CS 280
Spring 2024
Programming Assignment 1

Building a Lexical Analyzer for

A Simple Fortran 95-Like Language

February 19, 2024

Due Date: Sunday, March 3rd, 2024, 23:59
Total Points: 20

In this programming assignment, you will be building a lexical analyzer for small programming language, called Simple Fortran95-Like (SFort95), and a program to test it. This assignment will be followed by two other assignments to build a parser and an interpreter to the 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, and operators. The syntax definitions of the SFort95 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 ::= PROGRAM IDENT {Decl} {Stmt} END PROGRAM IDENT
2.  Decl ::= Type :: VarList
3.  Type ::= INTEGER | REAL | CHARARACTER [\(LEN = (ICONST \ *) \)]
4.  VarList ::= Var [= Expr] {, Var [= Expr]}
5.  Stmt ::= AssigStmt | BlockIfStmt | PrintStmt | SimpleIfStmt
6.  PrintStmt ::= PRINT *, ExprList
7.  BlockIfStmt ::= IF (RelExpr) THEN {Stmt} [ELSE {Stmt}] END IF
8.  SimpleIfStmt ::= IF (RelExpr) SimpleStmt
9.  SimpleStmt ::= AssigStmt | PrintStmt
10. AssignStmt ::= Var = Expr
11. ExprList ::= Expr {, Expr}
12. RelExpr ::= Expr [ ( == | < | > ) Expr ]
13. Expr ::= MultExpr { ( + | - | // ) MultExpr }
14. MultExpr ::= TermExpr { ( * | / ) TermExpr }
15. TermExpr ::= SFactor { ** SFactor }
16. SFactor ::= [+ | -] Factor
```

17. `Var = IDENT`
18. `Factor ::= IDENT | 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 identifiers are referred to by the *IDENT* terminal, which is defined as a sequence of characters that starts by a letter, and followed by zero or more letters, digits, or underscores '_' characters. **Note that all identifiers are case sensitive.** *IDENT* regular expression is defined as:

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

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

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

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

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

   For example, real number constants such as 12.0, 0.2, and .5 are accepted as real constants, but 2. is not. Note that "2." is recognized as an integer followed by a DOT.

4. A string literal is referred to by *SCONST* terminal, which is defined as a sequence of characters delimited by either single or double quotes, that should all appear on the same line. For example,
   `'Hello to CS 280.'` or `"Hello to CS 280."` are valid string literals. While, `'Hello to CS 280."` Or `"Hello to CS 280.'` are not.

5. The reserved words of the language and their corresponding tokens are:

Reserved Words	Tokens
program	PROGRAM
end	END
else	ELSE
if	IF
integer	INTEGER
real	REAL

character	CHARACTER
print	PRINT
len	LEN

Note: Reserved words are not case sensitive.

6. The operators of the language and their corresponding tokens are:

Operator Symbol	Token	Description
+	PLUS	Arithmetic addition or concatenation
-	MINUS	Arithmetic subtraction
*	MULT	Multiplication
/	DIV	Division
**	POW	Exponentiation
=	ASSOP	Assignment operator
==	EQ	Equality
<	LTHAN	Less than operator
>	GTHAN	Greater then operator
//	CAT	Concatenation

7. The delimiters of the language are:

Character	Token	Description
,	COMMA	Comma
(	LPAREN	Left Parenthesis
)	RPAREN	Right parenthesis
::	DCOLON	Double Colons
.	DOT	Dot
*	DEF	Default mark

8. A comment is defined by all the characters following the character "!" as starting delimiter till the end of the line. A recognized comment is skipped 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:

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 identifier lexeme (i.e., keyword, or IDENT) 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 lexeme, the keyword Token, and the line number, if it is found, or IDENT Token otherwise.

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 Token contained in the referenced tok object. If the Token is either ICONST, RCONST, or BCONST it will print out its token followed by the lexeme between two parenthses. If the Token is an IDENT, it will print out its token followed by the lexeme between two single quotes. If the Token is an SCONST, it will print out its token followed by the lexeme between two double quotes. See the examples 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 arguments. The notations for input arguments are as follows:

- -all (optional): if present, every token is printed out when it is seen followed by its lexeme using the format described below.
- -int (optional): if present, prints out all the unique integer constants in numeric order.
- -real (optional): if present, prints out all the unique real constants in numeric order.
- -str (optional): if present, prints out all the unique string constants in order
- -id (optional): if present, prints out all of the unique identifiers in alphanumeric order. Each identifier is followed by the number of its occurrences between parentheses.
- -kw (optional): if present, prints out all of the unique keywords in alphabetical order. Each keyword is followed by the number of its occurrences between parentheses.
- 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. If the file is empty, the program displays the message : “Empty File.” on a new line, and 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 out the message “Error in line N: Unrecognized Lexeme {lexeme}”, where N is the line number in the input file in which an unrecognized lexeme is detected, and then it should stop running. For example, a file that contains an unrecognized symbol, such as “?”, in line 1 of the file, the program should print out the message:

