COVER PAGE

CS323 Programming Assignment

Group 10

- **Joshua Ungheanu**
- **Derek Dorr**
- **Adam Weesner**

Assignment Number **[1]**

Due Dates:

Softcopy	10/2 in class by 4:00
Hardcopy	10/2 titanium by 11:55pm

Executable FileName [*LexerAnalyzer.exe*]
(A file that can be executed without compilation by the instructor)

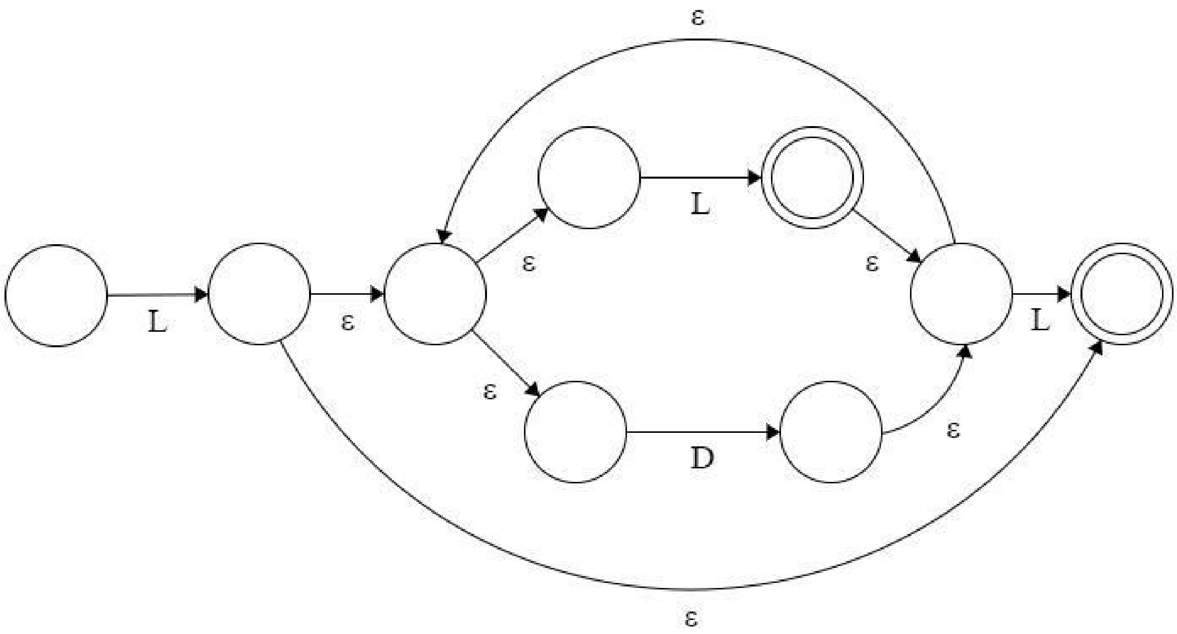
Operating System [*Windows 10*]

GRADE:

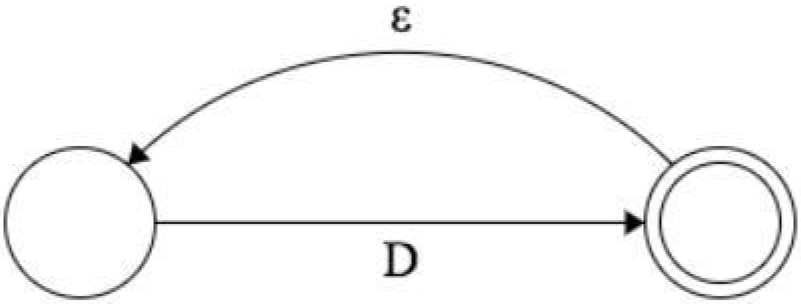
COMMENTS:

FSMs

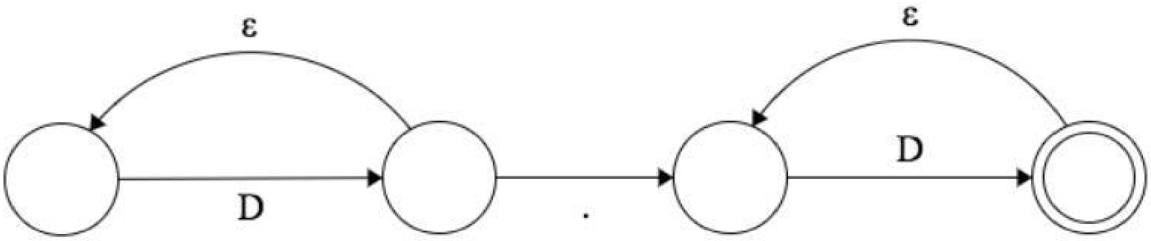
Identifier: L (L | D)+ L



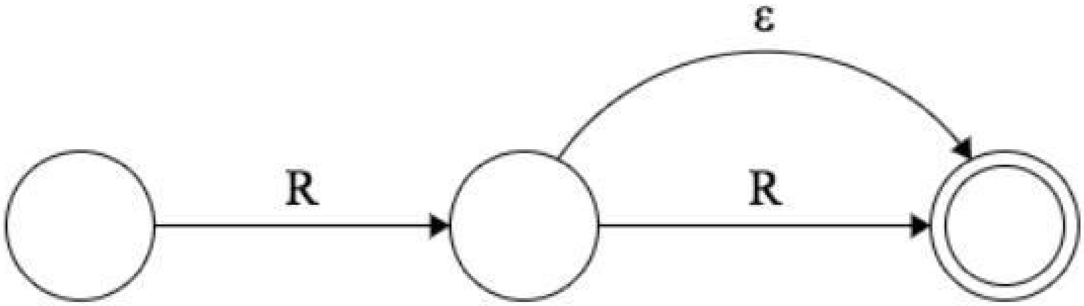
Integer: D^+



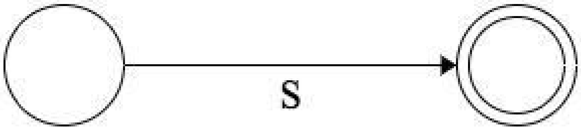
Real: $D^+ .D^+$

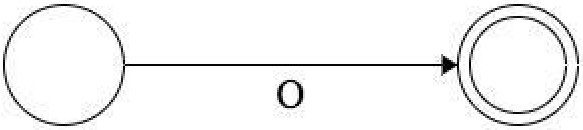


Relational operators -> Reloop: (R | RR)



Separator: S
Operator: O





1. Problem Statement

This first assignment is to write a lexical analyzer (lexer). To build the lexer, we would need to at least build FSMs for an identifier, integer and real. Here we built FSMs for those first 3 and relational operators, operations, and separators. The lexer should be able to read in a token and return a token when it is needed. The lexer should output a record for the token and a record for the actual "value" into an output file. In other words, our program should read in a file containing source code of Rat18S to generate tokens and then write out the results to an output file.

