PP1: Lexical Analysis

Handed Out: **Thursday, 9/4/2014, in class**
Date Due: **Friday, 9/19/2014, 11:59pm**

This assignment is **an individual project.** You may discuss your ideas with others, but your work should be your own.

Deliverables: Students in CISC 672: Complete the entire assignment as described.
Students in CISC 471: Do not complete the preprocessor. You can use the existing preprocessor.

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 Decaf

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 The Decaf Preprocessor

Before a Decaf compiler sees the input file, it will first be run past a preprocessor. A preprocessor is a filter that does text handling and re-arrangement before passing the input along to the real compiler. The usual C/C++ preprocessor manages preprocessor directives such as #include, #define, and #if, as well as tasks such as removing comments, concatenating adjacent string literals, and cleaning up whitespace. For fun, try running $gcc-E$ on various C input files to see the results of running the C preprocessor by itself to get a sense of the way this tool operates in an industrial-strength language.

A language preprocessor usually performs a few limited but important tasks. Often the messiness of these text-munging features is better handled in a separate program from the scanner proper. This allows the two tasks to operate more independently and as result, both end up being cleaner to implement.

Our Decaf preprocessor is going to be a very simple one. It will echo most input to the output unchanged. Ordinary tokens, string constants, whitespace, etc. are of no interest and just pass through as is. The preprocessor is only going to tackle two jobs: stripping comments and providing a simple #define mechanism.

Decaf adopts the two types of comments available in C++. A single-line comment is started by // and extends to the end of the line. Multi-line comments start with /* and end with the first subsequent $*/$. Any symbol is allowed in a comment except the sequence $*/$ which ends the current comment. Multi-line comments do not nest. Your preprocessor should consume all comments from the input stream and output only the newlines within multi-line comments, suppressing everything else. If a file ends with an unterminated comment, the preprocessor should report an error.

Any token which starts with the # symbol is handled as a preprocessor directive. The directives supported are $\#define$ to establish a new macro definition and $\#NAME$ to expand a previously defined name.

A Decaf macro definition has the form:

```
#define NAME replacement
```

The $NAME$ is a sequence of uppercase letters The *replacement* consists of all characters up to, but not including, the end of the line. Somewhat similar to the C preprocessor, this mechanism offers a find-and-replace substitution. However, in Decaf, to request expansion of the name, it must be preceded by a #. For example, after the preprocessor has seen this input:

```
#define COUNT 3 + 10
```

Subsequent occurrences of $\#COUNT$ will be replaced with $3 + 10$.

Your preprocessor can make the following simplifying assumptions:

- There must be exactly one space between #define and the name and one space between the name and the beginning of the replacement
- String literals are echoed unchanged (so "$inside\#COUNT$" will not be substituted)
- The replacement string is not re-processed, the characters are read and later echoed unchanged (if it contains another macro, it is not expanded, comments are not stripped, and so on)
- Macros with arguments are not supported
- The replacement can be entirely empty (and thus later use of the name will not trigger any replacement)
- A name can be redefined, the new replacement supercedes the previous from that point onward

A $\#define$ in the input stream must be followed by a properly formed definition, if not (lowercase name or some such), an invalid directive error is reported and the entire line is discarded. Any other use of # in the input must be followed by the name of an existing macro, anything else is reported as an invalid directive error and the token discarded.

4 Starter Files

The starting files for this project are available on the mlb.acad.ece.udel.edu machines at /usa/pollock/cisc672_471/decaf/pp1 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
dppmain.cc	main() for preprocessor
dpp.l	empty lex file for use in preprocessor (optional)
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
list.h	simple list class for storing a linear collection of elements
location.h	used to record the lexical position of a token or symbol

Copy the entire directory to your home directory. Your first task 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 *dppmain.cc/dpp.l* and *scanner.l*.

You should use our Makefile rather than directly invoking *flex* and *gcc* to build the project. The Makefile has targets for you to build the two separate prorams dpp and 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/program2.decaf
```

You can also test *dpp* by directly invoking it from the command-line. Note that provided main for *dcc* is already configured to automatically invoke *dpp* first to filter the input (so running *dcc* always runs *dpp* inside of itself as a first step).

5 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.

6 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.

7 Preprocessor

Although the preprocessor is a required part of this project (FOR CISC 672 students), you may decide not to implement it because of time constraints. In that case and FOR CISC 471 students, you can use the provided solution *dpp* executable instead. Be sure to modify the Makefile to not try to build a new *dpp* executable (change line 11 from saying PREPROCESSOR = dpp to just PREPROCESSOR =). Note (FOR CISC 672 students only), your maximum score can only be 40 points instead of 50 if you do not implement the preprocessor.

The preprocessor is small enough that you can write it in straight hand-coded C/C++ but it works out nicely in lex as well. Using C may be quickest given its familiarity, but learning how to use lex in other novel ways is also a worthwhile goal. We set up the starter files so that you can implement it either way depending on your preference. If you decide to use lex, we recommend that you first write the scanner to get your lex bearings and come back to implement the preprocessor. Our starter files will build a default "empty" preprocessor that echoes everything so do your testing on files that don't have comments or #defines to avoid this being an issue.

Comments are the easier of the preprocessor tasks, so tackle them first. Comments are suppressed; no output other than the newlines should make it through the preprocessor. Take care that comment characters inside string literals don't get misidentified as comments. If a file ends with an unclosed multi-line comment, an error is reported via a call to our *ReportError*() function. In order to match our output exactly, please use the standard error messages provided in *errors.h*.

The macro definition and replacement is a bit more complex, but your C/C++ skills should come in handy for the string manipulation. Every # in the input must be followed by either a proper define or a sequence of letters that identifies a previously defined name. Any other use of # is a reported as an invalid directive error and discarded.

You will need a hashtable to associate names with replacements. You can use STL hash map (although it's not part of the standard, most standard libraries include it). In that case make sure you use the correct include file (it should work on linux.cs.tamu.edu). You may also re-use a hashtable you implemented for a previous course or project, but you must have written the code yourself. The hashtable can be implemented as a set of C functions, an ADT, a C++ class, whatever you like. Do not worry about dynamically resizing your hashtable or anything fancy. Any reasonably efficient implementation will do. You should

put your hashtable implementation in its own file. You will need to modify the makefile to add the new source to the executable.

8 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]
```

9 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.

10 Deliverables

Electronically submit your entire project by committing and pushing to your bitbucket repository that you set up earlier and shared with us. You should submit a tar.gz of the project directory, called pp1-yourlastname.tar.gz. Be sure to include a brief README file, which is your chance to let us know about optional features you added for bonus points. The date of submission will be the last push date that you made to this project. **Remember that the deadline is Friday, 9/19/2014, 11:59pm**