CS 280
Fall 2020
Programming Assignment 1

February 22, 2021

Due Date: Wednesday, March 10, 2021, 23:59
Total Points: 20

In this programming assignment, you will be building a lexical analyzer for small programming language and a program to test it. This assignment will be followed by two other assignments to build a parser and interpreter to the same language. Although, we are not concerned about the syntax definitions of the language in this assignment, we intend to introduce it ahead of Programming Assignment 2 in order to show the language reserved words, constants, and operators. The syntax definitions of a Fortran-Like small programming language are given below using EBNF notations. The details of the meanings (i.e. semantics) of the language constructs will be given later on.

```
Prog = PROGRAM IDENT {Decl} {Stmt} END PROGRAM IDENT
Decl = Type : VarList
Type = INTEGER | REAL |CHAR
VarList = Var {,Var}
Stmt = AssigStmt | IfStmt | PrintStmt |ReadStmt
PrintStmt := PRINT , ExprList
IfStmt = IF (LogicExpr) THEN {Stmt} END IF
AssignStmt = Var = Expr
ReadStmt = READ , VarList
ExprList = Expr {, Expr}
Expr = Term {(+|-) Term}
Term = SFactor {(*|/) SFactor}
SFactor = Sign Factor | Factor
LogicExpr = Expr (== | <) Expr
Var = IDENT
Sign = + | -
Factor = IDENT | ICONST | RCONST | SCONST | (Expr)
```

Based on the language definitions, the lexical rules of the language and the assigned tokens to terminals are as follows:

1. The language has identifiers, referred to by *ident* terminal, which are defined to be a letter followed by zero or more letters or digit. It is defined as:
```
ident := Letter {(Letter|Digit)}
Letter := [a-z A-Z]
Digit := [0-9]
```

   The token for an identifier is IDENT.

2. Integer constants, referred to by *iconst* terminal, are defined as one or more digits. It is defined as:
```
iconst := [0-9]+
```
   The token for an integer constant is ICONST.

3. Real constants, referred to by *rconst* terminal, are defined as zero or more digits followed by a decimal point (dot) and one or more digits. It is defined as:
```
Rconst := ([0-9]*)\.([0-9]+)
```
   The token for a real constant is RCONST. For example, real number constants such as 12.0 and .2 are accepted, but 2. is not.

4. String literals, referred to by *sconst* terminal, are defined as a sequence of characters delimited by single or double quotes, that should all appear on the same line. The assigned token for a string constant is SCONST. For example, `"Hello to CS 280."` Or `'Hello to CS 280.'` are string literals. There are no escape characters. However, a string delimited by single quotes can have double quotes character as one of the characters of the string, and similarly a string delimited by double quotes characters can have a single quote as one of the characters of the string. For example, `"welcome to smith's home"` or `'welcome to "smith" home'` are acceptable strings.

5. The reserved words of the language are: *program, end, print, read, if, then, integer, real, char.* These reserved words have the following tokens, respectively: PROGRAM, END, PRINT, READ, IF, THEN, INTEGER, REAL, and CHAR.

6. The operators of the language are: +, -, *, /, //, =, (,), ==, <. These operators are for add, subract, multiply, divide, concatenate, assignment, left parenthesis, right parenthesis, equality, and less than operations. They have the following tokens, respectively: PLUS, MINUS, MULT, DIV, CONCAT, ASSOP, LPAREN, RPAREN, EQUAL, LTHAN.

7. The colon and comma characters are terminals with the following tokens: COLON, COMA.

8. A comment is defined by all the characters following the exclamation mark "!" to the end of the line. A comment does not overlap one line. A recognized comment is ignored and does not have a token.

9. White spaces are skipped. However, white spaces between tokens are used to improve readability and can be used as a one way to delimit tokens.

10. An error will be denoted by the ERR token.

11. End of file will be denoted by the DONE token.

Lexical Analyzer Requirements:

You will write a lexical analyzer function, called `getNextToken`, and a driver program for testing it. The `getNextToken` function must have the following signature:

```
LexItem getNextToken (istream& in, int& linenumber);
```

The first argument to `getNextToken` is a reference to an istream object that the function should read from. The second argument to `getNextToken` is a reference to an integer that contains the current line number. `getNextToken` should update this integer every time it reads a newline from the input stream. `getNextToken` returns a `LexItem` object. A `LexItem` is a class that contains a token, a string for the lexeme, and the line number as data members.

