PP1: Lexical Analysis

Date Due: **02/12/2021 Friday 11:59pm**

1 Goal

In the first programming project, you will get your compiler off to a great start by implementing the lexical analysis phase. For the first task of the front-end, you will use $flex$ to create a scanner for the Decaf programming language. A few transformations will be dealt with by a preprocessor, and then your scanner will run through the source program, recognizing Decaf tokens in the order in which they are read, until end-of-file is reached. For each token, your scanner will set its attributes appropriately (this will eventually be used by other components of your compiler) so that information about each token will be properly printed.

This is a fairly straightforward assignment and most students don't find it too time-consuming. Don't that lull you into procrastinating on getting started! If you have never used a tool like $flex$, learning its features and quirks will take some experimentation and debugging. Once you get up to speed, things should go relatively smoothly, but plan for enough time to thoroughly test your work to ensure its robustness before you submit.

Decaf shares many similarities with C/C++/Java, although not all features exactly match. Our hope is that the familiar syntax will make things easier on you, but do be aware of the differences.

2 Lexical Structure of Deaf

For the scanner, you are only concerned with being able to recognize and categorize the valid tokens from the input. Here is a summary of the token types in Decaf.

The following are keywords. They are all reserved.

void	*int*	*double*	*bool*
string	*class*	*interface*	*null*
this	*extends*	*implements*	*for*
while	*if*	*else*	*return*
break	*New*	*NewArray*	

An identifier is a sequence of letters, digits, and underscores, starting with a letter. Decaf is case-sensitive, e.g., if is a keyword, but IF is an identifier; *binky* and *Binky* are two distinct identifiers. Identifiers can be at most 31 characters long. Whitespace (i.e. spaces, tabs, and newlines) serves to separate tokens, but is otherwise ignored. Keywords and identifiers must be separated by whitespace or a token that is neither a keyword nor an identifier. $ifintthis$ is a single identifier, not three keywords. $if(23this$ scans as four tokens.

A boolean constant is either $true$ or $false$.

An integer constant can either be specified in decimal (base 10) or hexadecimal (base 16). A decimal integer is a sequence of decimal digits (0-9). A hexadecimal integer must begin with $0X$ or $0x$ (that is a zero, not the letter oh) and is followed by a sequence of hexadecimal digits. Hexadecimal digits include the decimal digits and the letters a through f (either upper or lowercase). Examples of valid integers: 8, 012, $0x0$, $0X12aE$

A double constant is a sequence of digits, a period, followed by any sequence of digits, maybe none. Thus, .12 is not a valid double but both 0.12 and 12. are valid. A double can also have an optional exponent, e.g., $12.2E+2$. For a double in this sort of scientific notation, the decimal point is required, the sign of the exponent is optional (if not specified, + is assumed), and the E can be lower or upper case. As above, $.12E + 2$ is invalid, but $12.E + 2$ is valid. Leading zeroes on the mantissa and exponent are allowed.

A string constant is a sequence of characters enclosed in double quotes. Strings can contain any character except a newline or double quote. A string must start and end on a single line, it cannot be split over multiple lines:

```
"this string is missing its close quote
this is not a part of the string above
```

Operators and punctuation characters used by the language includes:

```
+   -   *   /   %    <   <=   >   >=   =   ==   !=
&&   ||   !   ;   ,   .   [   ]   (   )   {   }
```

3 Starter Files

The starting files for this project are available from Blackboard as *pp1.tar.gz*. The directory contains the following files (the boldface entries are the ones you will need to modify):

Makefile	builds both preprocessor and scanner
dpp	Decaf preprocessor (binary file)
main.cc	main() for scanner
scanner.h	type definitions and prototype declarations for scanner
scanner.l	starting scanner skeleton
errors.h/.cc	error messages you are to use
utility.h/.cc	interface/implementation of various utility functions
samples/	directory of test input files
solution/	directory of solution executables

Copy the entire directory to your home directory. Your first order of business is to read through all the files to learn the lay of the land as well as absorb the helpful hints contained in the files.

