separator" or token =="Operator" or token =="Unknown": |
|  | Token\_Array.append(token) |
|  | token = "" |
|  | Lexeme\_Array.append(lexeme) |
|  | lexeme= "" |
|  |  |
|  | elif token =="Comment": |
|  | pass |
|  | else: |
|  | if token =="Identifier" and lexeme in Keywords: |
|  | Token\_Array.append(token) |
|  | token = "" |
|  | Lexeme\_Array.append(lexeme) |
|  | lexeme = "" |
|  | elif token =="Identifier" or token =="Unknown": |
|  | Token\_Array.append(token) |
|  | token = "" |
|  | Lexeme\_Array.append(lexeme) |
|  | lexeme= "" |
|  | elif token =="Comment": |
|  | pass |
|  | else: |
|  | Token\_Array.append(token) |
|  | token = "" |
|  | Lexeme\_Array.append(lexeme) |
|  | lexeme = "" |
|  |  |
|  | #Clear the buffer for the next line |
|  | Buffer\_Array.clear() |
|  |  |
|  | #### End of lexer() #### |
|  |  |
|  |  |
|  |  |
|  | #### Function to print Tokens and Lexemes table into a file #### |
|  | def print\_to\_file(): |
|  | outputFile = open(source\_filename+"\_lexer\_output" + ".txt", "w") |
|  | outputFile.write("Token" + "\t\t\t" + "Lexeme\n") |
|  | outputFile.write("-------------------------------\n") |
|  | i = 0 |
|  | while i < len(Token\_Array): |
|  | if Token\_Array[i] != "": |
|  | if Token\_Array[i] == "Keyword" or Token\_Array == "Unknown" or Token\_Array[i] == "Integer" or Token\_Array[i] == "Real": |
|  | outputFile.write(Token\_Array[i] + "\t\t\t" + Lexeme\_Array[i] + "\n") |
|  |  |
|  | else: |
|  | outputFile.write(Token\_Array[i] + "\t\t" + Lexeme\_Array[i] + "\n") |
|  | i += 1 |
|  | outputFile.close() |
|  |  |
|  |  |
|  | #### End of print\_to\_file() #### |
|  |  |
|  | def main(): |
|  | global source\_filename |
|  | #Get file |
|  | filename = input("Enter name of source file to be parsed: ") |
|  | file = open(filename, "r") |
|  | #Remove extension from the source filename |
|  | source\_filename = filename.split(".")[0] |
|  |  |
|  | source\_code\_text = file.readlines() |
|  |  |
|  | for line in source\_code\_text: |
|  | i = 0 |
|  |  |
|  | while i < len(line): |
|  | #Read each token from the source code line and add it into a buffer |
|  | crateBuffer(line[i]) |
|  | i += 1 |
|  |  |
|  | #Sort tokens into lexemes |
|  | lexer(Buffer\_Array) |
|  |  |
|  |  |
|  | #Print table to a file |
|  | print\_to\_file() |
|  |  |
|  | #Close the file |
|  | file.close() |
|  |  |
|  | #### End of main() #### |
|  |  |
|  | if \_\_name\_\_ == "\_\_main\_\_": |
|  | main() |

**Test Cases**

**Sample Input #1**

|  |  |
| --- | --- |
|  | ! Rat18S sample source code 1 (less than 10 lines) ! |
|  | int a; |
|  | float fah$ =9.8; |
|  | upper=90; |
|  | function calculate(a, fah$){ |
|  | while (fah$ <upper) { |
|  | a=23.00; |
|  | return a; |
|  | }} |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

**Sample Output #1**

|  |  |
| --- | --- |
|  | Token Lexeme |
|  | ------------------------------- |
|  | Keyword int |
|  | Identifier a |
|  | Separator ; |
|  | Keyword float |
|  | Identifier fah$ |
|  | Operator = |
|  | Real 9.8 |
|  | Separator ; |
|  | Identifier upper |
|  | Operator = |
|  | Integer 90 |
|  | Separator ; |
|  | Keyword function |
|  | Identifier calculate |
|  | Separator ( |
|  | Identifier a |
|  | Separator , |
|  | Identifier fah$ |
|  | Separator ) |
|  | Separator { |
|  | Keyword while |
|  | Separator ( |
|  | Identifier fah$ |
|  | Operator < |
|  | Identifier upper |
|  | Separator ) |
|  | Separator { |
|  | Identifier a |
|  | Operator = |
|  | Real 23.00 |
|  | Separator ; |
|  | Keyword return |
|  | Identifier a |
|  | Separator ; |
|  | Separator } |
|  | Separator } |

**Sample Input #2**

|  |  |
| --- | --- |
|  | ! Rat18S sample source code 2 (10<lines<20)! |
|  | int x1, y, z; |
|  | ! Commented functions |
|  | read(x1); |
|  | read(y); |
|  | ! |
|  | function compareValues$(){ |
|  | if ( x1 > y & x1 > z ){ ! compare values ! |
|  | write(x1); |
|  | } |
|  | else if ( y > x1 & y > z ) ! compare values ! |
|  | { |
|  | write(y); |
|  | } |
|  | else if ( z => x1 & z > y ){ ! compare values ! |
|  | write(z); |
|  | } |
|  | } |
|  |  |
|  |  |