2. How to use your program

First method: You should be able to click the "LexerAnalyzer.exe" executable file and the program will start running. Enter the name of the source file with the ".txt" extension, and the executable should run.

Second Method: *Can be done using a terminal from either; a Mac OSX, Linux, or titan server through Putty. Note that for method 2, In order to use the program, you should have your terminal setup to run an executable file. Look for the directory that contains the files to be tested (NOTE: using the terminal requires more steps). Once you have accessed to your directory, type the following command: cd [filepath] and hit enter, then type start [filename.exe]which contains our lexical analyzer code. Our program will take an input of a .txt path, which will be used to analyze. Note that the .txt must be from the directory which contains our 3 test cases. In order to test more test cases, it is recommended to add those extra ".txt" files into your directory. Once the .txt path have been selected, hit enter and our program will then read the file, generate the tokens, and finally write the results to an output file called "outFile.txt" Note that the outFile.txt will be overwritten every time you use a different test case to analyze. The executable file should be working fine on windows OS only and was provided to satisfy the requirements of the assignment.*

3. Design of your program

Our program was designed with the purpose of converting a sequence of characters from one of the test cases provided in our directory into a sequence of tokens. Our lexical analyzer breaks the syntaxes into a series of tokens and if an invalid token is found, an error will be displayed. Otherwise, legal tokens will be displayed on the terminal and saved to the outFile.txt. All valid letters, digits, symbols, and acceptance states are stored in a list while, keywords, relational operators, separators and operators are stored in a set. Our finite state machine is stored in a 2-Dimensional array where each value stores a different state based on the token's type. And the token types are stored in a dictionary where the key is the acceptance state and the value are the token type (e.g. identifier, integer, real, etc.).

4. Any Limitation

Our program was limited to less than 60 lines of source code. Any source code with more than 60 lines of code was not tested in this program.

5. Any shortcomings

The hardest part was to identify/classify all the tokens with the fsm, so we ended up only using the fsm to only identify spaces, separators, operators, and real numbers everything else was considered an identifier, then it finds strings later on.

SOURCE CODE

```
// Simple Lexer Assignment
// Derek Dorr, Jonathan Ungheanu, Adam Weesner
// CPSC 323 Fall 2018
// Shohrat Geldiyev
// 10/2/18

/***************************************************************************
To use this program, you just need to put a text file in the project directory
and when the program starts it will ask you to input the file name.

the rest is done automatically!
***************************************************************************/

#include<iostream>
#include<fstream>
#include<string>
#include<vector>
using namespace std;

enum FSM_TRANSITIONS {
	REJECT = 0,
	STRING,
	SEPARATOR,
	SPACE,
	OPERATOR,
	REAL
};

struct TokenType {
	string token;
	int lexeme;
	string lexemeName;
};

//Function Prototypes
int Get_FSM_Col(char currentChar);
string GetLexemeName(int lexeme);
vector<TokenType> Lexer(string text);
bool isKeyword(string token);
```