You should NOT modify *scanner.h*, *errors.h* or *main.cc* since our grading scripts depend on your output matching our defined constants and behavior. You can (but are not likely to) modify *utility.h/.cc*. You will definitely need to modify *scanner.l*.

You should use our Makefile rather than directly invoking *flex* and *gcc* to build the project. The Makefile has targets for to build the dcc. Each reads input from stdin and you can use standard UNIX file redirection to read from a file. For example, to invoke your compiler (scanner) on a particular input file, you would use:

```
% dcc < samples/ident.frag
```

To help you debug, a solution executable "dcc" is provided under the solutions directory.

4 Using flex

You'll find that *flex* is not the most user-friendly tool. For example, if you put a space or newline in the wrong place, it will often print "syntax error" with no line number or hint of what the true problem is. It may take some delving into the manual, a little experimentation, and some patience to learn its quirks. Here are a few suggestions:

- Be careful about spaces within patterns (it's easy to accidentally allow a space to be interpreted as part of the pattern or signal the end of pattern prematurely if you aren't attentive).
- Never put newlines between a pattern and an action.
- When in doubt, parenthesize with the pattern to ensure you are getting the precedence you intend.
- Enclose each action in curly braces (although not required for a single-line action, better safe than sorry).
- Use the definitions section to define pattern substitutions (names like Digit, Exponent, etc.). It makes for much more readable rules that are easier to modify, build upon, and debug.
- Always put parens around the body of a definition to ensure the correct precedence is maintained when it is substituted.
- You must put curly braces around the definition name when you are using it in another definition or a pattern, without them it will only match the literal name.

5 Scanner Implementation

The *scanner.l* lex input file in the starter project contains a skeleton you must complete. The *yylval* global variable is used to record the value for each lexeme scanned and the *yylloc* global records the lexeme position (line number and column). The action for each pattern will update the global variables and return the appropriate token code. Your goal is to modify *scanner.l* to:

- skip over white space
- recognize all keywords and return the correct token from *scanner.h*
- recognize punctuation and single-char operators and return the ASCII value as the token
- recognize two-character operators and return the correct token
- recognize int, double, bool, and string constants, return the correct token and set appropriate field of *yylval*
- recognize identifiers, return the correct token and set appropriate fields of *yylval*
- record the line number and first and last column in *yylloc* for all tokens
- report lexical errors for improper strings, lengthy identifiers, and invalid characters

We recommend adding token types one at a time to *scanner.l*, testing after each addition. Be careful with characters that has special meaning to lex such as $*$ and $-$ (see docs for how/when to suppress special-ness). The patterns for integers, doubles, and strings will require careful testing to make sure all cases are covered (see man page for strtol/atof for converting strings to numbers).

Recording the position of each lexeme requires you to track the current line and column numbers (you will need global variables) and update them as the scanner reads the file, mostly likely incrementing the line count on each newline and the column on each token. There is code in the starter file that installs a function to be automatically included in each action which is much nicer that repeating the call everywhere!

Lastly you need to be sure that your scanner reports the various lexical errors. The action for an error case should call our *ReportError*() function with one of the standard error messages provided in *errors.h*. For each character that cannot be matched to any token pattern, report it and resume lexing at the following character. If a string erroneously contains a newline, report an error and resume lexing at the beginning of the next line. If an identifier is longer than the Decaf maximum (31 characters), report the error and truncate the identifier to the first 31 characters (discarding the rest), resume lexing at the next token.

6 Testing

In the starting project, there is a samples directory containing various input files and matching .out files which represent the expected output. You should diff your output against ours as a first step in testing. Now examine the test files and think about what cases aren't covered. Construct some of your own input files to further test your preprocessor and scanner. What formations look like numbers but aren't? What sequences might confuse your processing of comments or macros? This is exactly the sort of thought-process a compiler writer must go through. Any sort of input is fair game to be fed to a compiler and you'll want to be sure yours can handle anything that comes its way, correctly tokenizing it if possible or reporting some reasonable error if not.

