CS 280
Spring 2023
Programming Assignment 1

Building a Lexical Analyzer for the SPL Language

February 16, 2023

Due Date: Sunday, March 5, 2023, 23:59
Total Points: 20

In this programming assignment, you will be building a lexical analyzer for small programming language, called Simple Perl-Like (SPL), and a program to test it. This assignment will be followed by two other assignments to build a parser and an interpreter to the SPL 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 determine the language terminals: reserved words, constants, identifier(s), and operators. The syntax definitions of the SPL language are given below using EBNF notations. However, the details of the meanings (i.e. semantics) of the language constructs will be given later on.

```
1.  Prog ::= StmtList
2.  StmtList ::= Stmt ;{ Stmt; }
3.  Stmt ::= AssignStme | WriteLnStmt | IfStmt
4.  WriteLnStmt ::= WRITELN (ExprList)
5.  IfStmt ::= IF (Expr) ‘{‘ StmtList ‘}’ [ ELSE ‘{‘ StmtList ‘}’ ]
6.  AssignStmt ::= Var = Expr
7.  Var ::= NIDENT | SIDENT
8.  ExprList ::= Expr { , Expr }
9.  Expr ::= RelExpr  [(-eq|==) RelExpr ]
10. RelExpr ::= AddExpr  [ ( -lt | -gt | < | > ) AddExpr ]
11. AddExpr :: MultExpr { ( + | - | .) MultExpr }
12. MultExpr ::= ExponExpr { ( * | / | **) ExponExpr }
13. ExponExpr ::= UnaryExpr { ^ UnaryExpr }
14. UnaryExpr ::= [( - | + )] PrimaryExpr
15. PrimaryExpr ::= IDENT | SIDENT | NIDENT | ICONST | RCONST | SCONST |
    (Expr)
```

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

1. The language has general identifiers, referred to by *IDENT* terminal, which are defined as a word that starts by a letter or an underscore ‘_’, and followed by zero or more letters, digits, or underscores ‘_’ characters. Note that all identifiers are case sensitive. It is defined as:

   ```
   IDENT := [Letter _] {( Letter | Digit | _ )}
   Letter := [a-z A-Z]
   Digit := [0-9]
   ```

2. The language variables are either numeric scalar variables or string scalar variables. Numeric variables start by a “$” and followed by an IDENT. While a string variable starts by “@” and followed by an IDENT. Their definitions are as follows:

   NIDENT := $ IDENT
   SIDENT := @ IDENT

3. Integer constant is referred to by *ICONST* terminal, which is defined as one or more digits. It is defined as:

   ```
   ICONST := [0-9]+
   ```

4. Real constant is a fixed-point real number referred to by *RCONST* terminal, which is defined as one or more digits followed by a decimal point (dot) and zero or more digits. It is defined as:

   ```
   RCONST := ([0-9]+)\.([0-9]*)
   ```

   For example, real number constants such as 12.0, and 0.2, 2. are accepted as real constants, but .2, and 2.45.2 are not. Note that “.2” is recognized as a dot (CAT operator) followed by the integer constant 2.

5. String literals is referred to by *SCONST* terminal, which is defined as a sequence of characters delimited by single quotes, that should all appear on the same line. For example, ‘Hello to CS 280.’ is a string literal. While, “Hello to CS 280.” Or ‘Hello to CS 280.” are not.

6. The reserved words of the language are: writeln, *if*, *else*. These reserved words have the following tokens, respectively: WRITELN, IF, ELSE.

7. The operators of the language are: +, -, *, /, ^, =, (,), {, }, ==, >, <, . (dot), ** (repeat), -eq, -lt, and -gt. These operators are for add, subtract, multiply, divide, exponent, assignment, left parenthesis, right parenthesis, numeric equality, numeric greater than, numeric less than, string concatenation, string repetition, string equality, string less-than, and string greater-than operations, respectively. They have the following tokens, respectively: PLUS, MINUS, MULT, DIV, EXPONENT, ASSOP, NEQ, NGTHAN, NLTHAN, CAT, SREPEAT, SEQ,

SLTHAN, and SGTHAN. Note that the string comparison operators -eq, -lt, and -gt are not case sensitive.

8. The semicolon, comma, left parenthesis, right parenthesis, left braces, and right braces characters are terminals with the following tokens: SEMICOL and COMMA, LPAREN, RPAREN, LBRACES, and RBRACES, respectively.

9. A comment is defined by all the characters following the characters "#" to the end of line. A recognized comment is skipped and does not have a token.

10. 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.

