Project Part 1: Lexer

New Attempt

Due Feb 8 by 11:59pm **Points** 100 **Submitting** a file upload **File Types** java

In this assignment, you will implement the lexer for our language. This is the first step in the parsing process which takes input source code, represented as a sequence of characters, and turns them into tokens - the building blocks of our grammar - for the parser to use in the next part.

- A grammar for our language will be released following the submission of Homework 1, as it contains solutions for our number and string regexes. For now, textual descriptions are provided instead - these are going to seem more complicated then they really are, but it's good practice and you can (and are encouraged to) ask clarifying questions as needed. Understanding written requirements is a really important skill!

Also, note the Crafting Interpreters link below is *highly* recommend to help you understand both the lexing process as well as the implementation approach we're using, especially peek and match. If anything qualifies as a textbook in this class, it's that.

Submission

You will submit `Lexer.java`, which implements your lexer. The `Token` and `ParseException` classes should be the same, and you do not need to submit any additional tests you've added. You should re-test your lexer with the provided test class prior to submitting.

- There will be one test submission on **Friday, February 5**.
- The final submission is **Monday, February 8**.

Lexer Overview

Recall that the job of the lexer is to combine characters into tokens which can be processed by the parser. For example, the input `1234` has the characters `['1', '2', '3', '4']`, which should produce an `INTEGER` token. More complex input, such as `LET x = 5;`, will produce multiple tokens.

If the lexer is unable to parse something successfully (for example, an unterminated string), then it will throw a `ParseException`. It may seem oddly named for now, but lexing is part of the parsing process and we use the same exception for parsing errors later on.

Crafting Interpreters

The Scanning section of Crafting Interpreters provides a good overview of the lexing process and was a starting point for the lexer architecture below. I highly recommend reading it to help with your understanding of the lexer process and this assignment.

Grammar

A grammar for our lexer is defined below, which is written in a specific form optimal for our approach. You can view a graphical form of our grammar on the following website:

- https://www.bottlecaps.de/rr/ui

```
identifier ::= [A-Za-z_] [A-Za-z0-9_-]*
number ::= [+\-]? [0-9]+ ('.' [0-9]+)?
character ::= ['] ([^'\n\r\\] | escape) [']
string ::= '"' ([^"\n\r\\] | escape)* '"'
escape ::= '\' [bnrt'"\\]
operator ::= [<>!=] '='? | 'any character'

whitespace ::= [ \b\n\r\t]
```

Notice that each rule corresponds to a provided `lex` method. You should ensure that all lexing for a rule is entirely within that rule's lex method - specifically, the `lexToken` method should **never** change the state of the char stream itself; it's only job is to delegate to the proper rule.

Token Types

We will use 6 types of tokens, defined below. You will need to implement lexing for all of these. The `LexerTests` class below includes tests for the provided examples, and you should add additional ones as you see fit.

- `IDENTIFIER`: Represents keywords and names used for variables, functions, etc. Allows alphanumeric characters, underscores, and hyphens (`[A-Za-z0-9_-]`), but cannot start with a digit or a hyphen.
  - Examples: `getName`, `thelegend27`
  - Non-Examples: `-five`, `1fish2fish3fishbluefish` (note these do not *fail*, they're just lexed as something more complex than an identifier)
- `INTEGER`/`DECIMAL`: Numbers, naturally either integers or decimals. As in our Regex homework, all numbers start with an optional sign (`+`/`-`) followed by one or more digits. Decimal numbers are determined by a decimal point `.` **and** one or more digits.
  - Note 2: The **and** requirement means that a decimal point that is not followed by digits is not part of a number. For example `5.toString()` starts with an `INTEGER`.
  - Examples: `1`, `123.456`, `-1.0`
  - Non-Examples: `1.`, `.5` (as above, note these do not *fail*, they're just lexed as something more complex than a number)
- `CHARACTER`: A character literal. Similar to string literals below, however start and end with a single quote (`'`) and must contain one and only one character. Escape characters are also supported starting with a backslash (`\`), which must be followed by one of `bnrt'"\` (and are considered one character). The character cannot be a single quote (`'`), since that ends a character literal, or a line ending (`\n`/`\r`), to avoid character literals spanning multiple lines.
  - Examples: `'c'`, `'\n'`
  - Non-Examples: `''`, `'abc'` (note these should throw ParseExceptions with the index at the missing/invalid character)
- `STRING`: A string literal. As in our Regex homework, strings start and end with a double quote (`"`) and support escape characters starting with a backslash (`\`), which must be followed by one of `bnrt'"\`. Characters cannot be a double quote (`"`), since that ends a string literal, or a line ending (`\n`/`\r`), to avoid string literals spanning multiple lines. This is particularly important for strings, which could cause cascading errors if they covered multiple lines (try an unterminated string vs an unterminated block comment and see what happens).
  - Examples: `""`, `"abc"`, `"Hello,\nWorld!"`
  - Non-Examples: `"unterminated`, `"invalid\escape"` (note these should throw ParseExceptions with the index at the missing/invalid character)
- `OPERATOR`: Any other character, excluding whitespace. Comparison operators (`<=`, `>=`, `!=`, `==`) are a special case and will be combined in to a single token, for all other characters an `OPERATOR` token is only that character.
  - Examples: `(`, `<=`
  - Non-Examples: ` ` (space), `\t` (tab)
- Whitespace characters (`\b\n\r\t`) should be skipped by the lexer and not emitted as tokens. However, they are still meaningful when determining where a token starts/ends (`12` is one `INTEGER` token, but `1 2` is two `INTEGER` tokens).

Examples

- `LET x = 5;`
  - `Token(IDENTIFIER, "LET", 0)`
  - `Token(IDENTIFIER, "x", 4)`
  - `Token(OPERATOR, "=", 6)`
  - `Token(INTEGER, "5", 8)`
  - `Token(OPERATOR, ";", 9)`
- `print("Hello, World!");`
  - `Token(IDENTIFIER, "print", 0)`
  - `Token(OPERATOR, "(", 5)`
  - `Token(STRING, "\"Hello, World!\"", 6)`
  - `Token(OPERATOR, ")", 21)`
  - `Token(OPERATOR, ";", 22)`

Provided Code

The following files are provided to help you help implement the parser. As always, you must implement

- Source Files (`src/main/java/plc/project`)
  - Lexer.java
  - Token.java
  - ParseException.java
- Test Files (`src/test/java/plc/project`)
  - LexerTests.java