**Sample Output #2**

|  |  |
| --- | --- |
|  | Token Lexeme |
|  | ------------------------------- |
|  | Keyword int |
|  | Unknown x1 |
|  | Separator , |
|  | Identifier y |
|  | Separator , |
|  | Identifier z |
|  | Separator ; |
|  | Identifier read |
|  | Separator ( |
|  | Unknown x1 |
|  | Separator ) |
|  | Separator ; |
|  | Identifier read |
|  | Separator ( |
|  | Identifier y |
|  | Separator ) |
|  | Separator ; |
|  | Keyword function |
|  | Identifier compareValues$ |
|  | Separator ( |
|  | Separator ) |
|  | Separator { |
|  | Keyword if |
|  | Separator ( |
|  | Unknown x1 |
|  | Operator > |
|  | Identifier y |
|  | Operator & |
|  | Unknown x1 |
|  | Operator > |
|  | Identifier z |
|  | Separator ) |
|  | Separator { |
|  | Identifier write |
|  | Separator ( |
|  | Unknown x1 |
|  | Separator ) |
|  | Separator ; |
|  | Separator } |
|  | Keyword else |
|  | Keyword if |
|  | Separator ( |
|  | Identifier y |
|  | Operator > |
|  | Unknown x1 |
|  | Operator & |
|  | Identifier y |
|  | Operator > |
|  | Identifier z |
|  | Separator ) |
|  | Separator { |
|  | Identifier write |
|  | Separator ( |
|  | Identifier y |
|  | Separator ) |
|  | Separator ; |
|  | Separator } |
|  | Keyword else |
|  | Keyword if |
|  | Separator ( |
|  | Identifier z |
|  | Operator => |
|  | Unknown x1 |
|  | Operator & |
|  | Identifier z |
|  | Operator > |
|  | Identifier y |
|  | Separator ) |
|  | Separator { |
|  | Identifier write |
|  | Separator ( |
|  | Identifier z |
|  | Separator ) |
|  | Separator ; |
|  | Separator } |
|  | Separator } |

**Sample Input #3**

|  |  |
| --- | --- |
|  | ! Rat18S sample source code 3 (more than 20 lines) ! |
|  | int first = 12.3; |
|  | int second =first/2; |
|  | int next; |
|  | int count = 0 |
|  |  |
|  | function rat$(first, second){ |
|  | while (count<10){ |
|  | next %= (second+first) |
|  | } |
|  |  |
|  | return next; |
|  | } |
|  |  |
|  | function write(arg$){ |
|  | print(arg$) |
|  | } |
|  |  |
|  | function main() |
|  | { |
|  | write(first); |
|  | write(second); |
|  |  |
|  | write(rat$(first,second)); |
|  |  |
|  | } |

**Sample Output #3**

|  |  |
| --- | --- |
|  | Token Lexeme |
|  | ------------------------------- |
|  | Keyword int |
|  | Identifier first |
|  | Operator = |
|  | Real 12.3 |
|  | Separator ; |
|  | Keyword int |
|  | Identifier second |
|  | Operator = |
|  | Identifier first |
|  | Operator / |
|  | Integer 2 |
|  | Separator ; |
|  | Keyword int |
|  | Identifier next |
|  | Separator ; |
|  | Keyword int |
|  | Identifier count |
|  | Operator = |
|  | Integer 0 |
|  | Keyword function |
|  | Identifier rat$ |
|  | Separator ( |
|  | Identifier first |
|  | Separator , |
|  | Identifier second |
|  | Separator ) |
|  | Separator { |
|  | Keyword while |
|  | Separator ( |
|  | Identifier count |
|  | Operator < |
|  | Integer 10 |
|  | Separator ) |
|  | Separator { |
|  | Identifier next |
|  | Operator = |
|  | Separator ( |
|  | Identifier second |
|  | Operator + |
|  | Identifier first |
|  | Separator ) |
|  | Separator } |
|  | Keyword return |
|  | Identifier next |
|  | Separator ; |
|  | Separator } |
|  | Keyword function |
|  | Identifier write |
|  | Separator ( |
|  | Identifier arg$ |
|  | Separator ) |
|  | Separator { |
|  | Identifier print |
|  | Separator ( |
|  | Identifier arg$ |
|  | Separator ) |
|  | Separator } |
|  | Keyword function |
|  | Identifier main |
|  | Separator ( |
|  | Separator ) |
|  | Separator { |
|  | Identifier write |
|  | Separator ( |
|  | Identifier first |
|  | Separator ) |
|  | Separator ; |
|  | Identifier write |
|  | Separator ( |
|  | Identifier second |
|  | Separator ) |
|  | Separator ; |
|  | Identifier write |
|  | Separator ( |
|  | Identifier rat$ |
|  | Separator ( |
|  | Identifier first |
|  | Separator , |
|  | Identifier second |
|  | Separator ) |
|  | Separator ) |
|  | Separator ; |
|  | Separator } |