Assignment 01: Lexical Analysis

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 develop a scanner for the Decaf 19 programming language. 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 record its attributes appropriately (this will eventually be used by other components of your compiler) so that information about each token will be properly printed.

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  null   for
while   if     else         return
break   Print  ReadInteger  ReadLine
```

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 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]
```

4 Grading

This phase of the project is worth 5 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.

5 Deliverables

Electronically submit your entire project to D2L. You should submit a tar.gz of the project directory. Be sure to include a brief README file.

Because we grade the submissions using scripts, it is important that everyone uses the same directory structure. The uploaded folder should contain your source code, all dependencies, and a sub-folder called 'workdir' where you should put two files 'build.sh', and 'exec.sh'. Running 'build.sh' should build your project, and running 'exec.sh <filepath>' should execute your compiler on a source code file at <filepath>', and all output should be written to the standard output stream, so that the grader can use 'exec.sh <filepath> > <outputpath>' to redirect your output into files and compare with the ground truth.

6 Using flex

For lexicon analysis, you are allowed to use lexer generation tools, such as Flex. You can find The documents for *Flex* at http://dinosaur.compilertools.net/#flex.