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

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

Lexical Analyzer Requirements:

A header file, `lex.h`, is provided for you. It contains the definitions of the `LexItem` class, and an enumerated type of token symbols, called `Token,` and the definitions of three functions to be implemented. These are:

```
extern ostream& operator<<(ostream& out, const LexItem& tok);
extern LexItem id_or_kw(const string& lexeme, int linenum);
extern LexItem getNextToken(istream& in, int& linenum);
```

You MUST use the header file that is provided. You may NOT change it.

I. You will write the lexical analyzer function, called `getNextToken,` in the file "lex.cpp". 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.

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.

II. You will implement the id_or_kw() function. Id_or_kw function accepts a reference to a string of a general identifier lexeme (i.e., keyword, IDENT, SIDENT, or NIDENT) and a line number and returns a LexItem object. It searches for the lexeme in a directory that maps a string value of a keyword to its corresponding Token value, and it returns a LexItem object containing the keyword Token if it is found. Otherwise, it returns a LexItem object containing a token for one of the possible types of identifiers (i.e., IDENT, SIDENT, or NIDENT).

III. You will implement the overloaded function operator<<. The operator<< function accepts a reference to an ostream object and a reference to a LexItem object, and returns a reference to the ostream object. The operator<< function should print out the string value of the Token in the tok object. If the Token is either an IDENT, NIDENT, SIDENT, ICONST, RCONST, SCONST, it will print out its token followed by its lexeme between parentheses. See the example in the slides.

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 input flags are as follows:

- -v (optional): if present, every token is printed out when it is seen followed by its lexeme between parentheses.
- -nconst (optional): if present, prints out all the unique numeric constants (i.e., integer or real) in numeric order.
- -sconst (optional): if present, prints out all the unique string constants in alphabetical order

- -ident (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 appearance of the same flag 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 IS 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.". Then the program should stop running.

3. 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. Then the program 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. Then the program 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 the list of all tokens if the "-v" is specified, followed by the summary information, then handles the flags "-idents", "-nconst", and "-sconst" in this order. The summary information are as follows:

   ```
   Lines: L
   Total Tokens: M
   Identifiers: N
   Numbers: O
   Strings: P
   ```

   Where L is the number of input lines, M is the number of tokens (not counting DONE), N is the number of identifiers tokens (e.g., IDENT, NIDENT, and SIDENT), O is the number of numeric constants, and P is the number of string literals. If the file is empty the value of L is zero, and the following output message is displayed.

```
Lines: 0
Empty File.
```

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, and NCONST the token name should be followed by a space and the lexeme in parentheses. In case of the token SCONST, the token name should be followed by a space and the lexeme between single quotes (‘’). For example, if an identifier “$circle” and a string literal ‘The center of the circle through these points is’ are recognized, the -v output for them would be:

```
NIDENT ($circle)
SCONST 'The center of the circle through these points is)
```

8. The -sconst 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 with single quotes, in alphabetical order. If there are no SCONSTs in the input, then nothing is printed.

9. The -nconsts option should cause the program to print the label NUMBERS: on a line by itself, followed by every unique numeric constant found, one number per line, in numeric order. If there are no NCONSTs in the input, then nothing is printed.

10. The -ident 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 identifiers in the input, then nothing is printed.

<u>Note:</u>
You are provided by a set of 16 test case files associated with Programming Assignment 1. Vocareum automatic grading will be based on these testing files. You may use them to check and test your implementation. These are available in compressed archive “PA 1 Test Cases.zip” on Canvas assignment. The testing case of each file is defined in the Grading table below.

Submission Guidelines

1.1. Submit all your implementation files for the “lex.cpp” and the testing program through Vocareum. The “lex.h” header file will be propagated to your Work Directory.

1.2. Submissions after the due date are accepted with a fixed penalty of 25% from the student’s score. No submission is accepted after Wednesday 11:59 pm, March 8, 2023.

Grading Table

Case	Test File	Points
1	Successful compilation	1
2	Cannot Open the File (cantopen)	1
3	Empty File (emptyfile)	1
4	Only one file name allowed (onefileonly)	1
5	No Specified Input File	1
6	Integers (ints)	1
7	Bad argument (badarg)	1
8	Valid strings (validstr)	1
9	Invalid strings I (invstr1)	1
10	Invalid strings II (invstr2)	1
11	Noflags (noflags)	1
12	Invalid Real (realerr)	1
13	Valid operators (validops)	1
14	Invalid Symbol (invsymbol)	1
15	Numeric constants (numerics)	2
16	All Identifiers (idents)	2
17	All flags set (allflags)	2
	Total	**20**