Concepts of Programming Languages, CSCI 305, Fall 2014
Lexical Analyzer for MiniGo14 in C#, Due Wednesday Oct. ~~10~~ 13 by midnight

Updated: Oct. 6, 2014

MiniGo14 Programs

A MiniGo14 program contains a function with the name 'main'. This is the function called by the operating system. The program can have one or more function definitions.

MiniGo14 allows arbitrary spaces, tabs and newlines except within a variable and an opening { cannot appear by itself on a line. Spaces are not required to separate tokens except between a floating point number and an identifier (to disambiguate 5.204enum and 5.204e3).

There are two forms of comments:

1. Line comments start with the character sequence // and stop at the end of the line. A line comment acts like a newline.
2. General comments start with the character sequence /* and continue through the character sequence */. A general comment containing one or more newlines acts like a newline, otherwise it acts like a space.

Comments do not nest.

Within MiniGo programs, comments can only contain lower case letters (a-z, _), digits (0-9), period (.), comma (,), underscore (_), star (*), colon (:), \t and/or \n.

The lexical analyzer should not pass comments to the parser.

Identifiers and Constants

MiniGo14 identifiers (variable and function names) names can be of any length. Identifiers must start with a character a-z or _. The reminder of the identifier name can contain characters a-z, _ or digits. Identifiers names cannot be keywords (e.g. const, else, default, float, for, func, if, int, return, var).

There are two forms of numeric constants, integer and float. The lexical analyzer should return these as separate tokens.

- Integer (indicated by int_lit): Can begin with a sign, is either 0 or starts with a non-zero digit. The non-zero integers can have any number of digits.
- Float (indicated by float_lit): Has an integer part, a decimal point, a fractional part, and an exponent part. The integer and fractional part comprise decimal digits; the exponent part is an e or E followed by an optionally signed decimal exponent.

One of the integer part or the fractional part may be elided; one of the decimal point or the exponent maybe elided.

Examples: 0.   72.40   072.40   2.71828   1.e+0   6.67428e-11   0076.   .25   .12345E+5   0.45e10

Declarations consist of "var", a list of one or more identifiers followed by a data type (int or float). Constant declarations are as above, except that "var" is replaced by "const".

Expressions

An expression specifies the computation of a value by applying operators and functions to operands. Operands denote the elementary values in an expression. An operand may be a literal, an identifier denoting a constant, variable, or function, or a parenthesized expression.

Unary operators consist of a unary operator followed by the operand. Binary expressions consist of an operand, followed by a binary operator, followed by an operand. Mixed mode expressions and assignments are allowed.

- Unary: - , +
- Binary: +, -, *, /

Comparison operators:

- ==
- !=
- <
- <=
- >
- >=

Precedence:

Highest	**()** (Parentheses increase the precedence of the operations.)
	* /
	+ -
	== != < <= > >=
Lowest	= (assignment operator)

All operators except the unary operators associate from left to right.

Binary operators of the same precedence associate from left to right. For instance, x / y * z is the same as (x / y) * z

Arithmetic operators apply to numeric values and yield a result of the same type as the first operand. The four standard arithmetic operators (+, -, *, /) apply to integer and floating-point.

Program Statements

In the following, statements may consist of a single statement, a block of statements, or nothing. Blocks of statements begin with a { and end with a }.

Assignment
- identifier = expression

If statement
- "if" condition block "else" statement_block
- The "else" statement_block can be elided

For statement
- "for" condition statement_block

Function call
- function_name(argument list);
- The argument list (actual parameter list) is a comma-separated list of expressions.

Return
- return expression;

Functions

Function definitions:
- Functions definitions have the format:
  "func" function_name(parameter list) return_type { function body }
- The parameter list (formal parameter list) is a comma-separated list of data type variable name pairs.
- The parameter list is optional.
- Only *int* and *float* are allowed for the return type, and the return type is optional
- Recursion is allowed.

Not Allowed

MiniGo14does not allow:
- Preprocessor commands
- I/0
- ~~Looping~~
- Character or strings
- Booleans

- Octal or hexadecimal numbers
- Composite types: arrays, strings, structures or unions
- Shifting

Assignment

Create a table-driven lexical analyzer for MiniGo14 in C#. Your lexical analyzer should identify the following tokens. Feel free to use a keyword lookup to distinguish the keywords from identifiers.

assign (=)
comma (,)
const
comp_op (==, !=, =<, =>, <, >)
else
default
float
float_lit
for
func
id
if
int
int_lit
lbrace
lparen
op (+, -, *, /)
rbrace
return
rparen
var

The interface to your program must:

- Describe the purpose of the lexical analyzer and how to use it.
- Allow the user to repeatedly enter a path to a file containing a MiniGo14 program. (If the lexical analyzer is in the middle of scanning a program, warn the user that the scanner is not done.)
- When a legal path is given, allow the user to repeatedly request a token, until the MiniGo14 program has been scanned or the user decides to quit scanning.
- Display the token identified and the lexeme associated with the token.
- Display helpful errors when the lexical analyzer finds an error. Display what characters caused the error.
- Allow the user to exit the program.

For maximum credit:

- Your program should be well commented, which includes descriptive variable, class and method names.
- Your program should be well designed.
- Your program should not include any extra, unused code.

Grading:

	Points	
Describe the purpose of the lexical analyzer and how to use it.	5	
Allow the user to repeatedly enter a path to a file containing a MiniGo14 program. (If the lexical analyzer is in the middle of scanning a program, warn the user that the scanner is not done.)	5	
When a legal path is given, allow the user to repeatedly request a token, until the MiniGo14 program has been scanned or the user decides to quit scanning, and displays the token identified and the lexeme associated with the token.	60	
Display helpful errors when the lexical analyzer finds an error. Display what characters caused the error.	10	
Allow the user to exit the program.	5	
Program is well commented, which includes descriptive variable, class and method names	5	
Program is well designed	5	
Program does not include any extra, unused code.	5	

You cannot use the Unix tool lex for this assignment.

Here is a sample MiniGo14 program:

```
/* here is a sample program.
   the lexical analyzer should not detect errors in this program. */

        func  main(argc int) int  {                              // function called by os
                var x, y, z int
                x=0
                y=1
                z = sum(x,y)
                y = z+mult(x,y)
                return z
        }

        func sum(a int, b int) int {
                return a+b
        }

        func mult(a int, b int) int {
                return a*b
        }
```

To submit your program, zip the directory containing the .sln file and email the zipped file to me. You may include the bin file or not.

Also, turn in a paper, human-readable copy of your scanning table. You may include the bin file or not. You do not need to turn in a paper copy of your nfa or dfa.