Assignment 2

Fri Feb 5 (125 points)

For this assignment, you will code a lexical analyzer for the PicoScript language. You can use my lexdemo project as an example. I have also provided a template project for you to use in the assn2 project — it defines the `Token` class (which you shouldn't need to change), `LexError` for reporting lexical errors, a `LexerTest` which is full of test cases, and the skeleton of the `Lexer` class.

(If you already cloned the `cs664pub` repository, you can use **VCS » Update Project** or `git pull` from the command line to download the latest changes.)

Like the list language in lexdemo, PicoScript is LL(1). This means we need only one character of look-ahead to decide which token to produce. The pieces in `Lexer` that are missing are marked with `// TODO` comments. The bulk of it is in `nextToken`, but you can also add code to `consume` to update the line and column counters. For most tests, we don't care about tracking the positions of tokens, but there are a few tests that verify positions explicitly. You are welcome to add additional methods, similar to how we used `scanInteger` in the lexdemo lexer.

In both lexdemo and the assn2 template, there is a `main` method in the lexer class that accepts strings on the standard input. Here is a sample run of my solution, where I typed only the line that begins with `/square` – the rest is output by the program.

```
Patiently awaiting your code (^D or ^Z to end)
/square{dup mul}def
SYM(square)@1:1
LBRACE@1:8
OP(dup)@1:9
OP(mul)@1:13
RBRACE@1:16
OP(def)@1:17
EOF
```

To submit your program, make sure the files in your `assn2` directory are added to Git VCS, then commit and push.

Tips

You can just copy the entire `assn2` directory from the project template into your own Git working directory, and use it as a starting point. Either run the `Lexer` class to try it interactively, or run `LexerTest` to apply all the test cases.

To understand a little more concretely what it has to do, let's break down the demonstration block shown above. The PicoScript program `/square{dup mul}def`

is passed to `new Lexer()` as a `StringReader`, so we can read one character at a time using `reader.read()`. (That part is already provided — see the `Lexer.consume` method.)

Then, the main program calls `Lexer.nextToken` repeatedly. Each time, it returns the next token in the program, which produces the list in the example above. So let's examine the first one. The first character in lookahead is the slash '/' from the input string. We know that the slash indicates the start of a **symbol.** So you consume and buffer any remaining characters until you see something that *cannot* be part of an identifier, like a space, parentheses, curly braces, or EOF. And then you can return

```
return new Token(Token.Type.SYM, buffer.toString());
```

as the first token. It's exactly the same procedure as `scanIdentifier` in the lexdemo project.

Now the next time `nextToken` is called, the lookahead character will be the left brace. So you just

```
return new Token(Token.Type.LBRACE);
```

The third time `nextToken` is called, the lookahead character will be the letter 'd' from dup. Any alphabetic character starts an **operator.** Consume and buffer remaining characters until you see something that cannot be part of an identifier (sound familiar? again, just like `scanIdentifier` in the lexdemo project). And the third token will be

```
return new Token(Token.Type.OP, buffer.toString());
```

These three pieces I've stepped through correspond to the first three lines of output:

```
SYM(square)@1:1
LBRACE@1:8
OP(dup)@1:9
```

That output also shows the line and column numbers, but don't worry about that until all the different token types are working. So really at this point your output would just not include the line and column numbers:

```
SYM(square)
LBRACE
OP(dup)
```

Now what is the lookahead character? Following the `dup` in my program is a space. So when the lookahead is a space, you should consume it, ignore it, and check the next character. Now the lookahead is '`m`' which is alphabetic, so will be an operator again.

Because PicoScript is LL(1), one character is enough to determine what token type we'll output:

- Slash indicates a symbol
- Alphabetic $\rightarrow$ operator
- Digit $\rightarrow$ integer
- Left brace
- Right brace
- Percent sign for comment (ignored)
- White space is ignored