Note that lexical analysis is responsible only for correctly breaking up the input stream and categorizing each token by type. The scanner will accept syntactically incorrect sequences such as:

```
int if array + 4.5 [ bool]
```

7 Hints

Some hints to help you code:

1. Our reference systems are the fox servers (fox01.cs.utsa.edu to fox06.cs.utsa.edu). Your assignments will be graded on these servers.

2. You can find The documents for *Flex* at http://dinosaur.compilertools.net/#flex. It is worthwhile to spend some time on *Flex*'s document before you start coding.

3. Additionally, there are some useful online *Flex* tutorials at here.

4. Google is always your best friend. You can find answers to almost all of your questions online.

5. The *scanner.l* file is compiled by *flex* command, which generates two files, *lex.yy.c* and *lex.yy.h.* Essentially, *flex* translates the *.l* file into C source files, which are later compiled to generate the scanner. Consequently, except the patterns, you will write mostly C code in the *scanner.l* file.

6. A typically *Flex* source file contains the following sections separated by "%%" and "%{":

```
%{
   /* section 1: header files, macros and declarations that are directly
      copied into lex.yy.c */
%}
   /* section 2: definitions (predefined regular expressions) */
```

```
%%
   /* section 3: rules */
%%
   /* section 4: user subroutines (C functions) that are directly copied
      into lex.yy.c */
```

7. A rule defines the regular expression of a token and the action associated with the token. That is, a rule is,

   ```
   regular_expression_for_token          {actions for token in C code}
   ```

   For example, to match the keyword "int", we may have a rule,

   ```
   "int"                                 {printf{"found int.\n"); return T_Int;}
   ```

   This rule specifies that "int" is a token, and after finding "int", the scanner should print "found int" and return *T_Int*.

8. The enumerate values to represent the tokens (i.e., *T_??*) are defined in *scanner.h*.

9. The scanner is invoked by *yylex()* function in the *main* function. Each time *yylex()* is called, it recognizes a token and executes the action associated with the token. You should be familiar with *main.cc* file, which controls the execution of the scanner and the output of *dcc*.

10. You need to properly set the value *yylval* associated with a token in its action. For example, for an integer constant token "10", its value *yylval* should be the integer 10. There are quite a few tokens require values.

11. You also need to record the starting and ending location of a token. A location has row and column numbers and is saved into the variable *yylloc*. *yylloc* is a *struct yyltype* typed variable, you should Google the definition of this type.

12. You will need to manually track the row and column numbers of current token. Each time a token is found, you should advance the current column number by *yyleng* (what is *yyleng*?). Each time a newline character is read, you should advance the current line number by one.

13. "DoBeforeEachAction" is good place to advance the column numbers. You should also initialize all your variables in "InitScanner."

14. You should read *main.cc* to understand how *yylval* and *yylloc* are used, so that you know how to properly set their values.

15. Note that both *yylval* and *yylloc* are global variables in *lex.yy.c*. These two variables will be automatically generated by *yacc/bison* in the future projects.

16. Do not forget punctuations, such as ";" and "(", and white spaces.

17. *Flex* prefers the longest match, then the first match, when a string matches two regular expressions.

8 Grading

This project is worth 50 points. Most of the points will be allocated for correctness. We will run your program through the given test files from the samples directory as well as other tests of our own, using $diff - w$ to compare your output to that of our solution.

9 Deliverables

Electronically submit your entire project to Blackboard. You should submit a tar.gz of the project directory. Be sure to include a brief README file, which is your chance to let us know about optional features you added for bonus points.

Because we grade the submissions using scripts, it is important that everyone uses the same directory structure as the original *pp1.tar.gz*, as specified in Section 3. More specifically, inside your tar.gz, there should be a directory named “pp1”, and all of your source files should be within the directory “pp1” without any sub-directories. There is no need to attach the *samples* directory. There is a 40% penalty for anyone who fails to follow this directory structure.