

COVER PAGE

CPSC 323

Royce Nguyen

Assignment 1

Due 10/2/2017

Turned in 10/2/17

“Lexical Analyzer.exe”

CS 110B

Windows OS

GRADE:

COMMENTS:

Assignment 1 Documentation

1. Problem Statement

To write a lexical analyzer for the RAT17F language.

2. How to use program

To use the program, open the directory containing all the program files. In order to have the lexer generate an output of tokens and lexemes, open the file “RAT17F.txt” and input your code in the RAT17F language. The default input of the program is the sample input provided by the assignment instructions. Then, save the file and close it. Next, open the file “Lexical Analyzer.exe.” An output list consisting of all the tokens and their values should be listed. If a word or character does not match the lexical rules of the language, it will not be listed. To repeat, simply change the contents of the text file and run the executable file to get the lexical analysis of the text.

3. Design of your program

My program consisted of several void and Boolean expressions in order to create the lexical analyzer. I used a parse function to parse through the given text file. Additionally, I used Boolean functions to analyze the character returned from the parse to see if it fit the criteria to be either a separator, operator, keyword, identifier, integer, or float. In order to differentiate between a keyword and identifier, I used a function to check if the character parsed is alphanumeric which I then parsed the entire word into an array. Using that array, I compared it to the array of keywords and if it matched with any, it would be a keyword, otherwise it would be an identifier. The main concept of my program code was to parse each character one-by-one and analyze it to fit a token.

4. Limitations

None

5. Shortcomings

My program code had several shortcomings in relation to the project requirements. I was unable to effectively implement a way for the

lexer to output the difference between an integer and a real. As the lexer printed the token, I was able to get the entire real number to print but the token would still be labeled as integer. Therefore, to make up for the shortcoming, I simply re-labeled the token as “Number.”

Another shortcoming in the program code was the fact that I was unable to effectively find a way to print two-character relational operators such as the \leq , \geq , and $==$ signs. Instead, I wrote specific if-statements to check if the character parsed was $<$, $>$, or $=$. Then I would parse to the next character and if it was a $=$, then I would print that as well, otherwise I would just end it. Besides those, my code was able to effectively print out the tokens and lexemes of a file written in RAT17F.

Additionally, I realized a flaw in my code in which the array I kept all the RAT17F keywords did not have all the words. I had forgotten to input the keywords “return” and “else” when inputting the keywords into the array. Therefore, the code I submitted online had that flaw. The source code I provided in this documentation has the updated code.

Source Code

```
//Name: Royce Nguyen
//E-mail: roycenguyen@csu.fullerton.edu
//Course: CPSC 323
//Assignment: Assignment 1

//This program is a lexical analyzer that parses a file written in the programming
language RAT17F and outputs its tokens and token types
//Language MUST be inputted into the file "RAT17F.txt" in order for program to
parse the file.

#include <iostream>
#include <fstream>
#include <string>

using namespace std;

char c, word[15];
ifstream myFile("RAT17F.txt");

char parseChar()    // Parses the file one character every time it is called
{
    c = myFile.get();
    return c;
} // End of parseChar

bool isOperator()    // Checks if character qualifies as an operator
{
    char operators[] = { '<=', '>=', '+', '-', '=', '/', '*', '<', '>',
'%', ':' };
    for (int i = 0; i <= 12; i++)
    {
        if (c == operators[i])
        {
            return true;
        }
    }
    return false;
} // End of isOperator

bool isSeparator() // Checks if character qualifies as a separator
{
    char separators[] = { '{', '}', '(', ')', ',', ';' };
    for (int i = 0; i <= 5; i++)
    {
        if (c == separators[i])
        {
            return true;
        }
    }
    return false;
} // End of isSeparator

bool isKeyword(char word[])    // Checks if character is in a keyword
{

```

```

        char keywords[8][10] = { "int", "if", "fi", "while", "read", "write",
        "else", "return" };
        for (int i = 0; i <= 32; i++)
        {
            if (strcmp(keywords[i], word) == 0)
            {
                return true;
                break;
            }
        }

        return false;
    } // End of isKeyword

    bool isNum(char c) // Checks if character is part of a number
    {
        if (((c - '0') <= 9) && ((c - '0') >= 0))
        {
            return true;
        }
        return false;
    } // End of isNum

    void lexer() //Returns tokens and their values (lexeme)
    {
        int counter = 0;
        if (!myFile.is_open())
        {
            cout << "File 'RAT17F.txt' could not be opened." << endl;
        }

        cout << "Token\t\tLexeme" << endl;
        cout << "-----" << endl;

        while (!myFile.eof())
        {
            parseChar();
            if (isOperator()) // Operators
            {
                cout << "Operator\t";
                if (c == '<' || '>' || '=' || ':') // Checks two-characters
                {
                    cout << c;
                    c = parseChar();
                    if (c == '=')
                    {
                        cout << c;
                        c = parseChar();
                    }
                }
                cout << endl;
            }
            else if (isSeparator()) // Separators
            {
                cout << "Separator\t" << c << endl;
                if (c == '%')
                {