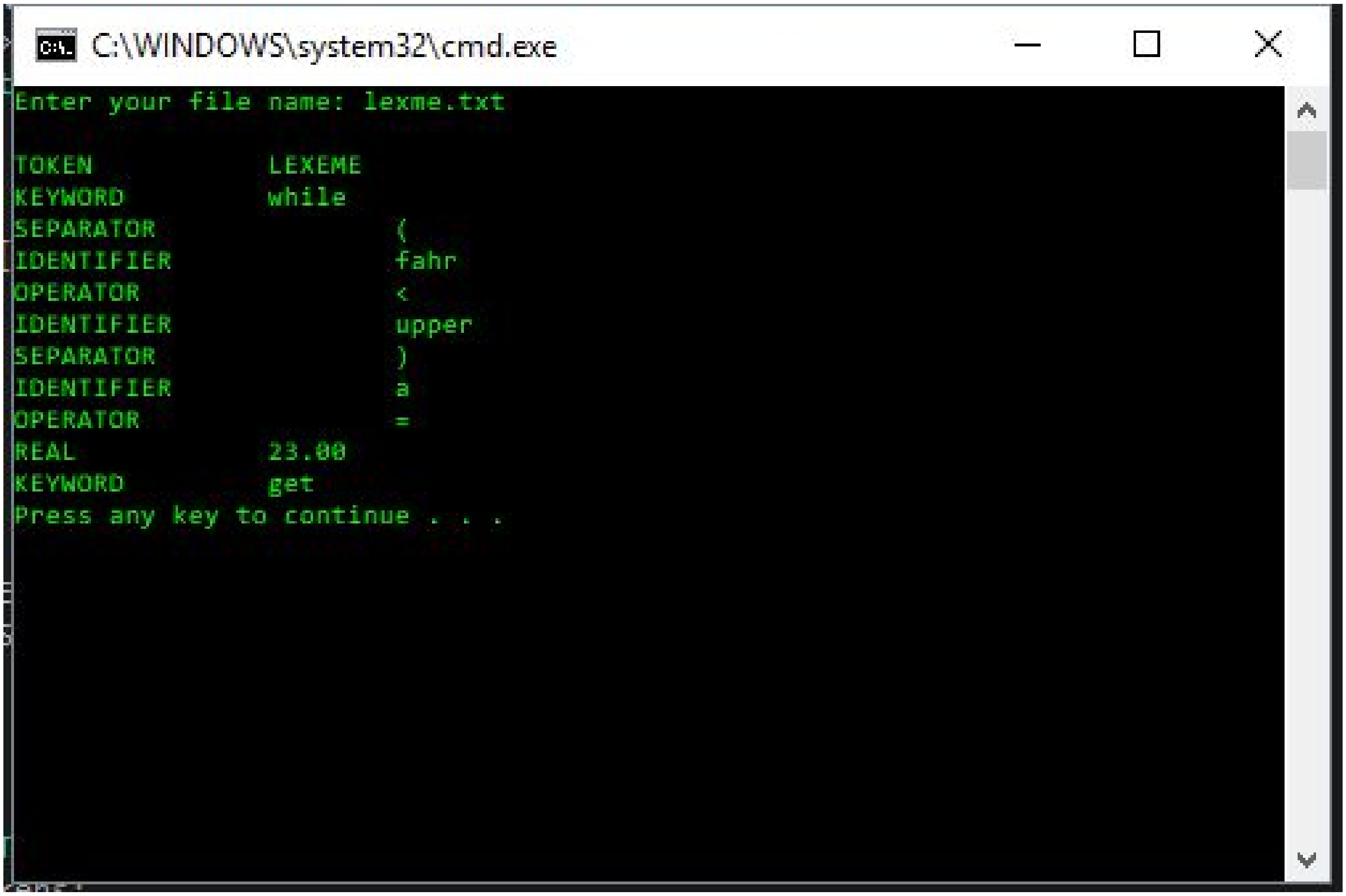
```
//                                              string,   separator,  space,   operator,
real
int stateTable[][6] = { {0,    STRING,    SEPARATOR,  SPACE,   OPERATOR,  REAL},
                            {STRING,    STRING,    REJECT,    REJECT,  REJECT,    REJECT},
                            {SEPARATOR, REJECT,    REJECT,    REJECT,  REJECT,    REJECT},
                            {SPACE,     REJECT,    REJECT,    REJECT,  REJECT,    REJECT},
                            {OPERATOR,  REJECT,    REJECT,    REJECT,  REJECT,    REJECT},
                            {REAL,      REJECT,    REJECT,    REJECT,  REJECT,    REAL } };

//DICTIONARY
const int DICSIZE = 10;
string keywd[DICSIZE] = { "while", "get", "int", "put", "if", "else", "endif", "return", "print",
"end" };

int quoteCount = 0;

int main() {
	vector<TokenType>tokenVec;
	TokenType tokens;
	fstream inFile;
	string text = "";
	string fileName;

	cout << "Enter your file name: ";
	cin >> fileName;
	cout << endl;

	inFile.open(fileName);
	if (inFile.fail()) {
		cout << "\nUNABLE TO OPEN FILE " << fileName << endl;
		exit(1);
	}
	cout << "TOKEN\t\tLEXEME\n";
	while (getline(inFile, text)) {
		tokenVec = Lexer(text);
		//Cheats the FSM and re-assigns 'strings' to pre defined lexemes
		//maybe if i made the vector global i can do this in get_fsm_col...
		for (unsigned j = 0; j < tokenVec.size(); j++) {

			if (isKeyword(tokenVec[j].token) == true) {
				tokenVec[j].lexemeName = "KEYWORD";
			}//send to find keyword, if not keyword then its an identifier, if not
then its a string

			if (tokenVec[j].token[0] == 34) {
```

```
                quoteCount++;
            }
            if (quoteCount == 1 || quoteCount == 2) {
                tokenVec[j].lexemeName = "STRING";
                if (quoteCount == 2) {
                    quoteCount = 0;
                }
            }


            cout << tokenVec[j].lexemeName << "\t\t" << tokenVec[j].token <<
endl;
        }
    }
    inFile.close();

    /*Every character read makes the program check to see
    what type of char it is and changes states accordingly*/

    return 0;
}

//The Lexer, parses thru the text file and finds tokens and updates the state of the FSM
vector<TokenType> Lexer(string text) {
    TokenType access;
    vector<TokenType> myTokens;
    string currentToken = "";
    int currentState = REJECT;
    int prevState = REJECT;
    int col = REJECT;
    char currentChar = ' ';

    for (unsigned i = 0; i < text.size();) {
        currentChar = text[i];

        col = Get_FSM_Col(currentChar);

        currentState = stateTable[currentState][col];

        if (currentState == REJECT) {
            if (prevState != SPACE) {
                access.token = currentToken;
                access.lexeme = prevState;
                access.lexemeName = GetLexemeName(access.lexeme);
                myTokens.push_back(access);
            }
            currentToken = "";
        }
        else {
            currentToken += currentChar;
            ++i;
        }
        prevState = currentState;
    }
    if (currentState != SPACE && currentToken != "") {
        access.token = currentToken;
        access.lexeme = currentState;
        access.lexemeName = GetLexemeName(access.lexeme);
        myTokens.push_back(access);
    }

    return myTokens;
}

//recieves each individual character, decides what type it is and returns
//a value that switches the machine's state accordingly
int Get_FSM_Col(char currentChar) {
    int value = currentChar;

    if (isspace(currentChar)) {
        return SPACE;
    }
    else if (isalpha(currentChar) || value == 36) {
        return STRING;
    }
    else if (value == 34 || value == 40 || value == 41 || value == 125 || value == 123 ||
value == 44 || value == 37 || value == 59 || value == 58 || value == 91 || value == 93 ) {
        return SEPARATOR;
    }
    else if (value == 60 || value == 61 || value == 62 || value == 42 || value == 43 || value
== 45 || value == 47) {
        return OPERATOR;
    }
    else if (isdigit(currentChar)) {
        return REAL;
    }
    else {//may need to remove else statement...
        return REJECT;
    }
}
```