```
Error in line 1: Unrecognized Lexeme {?}
```

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 “-all” is specified, followed by the summary information, then handles the flags using the following order: “-id”, “-kw“, -int”, “-real”, then “-str”. If no flags are found, only the summary information is displayed. The summary information is displayed according to following format:

```
Lines: L
Total Tokens: M
Identifiers: N
Integers: O
Reals: P
Strings: Q
```

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), O is the number of integer constants, P is the number of real constants, and Q is the number of string literals. If the file is empty, the following output message is displayed.

```
Empty File.
```

7. If the -all 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 IDENT, ICONST, RCONST, SCONST, and ERR tokens, the token name should be followed

by a colon and the lexeme between double quotes. For example, if an identifier "circle" and a string literal 'The center of the circle through these points is' are recognized, the -all output for them would be:

```
IDENT: 'circle'
SCONST: "The center of the circle through these points is"
```

8. The -id option should cause the program to print the label IDENTIFIERS: followed by a comma-separated list of every identifier found, in alphanumeric order. Each identifier is followed by the number of its occurrences between parentheses. If there are no identifiers in the input, then nothing is printed.

9. The -kw option should cause the program to print the label KEYWORDS: followed by a comma-separated list of every keyword found, in alphanumeric order. Each keyword is followed by the number of its occurrences between parentheses. If there are no keywords in the input, then nothing is printed.

10. The -int option should cause the program to print the label INTEGERS: on a line by itself, followed by comma-separated list of every unique integer constant found, in numeric order. If there are no ICONSTs in the input, then nothing is printed.

11. The -real option should cause the program to print the label REALS: on a line by itself, followed by comma-separated list of every unique real constant found, in numeric order. If there are no RCONSTs in the input, then nothing is printed.

12. The -str option should cause the program to print the label STRINGS: on a line by itself, followed by comma-separated list of every unique string constant found, delimited by double quotes, in alphabetical order. If there are no SCONSTs in the input, then nothing is printed.

Note:

You are provided by a set of 19 testing 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, March 6, 2024, 23:59.

Grading Table

Case	Test File	Points
1	Successful compilation	1
2	Cannot Open the File (cantopen)	1
3	Empty File (empty)	1
4	Only one file name allowed (onefileonly)	1
5	No Specified Input File (nofilename)	1
6	Integer constants (integers)	1
7	Bad argument (badarg)	1
8	Real constant error (realserr)	1
9	Invalid strings I (invstr1)	1
10	Invalid strings II (invstr2)	1
11	Noflags (noflags)	1
12	Comments (comments)	1
13	Valid operators (validops)	1
14	Invalid Symbol (invsymbol)	1
15	All flags set (allflags) (-all -id -kw -str -int -real)	1
16	Identifier (idents)	1
17	Keywords (keywords)	1
18	Constants only (constants)	1
19	Program 1 with (-int, -real) flags set (prog1)	1
20	Program 2 with (-id, -kw, -str) flags set (prog2)	1
	Total	**20**