A header file, `lex.h`, is provided for you. It contains a definition of the `LexItem` class, and a definition of an enumerated type of token symbols, called `Token`. You MUST use the header file that is provided. You may NOT change it.

Note that the `getNextToken` function performs the following:

1. Any error detected by the lexical analyzer should result in a `LexItem` object to be returned with the ERR token, and the lexeme value equal to the string recognized when the error was detected.

2. Note also that both ERR and DONE are unrecoverable. Once the `getNextToken` function returns a `LexItem` object for either of these tokens, you shouldn't call `getNextToken` again.

3. Tokens may be separated by spaces, but in most cases are not required to be. For example, the input characters "3+7" and the input characters "3 + 7" will both result in the sequence of tokens ICONST PLUS ICONST. Similarly, The input characters

   "Hello" "World", and the input characters "Hello""World"

   will both result in the token sequence SCONST SCONST.

Testing Program Requirements:

It is recommended to implement the lexical analyzer in one source file, and the main test program in another source file. The testing program is a `main()` function that takes several command line flags. The notations for each input flag are as follows:

- -v (optional): if present, every token is printed when it is seen followed by its lexeme between parentheses.
- -iconsts (optional): if present, prints out all the unique integer constants in numeric order.
- -rconsts (optional): if present, prints out all the unique real constants in numeric order.
- -sconsts (optional): if present, prints out all the unique string constants in alphabetical order
- -ids (optional): if present, prints out all of the unique identifiers in alphabetical order.
- filename argument must be passed to main function. Your program should open the file and read from that filename.

Note, your testing program should apply the following rules:

1. The flag arguments (arguments that begin with a dash) may appear in any order, and may appear multiple times. Only the last one is considered.
2. There can be at most one file name specified on the command line. If more than one filename is provided, the program should print on a new line the message "ONLY ONE FILE NAME ALLOWED" and it should stop running. If no file name is provided, the program should print on a new line the message "NO SPECIFIED INPUT FILE NAME FOUND", and should stop running.
3. No other flags are permitted. If an unrecognized flag is present, the program should print on a new line the message "UNRECOGNIZED FLAG {arg}", where {arg} is whatever flag was given, and it should stop running.
4. If the program cannot open a filename that is given, the program should print on a new line the message "CANNOT OPEN THE FILE {arg}", where {arg} is the filename given, and it should stop running.
5. If `getNextToken` function returns ERR, the program should print "Error in line N ({lexeme})", where N is the line number of the token in the input file and lexeme is its corresponding lexeme, and then it should stop running. For example, a file that contains an invalid real constant, as 15., in line 1 of the file, the program should print the message:

```
Error in line 1 (15)
```

6. The program should repeatedly call `getNextToken` until it returns DONE or ERR. If it returns DONE, the program prints summary information, then handles the flags -sconsts, -iconsts, rconsts and -ids, in that order.
   The summary information are as follows:

```
Lines: L
Tokens: N
```

Where L is the number of input lines and N is the number of tokens (not counting DONE). If L is zero, no further lines are printed.

7. If the -v option is present, the program should print each token as it is read and recognized, one token per line. The output format for the token is the token name in all capital letters (for example, the token LPAREN should be printed out as the string LPAREN. In the case of the tokens IDENT, ICONST, RCONST, and SCONST, the token name should be followed by a space and the lexeme in parentheses. For example, if the identifier "circle" and a string literal "The center of the circle through these points is" are recognized, the -v output for them would be:

```
IDENT (circle)
SCONST (The center of the circle through these points is)
```

8. The -sconsts option should cause the program to print the label STRINGS: on a line by itself, followed by every unique string constant found, one string per line without double quotes, in alphabetical order. If there are no SCONSTs in the input, then nothing is printed.
9. The -iconsts option should cause the program to print the label INTEGERS: on a line by itself, followed by every unique integer constant found, one integer per line, in numeric order. If there are no ICONSTs in the input, then nothing is printed.
10. The -rconsts option should cause the program to print the label REALS: on a line by itself, followed by every unique real constant found, one real number per line, in numeric order. If there are no RCONSTs in the input, then nothing is printed.
11. The -ids option should cause the program to print the label IDENTIFIERS: followed by a comma-separated list of every identifier found, in alphabetical order. If there are no IDENTs in the input, then nothing is printed.