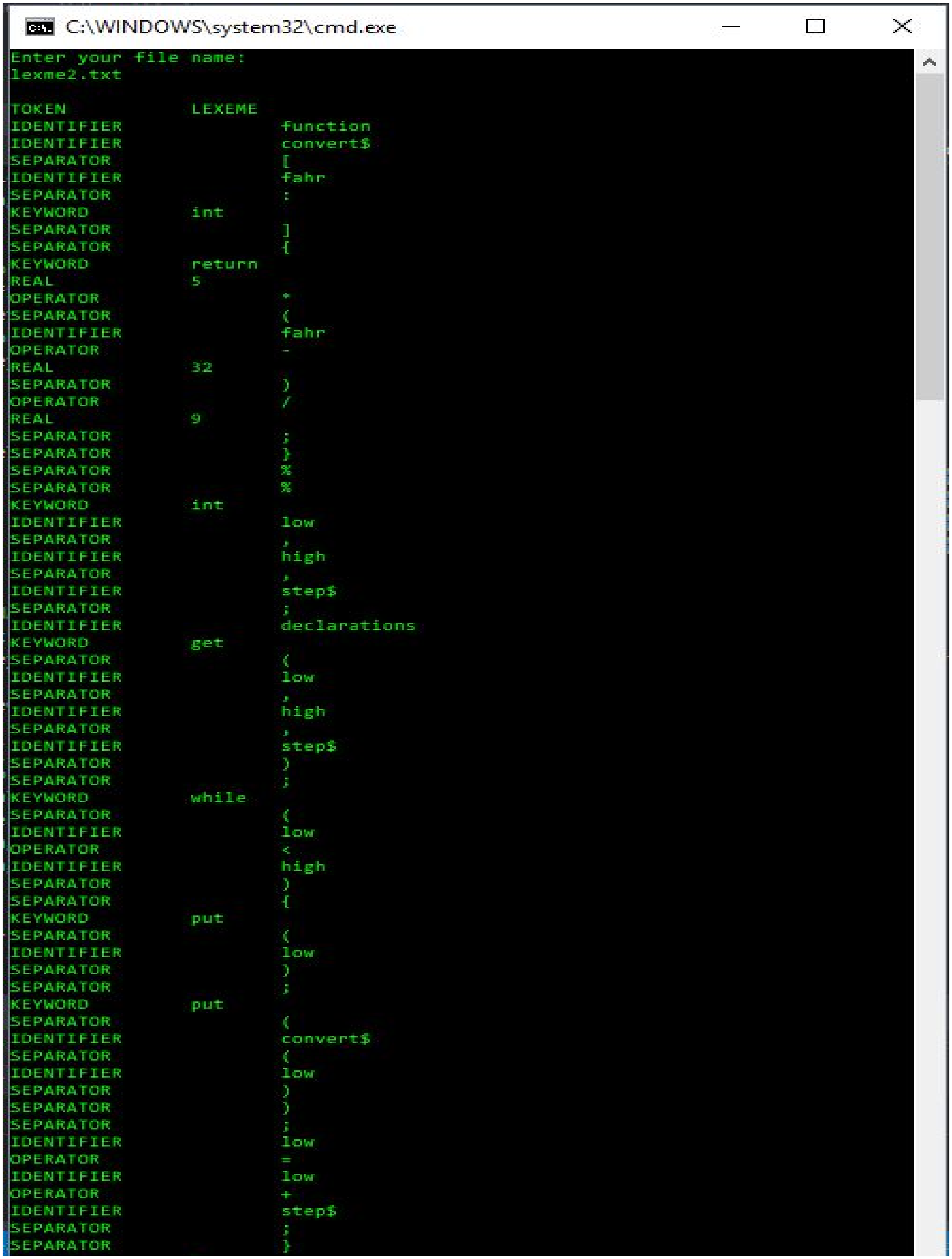
```
//Converts the enumerated lexeme into a string then returns the string
string GetLexemeName(int lexeme) {
	switch (lexeme) {
	case STRING://changing this to identifier , will double for loop the vector and chars to
find quotes for strings...
		return "IDENTIFIER";
		break;
	case SEPARATOR:
		return "SEPARATOR";
		break;
	case SPACE:
		return "SPACE";
		break;
	case OPERATOR:
		return "OPERATOR";
		break;
	case REAL:
		return "REAL";
		break;
	default:
		return "ERROR";
		break;
	}
}

bool isKeyword(string token) {
	for (int i = 0; i < DICSIZE; i++) {
		if (token == keywd[i]) {
			return true;
		}
	}
	return false;
}
```

TEST CASES

Case - 1



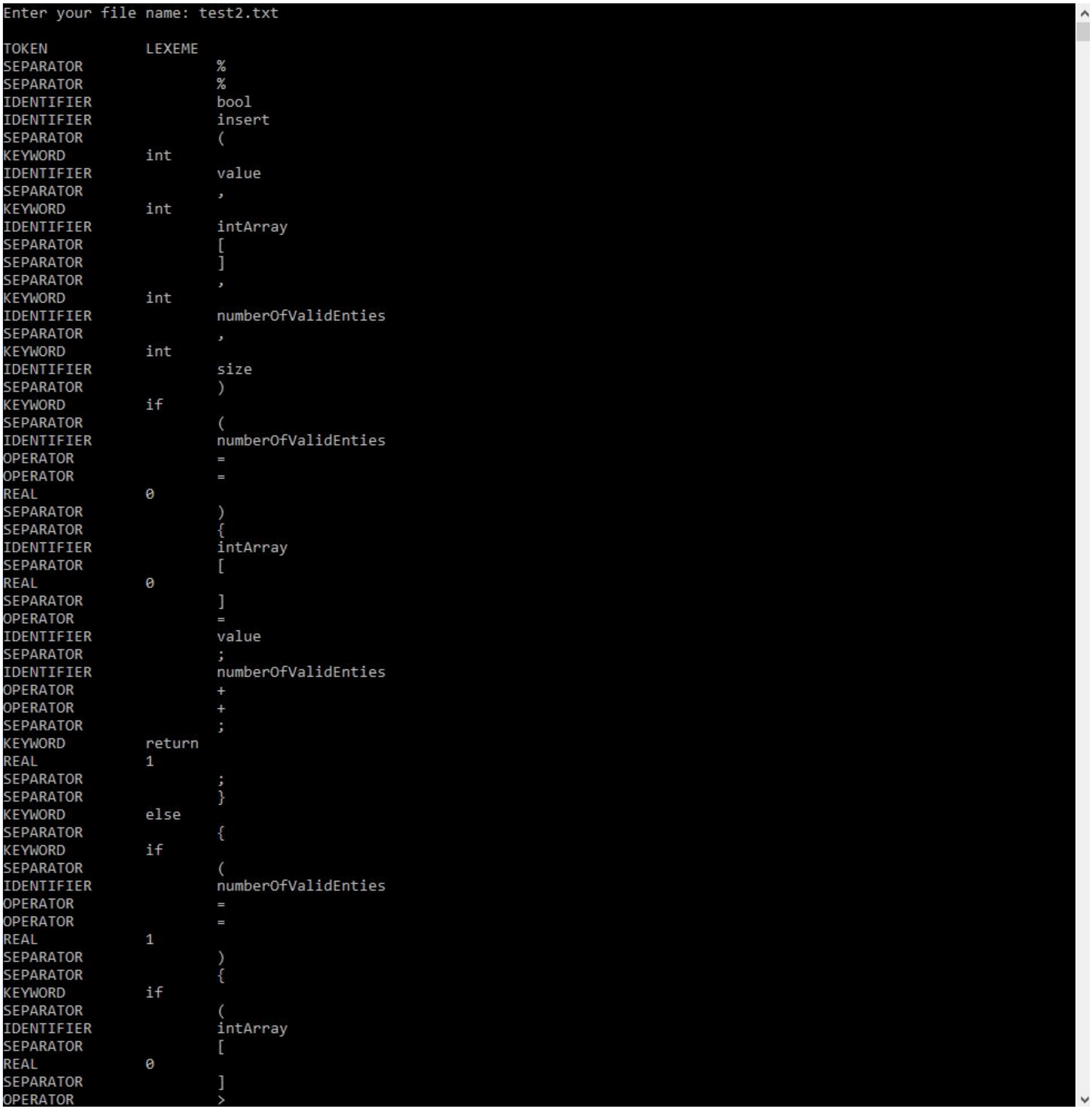
```
Enter your file name: lexme.txt

TOKEN               LEXEME
KEYWORD             while
SEPARATOR                     (
IDENTIFIER                    fahr
OPERATOR                      <
IDENTIFIER                    upper
SEPARATOR                     )
IDENTIFIER                    a
OPERATOR                      =
REAL                23.00
KEYWORD             get
Press any key to continue . . .
```



