CS 280
Spring 2025
Project Assignment 1

Building a Lexical Analyzer for Simple Ada-Like Language

February 24, 2025

Due Date: March 12, 2025
Total Points: 20

In this programming assignment, you will be building a lexical analyzer for small programming language, called Simple Ada-Like (SADAL), 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 SADAL 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 ::= PROCEDURE ProcName IS ProcBody
2.  ProcBody ::= DeclPart BEGIN StmtList END ProcName ;
3.  ProcName ::= IDENT
4.  DeclPart ::= DeclStmt { DeclStmt }
5.  DeclStmt ::= IDENT {, IDENT } : [CONSTANT] Type [(Range)] [:= Expr] ;
6.  Type ::= INTEGER | FLOAT | BOOLEAN | STRING | CHARACTER
7.  StmtList ::= Stmt  { Stmt }
8.  Stmt ::= AssignStmt | PrintStmts | GetStmt | IfStmt
9.  PrintStmts ::= (PutLine | Put) ( Expr) ;
10. GetStmt := Get (Var) ;
11. IfStmt ::= IF Expr THEN StmtList { ELSIF Expr THEN StmtList }   [ ELSE
    StmtList ] END IF ;
12. AssignStmt ::= Var := Expr ;
13. Expr ::= Relation {(AND | OR) Relation }
14. Relation ::= SimpleExpr [  ( = | /= | < | <= | > | >= )  SimpleExpr ]
15. SimpleExpr ::= STerm {  ( + | - | & ) STerm }
16. STerm ::= [ ( + | - ) ] Term
17. Term  ::= Factor { ( * | / | MOD ) Factor }
18. Factor ::= Primary [** Primary ] | NOT Primary
19. Primary ::= Name | ICONST | FCONST | SCONST | BCONST | CCONST | (Expr)
20. Name ::= IDENT [ ( Range ) ]
21. Range ::= SimpleExpr [. . SimpleExpr ]
```

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 can be followed by zero or more letters, digits, or underscore '_' characters, where no two or more consecutive underscores are allowed. **Note that all identifiers are case insensitive.** *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 a sequence of one or more digits followed by by an optional exponent. It is defined as:

   ```
   ICONST ::= Digit+ Exponent?
   Exponent ::= E ( +? | - ) Digit+
   ```

3. Floating-point constant is a real number referred to by *FCONST* terminal. It is defined as a sequence of one or more digits, forming the integer part, a decimal point, and followed by a one or more digits, forming the fractional part, and an optional exponent. The regular expression definition for FCONST is as follows:

   ```
   FCONST ::= Digit+ \. Digit+ Exponent?
   ```

   For example, the values 5.25, 1.75E-2, 0.75, and 1.0 are valid numeric values, while the values 5., .25, 5.7.4 are not. Note that "5." 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 double quotes. For example,
   `"Hello to CS 280."` are valid string literals. While, `'Hello to CS 280."` Or `"Hello to CS 280.'` are not.

5. A character constant is referred to by *CCONST*, which is defined as any ASCII character between single quotes.

6. A Boolean constant is referred to by *BCONST*, which is includes the two possible values "true" and "false".

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

Reserved Words	Tokens
procedure	PROCEDURE
String	STRING
else	ELSE
if	IF

Integer	INT
Float	FLOAT
Character	CHAR
Put	Put
PutLine	PUTLN
Get	GET
Boolean	BOOL
true	TRUE
false	FALSE
elsif	ELSIF
is	IS
end	END
begin	BEGIN
then	THEN
constant	CONST
mod	MOD
and	AND
Or	OR
not	NOT

Note: Reserved words are not case sensitive.

8. 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
:=	ASSOP	Assignment operator
=	EQ	Equality
/=	NEQ	Not Equality
<	LTHAN	Less than operator
>	GTHAN	Greater then operator
<=	LTE	Less than or equal
>=	GTE	Greater than or equal
mod	MOD	Modulus
&	CONCAT	Concatenation
and	AND	Logic Anding
or	OR	Logic Oring
not	NOT	Complement
**	EXP	Exponentiation

9. The delimiters of the language are:

Character	Token	Description
,	COMMA	Comma
;	SEMICOL	Semi-colon
(	LPAREN	Left Parenthesis

)	RPAREN	Right parenthesis
:	COLON	Colon
.	DOT	Dot

10. A comment is defined by all the characters on a line following the sequence of two dash characters, "–" till the end of line.

11. 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.

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

13. 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. However, there is one exception to be considered when the keyword token is either TRUE or FALSE, where the returned LexItem object would hold a BCONST token instead.

III. You will implement the overloaded operator 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 contained in the referenced tok. If the Token is either ICONST, RCONST, or BCONST it will print out a colon followed by its lexeme between two parentheses. If the Token is an IDENT, it will print out a colon followed by its lexeme between two angle brackets. If the Token is an SCONST, it will print out a colon followed by its lexeme between two double quotes. If the Token is a CCONST, it will print out a colon followed by its lexeme between two single quotes. Finally, if the Token is an ERR, it will print out the following message:

    "ERR: Unrecognized Lexeme {lexeme} in line N",

    where N is the line number in the input file. All printed out tokens should be followed by a newline.

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.
- -num (optional): if present, prints out all the unique numeric constants (Integers or Float) in numeric order.
- -str (optional): if present, prints out all the unique string or character constants in order.
- -id (optional): if present, prints out all of the unique identifiers in alphanumeric order.
- -kw (optional): if present, prints out all of the unique keywords in Token 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. 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 token as it has been specified in the overloaded operator function, `operator<<()`. Afterwards, the program 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: Unrecognized Lexeme {?} in line 1`
6. The program should repeatedly call `getNextToken` until it returns DONE or ERR. If it returns DONE, the program processes the input flags according to the following order:
   - Displaying the list of all tokens if the "-all" is specified, and
   - Displaying the summary information,

   Then any other specified flags would have their information displayed based on the following order: "-num", "-str", "-id", and "-kw".

   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
Numerals: N
```

```
Characters and Strings: O
Identifiers: P
Keywords: Q
```

