P1: Lexical Analysis

Due on Midnight 1/23/2023

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 **lex** to create a scanner for the Decaf programming language. A preprocessor will deal with a few transformations, and then your scanner will run through the source program, recognizing Decaf tokens in the order in which they are read until the end-of-file is reached. For each token, your scanner will set its attributes appropriately (other components of your compiler will eventually use this) so that information about each token will be printed correctly.

This is a fairly straightforward assignment, and most students don't find it too time-consuming. Don't let that lull you into procrastinating on getting started! If you have never used a tool like **lex**, 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 it.

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

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) separates 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 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 decimal and 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 mentioned 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 include:

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

The Decaf Preprocessor

Before a Decaf compiler sees the input file, it will be run past a preprocessor. A preprocessor is a filter that handles text and re-arranges it before passing the input to the actual 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 understand how this tool operates in an industrial-strength language.

A language preprocessor usually performs a few limited but essential 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 a 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 will only 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 begin 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. The preprocessor should report an error if a file ends with an unterminated comment.

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. Similar to the C preprocessor, this mechanism offers a find-and-replace substitution. However, in Decaf, it must be preceded by a # to request expansion of the name. 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, and 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 empty (and thus, later use of the name will expand to

nothing).

• A name can be redefined, and the new replacement supersedes the previous from that point onwards.

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. The name of an existing macro must follow any other use of # in the input, anything else is reported as an invalid directive error, and the token is discarded.

Starter files

The starting files for this project are at ctools.umich.edu. 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
- list.h — simple list class for storing a linear collection of elements
- location.h — location structure for the lexical position of a token or symbol
- 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 need to modify dppmain.cc, dpp.l and scanner.l.

You should use our Makefile rather than directly invoking lex and gcc to build the project. The Makefile has targets to build the two separate programs 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 the following:

```
$ dcc < samples/t1.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).

Using lex

You'll find that lex is not the user-friendliest 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 the actual problem. Learning its quirks may take some delving into the manual, a little experimentation, and patience. 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 the 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 parentheses 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 use it in another definition or a pattern; without them it will only match the literal name.

Scanner Implementation

The scanner.l 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 spaces.
- 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 an 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 specialness). The patterns for integers, doubles, and strings will require careful testing to ensure all cases are covered (see man pages for strtol and 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 than 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 scanning at the following character. If a string erroneously contains a newline, report an error and resume scanning 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 scanning at the next token.

Preprocessor Implementation [Optional]

The preprocessor is optional. If you decide not to implement it, use the provided solution/dpp executable instead for debugging. In this case, be sure to modify the Makefile not to try to build a new dpp executable (change line 11 from saying "PREPROCESSOR = dpp" to just "PREPROCESSOR =").

The preprocessor is small enough to write it in straight hand-coded C/C++, but it also works

out nicely in lex. 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 you can implement them either way, depending on your preference. If you decide to use lex, we recommend you first write the scanner to get your lex bearings and return 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 aren't misidentified as comments. If a file ends with an unclosed multi-line comment, an error is reported via a call to our ReportError() function. To match our output exactly, please use the standard error messages provided in errors.h.

The macro definition and replacement are more complex, but your C/C++ skills should be handy for 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 reported as an invalid directive error and discarded.

You will need a hashtable to associate names with replacements. You may reuse 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, or whatever you like. Do not worry about dynamically resizing your hashtable or anything fancy. Any reasonably efficient implementation will do. You should put your hash table implementation in its file. You must modify the Makefile to add the new source to the executable.

Testing

In the starting project, there is a samples/ directory containing various input files and matching .out files representing 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 input files to test your preprocessor and scanner further. What formations look like numbers but aren’t? What sequences might confuse your processing of comments or macros? This is precisely the thought process a compiler writer must go through. Any sort of input is fair game to be fed to a compiler. You’ll want to be sure yours can handle anything that comes it's way, correctly tokenizing it if possible or reporting some reasonable error if not. You may also construct your test cases and generate the expected outputs with the provided solution/dcc executable.

Note that lexical analysis is responsible 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}
```

Grading

This project is worth 12 points, with extra 4 bonus points if you implement the optional preprocessor. Most of the points will be allocated for correctness. We will run your program through the given test files from the samples directory and other tests, using `diff -w` to compare your output to our solution.

Submission

Submit your *.c, *.cc, *.cpp, *.h, *.hpp, *.l to our autograder (https://autograder.io/).