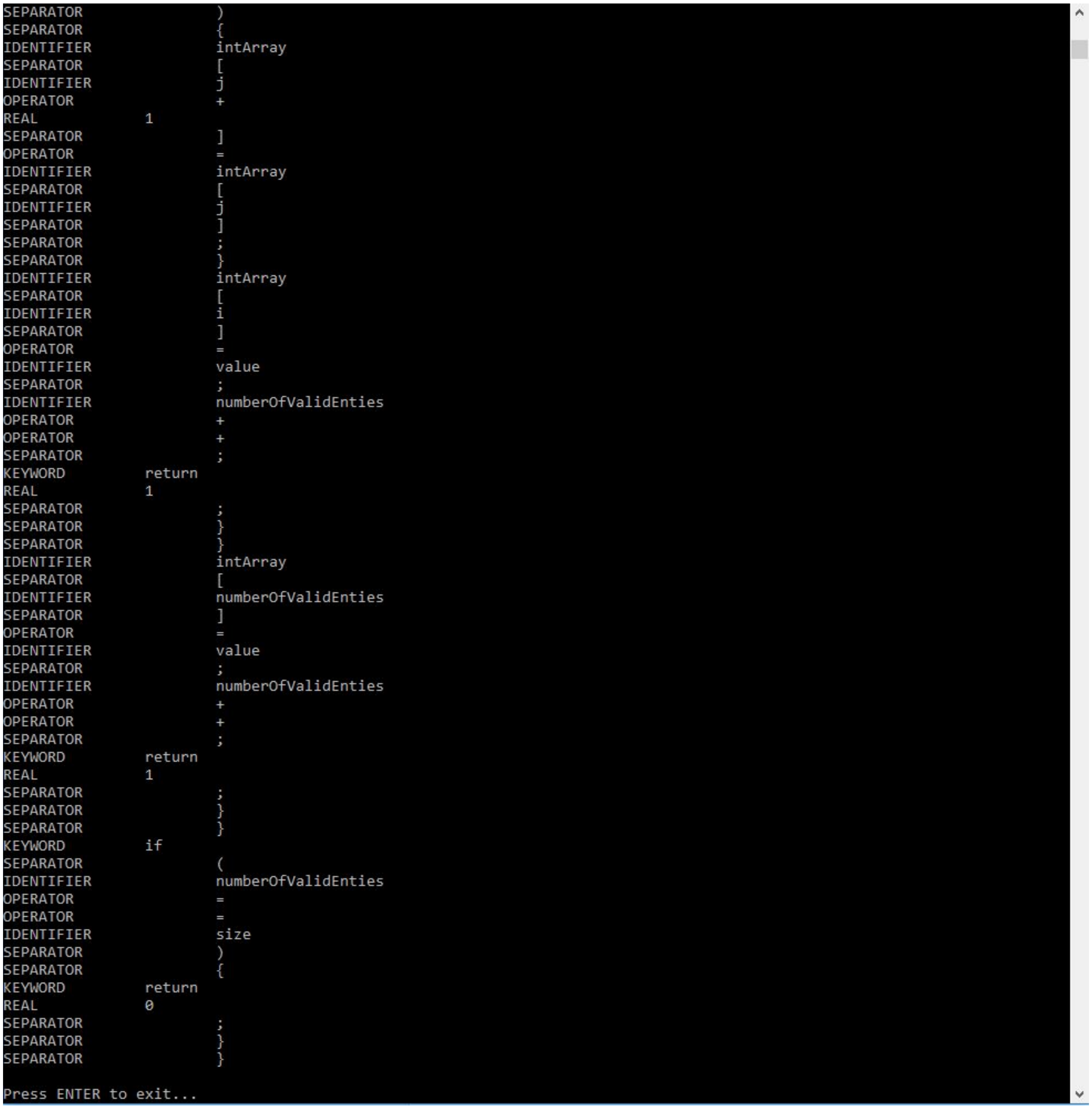
```
Enter your file name:
lexme2.txt

TOKEN           LEXEME
IDENTIFIER              function
IDENTIFIER              convert$
SEPARATOR               [
IDENTIFIER              fahr
SEPARATOR               :
KEYWORD         int
SEPARATOR               ]
SEPARATOR               {
KEYWORD         return
REAL            5
OPERATOR                *
SEPARATOR               (
IDENTIFIER              fahr
OPERATOR                -
REAL            32
SEPARATOR               )
OPERATOR                /
REAL            9
SEPARATOR               ;
SEPARATOR               }
SEPARATOR               %
SEPARATOR               %
KEYWORD         int
IDENTIFIER              low
SEPARATOR               ,
IDENTIFIER              high
SEPARATOR               ,
IDENTIFIER              step$
SEPARATOR               ;
IDENTIFIER              declarations
KEYWORD         get
SEPARATOR               (
IDENTIFIER              low
SEPARATOR               ,
IDENTIFIER              high
SEPARATOR               ,
IDENTIFIER              step$
SEPARATOR               )
SEPARATOR               ;
KEYWORD         while
SEPARATOR               (
IDENTIFIER              low
OPERATOR                <
IDENTIFIER              high
SEPARATOR               )
SEPARATOR               {
KEYWORD         put
SEPARATOR               (
IDENTIFIER              low
SEPARATOR               )
SEPARATOR               ;
KEYWORD         put
SEPARATOR               (
IDENTIFIER              convert$
SEPARATOR               (
IDENTIFIER              low
SEPARATOR               )
SEPARATOR               )
SEPARATOR               ;
IDENTIFIER              low
OPERATOR                =
IDENTIFIER              low
OPERATOR                +
IDENTIFIER              step$
SEPARATOR               ;
SEPARATOR               }
```