```

```

        cout << c << c << endl;
        parseChar();
        parseChar();
    }
    else if (c == '@')
    {
        cout << "Separator\t" << c << endl;
        parseChar();
    }
}
else if (isNum(c)) // Numbers
{
    cout << "Number\t\t";
    while (((c - '0') <= 9) && ((c - '0') >= 0))
    {
        cout << c - '0';
        c = parseChar();
        if (c == '.')
        {
            cout << c;
            c = parseChar();
        }
    }
    cout << endl;
}
else if (isalnum(c) || c == '#') // Checks string
{
    word[counter++] = c; // Moves character into string array
}
else if ((c == ' ' || c == '\n') && (counter != 0)) // Words
{
    word[counter] = '\0';
    counter = 0;

    if (isKeyword(word))
    {
        cout << "Keyword\t\t" << word << endl;
    }
    else
    {
        cout << "Identifier\t" << word << endl;
    }
}
}
} // End of lexer

int main()
{
    lexer();

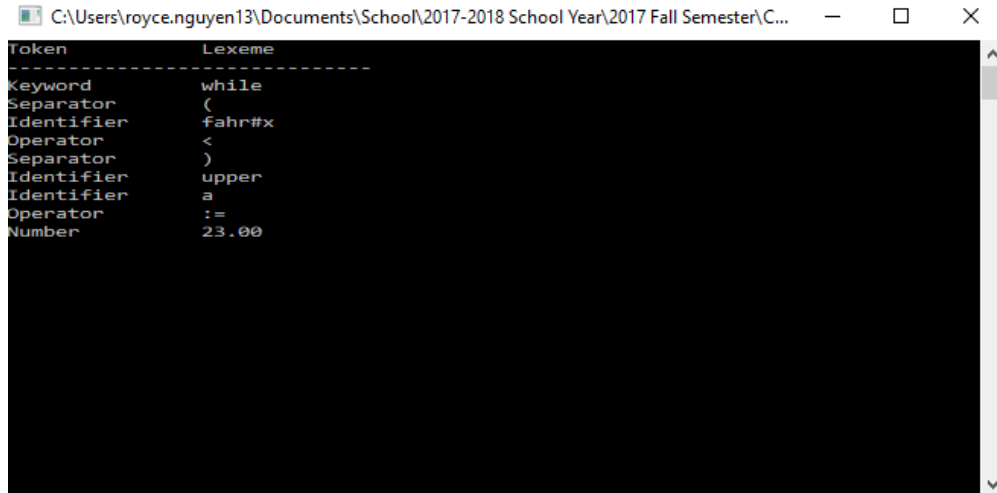
    getchar();
    system("pause");
    return 0;
} // End of main

```

Test Cases

1. Test case: < 10

while (fahr#x < upper) a := 23.00



The screenshot shows a text editor window with the following content:

Token	Lexeme
Keyword	while
Separator	(
Identifier	fahr#x
Operator	<
Separator)
Identifier	upper
Identifier	a
Operator	:=
Number	23.00

2. Test case: < 20

```
convert (fahr : integer)
{
    return 5*(fahr-32)/9;
}

%%

integer low, high, step#xy;

read (low, high, step#xy);
while (low < high)
{
    write (low);
    write (convert[low]);
    low := low + step#xy;
}
```

```

Keyword      while
Separator    (
Identifier    low
Operator      <
Separator    )
Identifier    high
Separator    {
Keyword      write
Separator    (
Separator    )
Separator    ;
Identifier    low
Keyword      write
Separator    (
Separator    )
Separator    ;
Identifier    convertlow
Identifier    low
Operator      :=
Identifier    low
Operator      +
Separator    ;
Identifier    step#xy
Separator    }

```

3. Test case: > 20

```

convert (fahr : integer)
{
    return 5*(fahr-32)/9;
}

%%

integer low, high, step#xy;

read (low, high, step#xy);
while (low < high)
{
    write (low);
    write (convert[low]);
    low := low + step#xy;
}
if (high < low)
    return high;
else if (high = low)
    return low;
else
{
    return high;
    return low;
}
fi

```


Token	Lexeme
Identifier	convert
Separator	(
Identifier	fahr
Operator	:
Separator)
Identifier	integer
Separator	{
Identifier	return
Number	5
Separator	(
Operator	-
Number	2
Operator	/
Separator	;
Identifier	fahr
Separator	}
Operator	%
Identifier	integer
Separator	,
Identifier	low
Separator	,
Identifier	high
Separator	;
Identifier	step#xy
Keyword	read
Separator	(
Separator	,
Identifier	low
Separator	,
Identifier	high
Separator)
Separator	;
Identifier	step#xy
Keyword	while
Separator	(
Identifier	low
Operator	<
Separator)
Identifier	high
Separator	{
Keyword	write
Separator	(
Separator)
Separator	;
Identifier	low
Keyword	write
Separator	(
Separator)
Separator	;
Identifier	convertlow
Identifier	low
Operator	:=
Identifier	low
Operator	+
Separator	;
Identifier	step#xy
Separator	}