Where L is the number of input lines, M is the number of tokens (not counting DONE), N is the number of total numeric constants (ICONST and FCONST), O is the total number of string and character constants, P is the total number of identifier tokens, and Q is the number of keywords. 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 formats of printed tokens are as explained in the overloaded operator function, `operator<<`, above.
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. 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 **Token value 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 -num option should cause the program to print the label NUMERIC CONSTANTS: on a line by itself, followed by comma-separated list of every unique real or integer constant found, in numeric order. If there are no ICONST or FCONSTs in the input, then nothing is printed.
11. The -str option should cause the program to print the label CHARACTERS AND STRINGS: on a line by itself, followed by comma-separated list of every unique string or character constant found, delimited by double quotes, in alphabetical order. If there are no SCONSTs and CCONSTs 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

- Please name your files as "firstinitial_lastname_lex.cpp" and "firstinitial_lastname_main.cpp". Where, "firstinitial" and "lastname" refer to your first name initial letter and last name, respectively. Uploading and submission of your implementations is via Vocareum. Follow the link on Canvas for PA 1 Submission page.

- Extended submission period of PA 1 will be allowed after the announced due date for 3 days with a fixed penalty of 25% deducted from the student's score. No submission is accepted after Saturday 11:59 pm, March 15, 2025.

Grading Table

Case	Test File	Points
1	No Specified Input File (nofile)	1
2	Cannot Open the File (cantopen)	1
3	Empty File (empty)	1
4	Only one file name allowed (onefileonly -all idents)	1
5	Bad argument (badarg with -al flag)	1
6	Identifiers (idents with -all and -id flags)	1
7	Real constant error (realerr with -all and -num flags)	1
8	Noflags (noflags)	1
9	Invalid strings I (invstr1 with -all -str)	1
10	Invalid strings II (invstr2 with -all and -str flags)	1
11	Comments (comments with -all, -id and -kw flags)	1
12	Invalid character constant 1 (invchar1 with -all flag)	1
13	Invalid character constant 2 (invchar2 with -all flag)	1
14	Invalid Symbol (invsymbol with -all and -kw flags)	1
15	Integers (integers with -all and -num flags)	1
16	Identifiers and keywords (ids_keywords with id and -kw flags)	1
17	Valid operators (validops with -all flag)	1
18	All Constants (constants with -all, -num, and -str flags)	1
19	Program 1 (prog1 with -id, num, -str, -kw flags)	1
	Successful compilation	1
	Total	**20**