Case - 2

```
%%
bool insert(int value, int intArray[], int & numberOfValidEnties, int size)
 if (numberOfValidEnties == 0)
 {
  intArray[0] = value;
  numberOfValidEnties++;
  return 1;
 }
 else
 {
  if (numberOfValidEnties == 1)
  {
    if (intArray[0] > value)
    {
     int temp = intArray[0];
     intArray[0] = value;
     intArray[1] = temp;
     numberOfValidEnties++;
     return 1;
    }
    else
    {
     intArray[1] = value;
     numberOfValidEnties++;
     return 1;
    }
  }
  else
  {
   for (int i = 0; i < numberOfValidEnties; i++)
   {
    if (value <= intArray[i])
    {
     for (int j = numberOfValidEnties - 1; j >= i; j--)
     {
      intArray[j + 1] = intArray[j];
     }
     intArray[i] = value;
     numberOfValidEnties++;
     return 1;
    }
   }
   intArray[numberOfValidEnties] = value;
   numberOfValidEnties++;
   return 1;
  }
 }
 if (numberOfValidEnties == size)
 {
  return 0;
 }}
```



```
Enter your file name: test2.txt

TOKEN           LEXEME
SEPARATOR               %
SEPARATOR               %
IDENTIFIER              bool
IDENTIFIER              insert
SEPARATOR               (
KEYWORD         int
IDENTIFIER              value
SEPARATOR               ,
KEYWORD         int
IDENTIFIER              intArray
SEPARATOR               [
SEPARATOR               ]
SEPARATOR               ,
KEYWORD         int
IDENTIFIER              numberOfValidEnties
SEPARATOR               ,
KEYWORD         int
IDENTIFIER              size
SEPARATOR               )
KEYWORD         if
SEPARATOR               (
IDENTIFIER              numberOfValidEnties
OPERATOR                =
OPERATOR                =
REAL            0
SEPARATOR               )
SEPARATOR               {
IDENTIFIER              intArray
SEPARATOR               [
REAL            0
SEPARATOR               ]
OPERATOR                =
IDENTIFIER              value
SEPARATOR               ;
IDENTIFIER              numberOfValidEnties
OPERATOR                +
OPERATOR                +
SEPARATOR               ;
KEYWORD         return
REAL            1
SEPARATOR               ;
SEPARATOR               }
KEYWORD         else
SEPARATOR               {
KEYWORD         if
SEPARATOR               (
IDENTIFIER              numberOfValidEnties
OPERATOR                =
OPERATOR                =
REAL            1
SEPARATOR               )
SEPARATOR               {
KEYWORD         if
SEPARATOR               (
IDENTIFIER              intArray
SEPARATOR               [
REAL            0
SEPARATOR               ]
OPERATOR                >
```

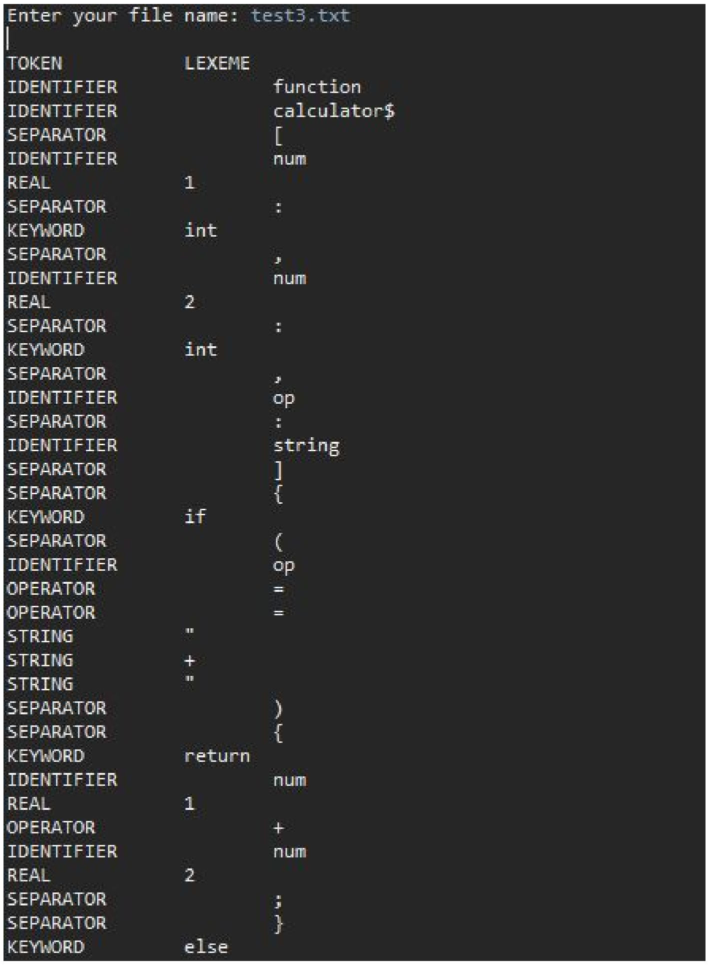


```
SEPARATOR               )
SEPARATOR               {
IDENTIFIER              intArray
SEPARATOR               [
IDENTIFIER              j
OPERATOR                +
REAL            1
SEPARATOR               ]
OPERATOR                =
IDENTIFIER              intArray
SEPARATOR               [
IDENTIFIER              j
SEPARATOR               ]
SEPARATOR               ;
SEPARATOR               }
IDENTIFIER              intArray
SEPARATOR               [
IDENTIFIER              i
SEPARATOR               ]
OPERATOR                =
IDENTIFIER              value
SEPARATOR               ;
IDENTIFIER              numberOfValidEnties
OPERATOR                +
OPERATOR                +
SEPARATOR               ;
KEYWORD         return
REAL            1
SEPARATOR               ;
SEPARATOR               }
SEPARATOR               }
IDENTIFIER              intArray
SEPARATOR               [
IDENTIFIER              numberOfValidEnties
SEPARATOR               ]
OPERATOR                =
IDENTIFIER              value
SEPARATOR               ;
IDENTIFIER              numberOfValidEnties
OPERATOR                +
OPERATOR                +
SEPARATOR               ;
KEYWORD         return
REAL            1
SEPARATOR               ;
SEPARATOR               }
SEPARATOR               }
KEYWORD         if
SEPARATOR               (
IDENTIFIER              numberOfValidEnties
OPERATOR                =
OPERATOR                =
IDENTIFIER              size
SEPARATOR               )
SEPARATOR               {
KEYWORD         return
REAL            0
SEPARATOR               ;
SEPARATOR               }
SEPARATOR               }

Press ENTER to exit...
```

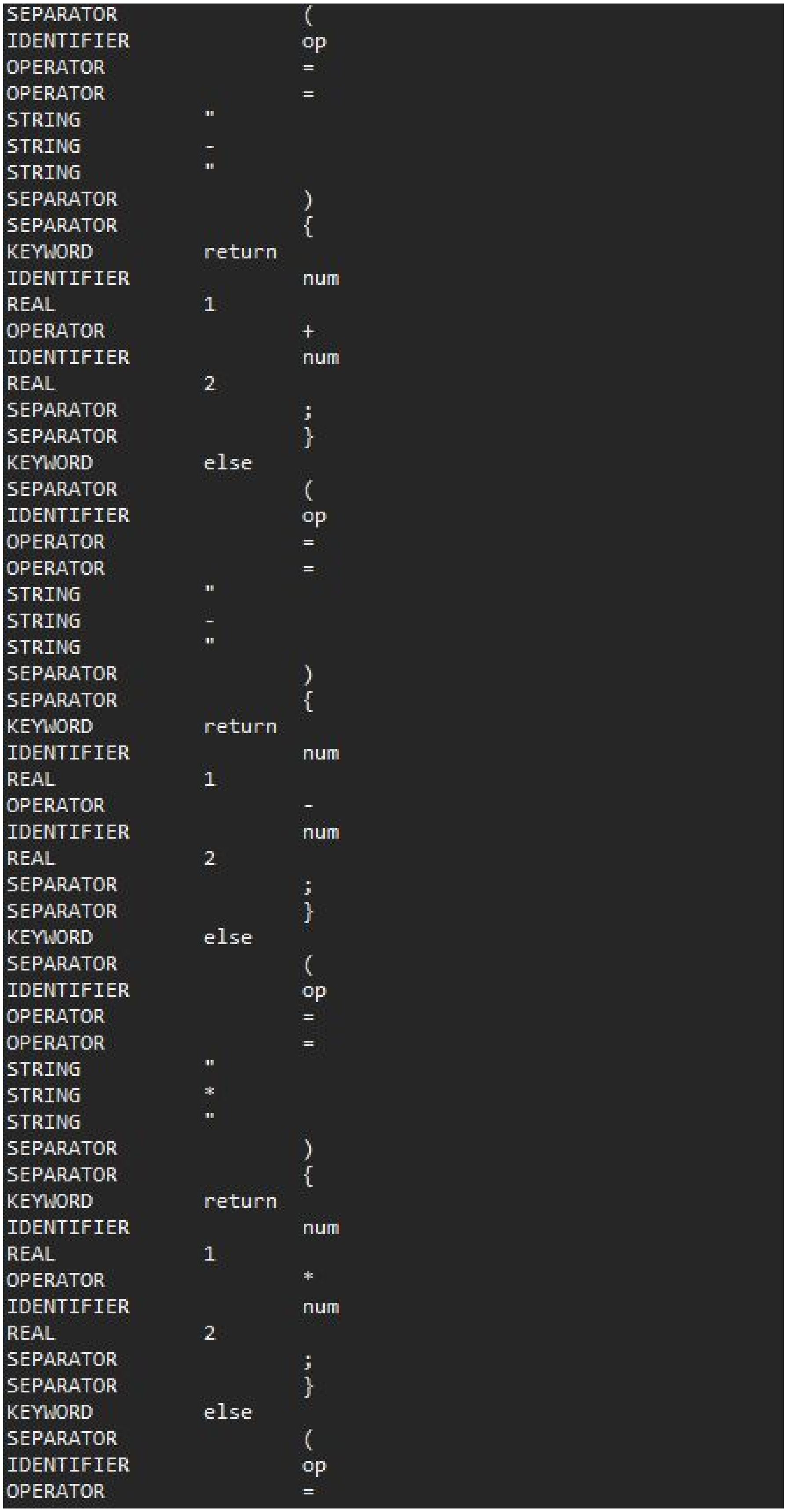
Case - 3

```
function calculator$ [num1:int, num2:int, op:string]
{
   if (op == "+") {
              return num1 + num2; }
   else (op == "-") {
              return num1 - num2; }
   else (op == "*") {
              return num1 * num2; }
```

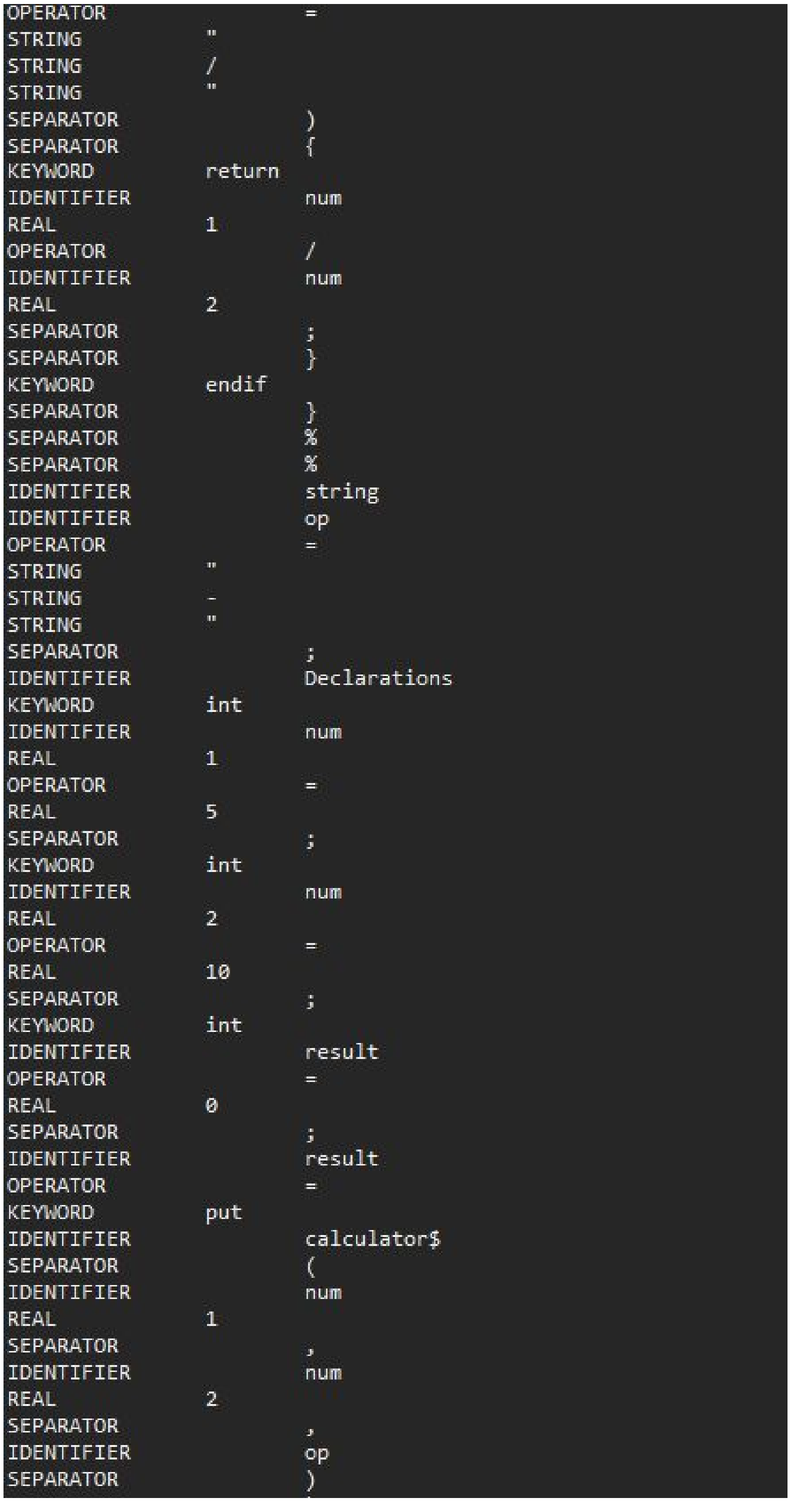
```
  else (op == "/") {
         return num1 / num2; }
  endif
}

%%
string op = "-"; ! Declarations !
int num1 = 5;
int num2 = 10;
int result = 0;

result = put calculator$ (num1, num2, op));
put (result);
```



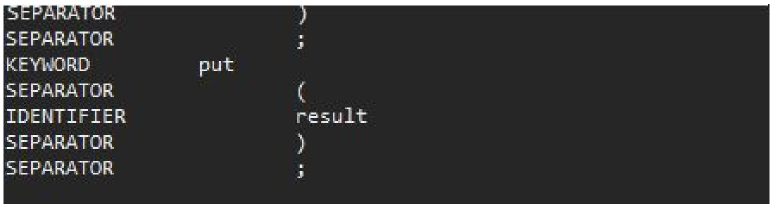
```
Enter your file name: test3.txt

TOKEN           LEXEME
IDENTIFIER              function
IDENTIFIER              calculator$
SEPARATOR               [
IDENTIFIER              num
REAL            1
SEPARATOR               :
KEYWORD         int
SEPARATOR               ,
IDENTIFIER              num
REAL            2
SEPARATOR               :
KEYWORD         int
SEPARATOR               ,
IDENTIFIER              op
SEPARATOR               :
IDENTIFIER              string
SEPARATOR               ]
SEPARATOR               {
KEYWORD         if
SEPARATOR               (
IDENTIFIER              op
OPERATOR                =
OPERATOR                =
STRING          "
STRING          +
STRING          "
SEPARATOR               )
SEPARATOR               {
KEYWORD         return
IDENTIFIER              num
REAL            1
OPERATOR                +
IDENTIFIER              num
REAL            2
SEPARATOR               ;
SEPARATOR               }
KEYWORD         else
```



```
SEPARATOR               (
IDENTIFIER              op
OPERATOR                =
OPERATOR                =
STRING          "
STRING          -
STRING          "
SEPARATOR               )
SEPARATOR               {
KEYWORD         return
IDENTIFIER              num
REAL            1
OPERATOR                +
IDENTIFIER              num
REAL            2
SEPARATOR               ;
SEPARATOR               }
KEYWORD         else
SEPARATOR               (
IDENTIFIER              op
OPERATOR                =
OPERATOR                =
STRING          "
STRING          -
STRING          "
SEPARATOR               )
SEPARATOR               {
KEYWORD         return
IDENTIFIER              num
REAL            1
OPERATOR                -
IDENTIFIER              num
REAL            2
SEPARATOR               ;
SEPARATOR               }
KEYWORD         else
SEPARATOR               (
IDENTIFIER              op
OPERATOR                =
OPERATOR                =
STRING          "
STRING          *
STRING          "
SEPARATOR               )
SEPARATOR               {
KEYWORD         return
IDENTIFIER              num
REAL            1
OPERATOR                *
IDENTIFIER              num
REAL            2
SEPARATOR               ;
SEPARATOR               }
KEYWORD         else
SEPARATOR               (
IDENTIFIER              op
OPERATOR                =
```



```
OPERATOR                =
STRING          "
STRING          /
STRING          "
SEPARATOR               )
SEPARATOR               {
KEYWORD         return
IDENTIFIER              num
REAL            1
OPERATOR                /
IDENTIFIER              num
REAL            2
SEPARATOR               ;
SEPARATOR               }
KEYWORD         endif
SEPARATOR               }
SEPARATOR               %
SEPARATOR               %
IDENTIFIER              string
IDENTIFIER              op
OPERATOR                =
STRING          "
STRING          -
STRING          "
SEPARATOR               ;
IDENTIFIER              Declarations
KEYWORD         int
IDENTIFIER              num
REAL            1
OPERATOR                =
REAL            5
SEPARATOR               ;
KEYWORD         int
IDENTIFIER              num
REAL            2
OPERATOR                =
REAL            10
SEPARATOR               ;
KEYWORD         int
IDENTIFIER              result
OPERATOR                =
REAL            0
SEPARATOR               ;
IDENTIFIER              result
OPERATOR                =
KEYWORD         put
IDENTIFIER              calculator$
SEPARATOR               (
IDENTIFIER              num
REAL            1
SEPARATOR               ,
IDENTIFIER              num
REAL            2
SEPARATOR               ,
IDENTIFIER              op
SEPARATOR               )
```



```
SEPARATOR               )
SEPARATOR               ;
KEYWORD         put
SEPARATOR               (
IDENTIFIER              result
SEPARATOR               )
SEPARATOR               ;
```