Project Checkpoint 1: Lexer

50 Points

2/10/2025

Attempt 1

Review Feedback
2/10/2025

Attempt 1 Score:

View Feedback

Anonymous Grading: **no**

Unlimited Attempts Allowed
1/29/2025 to 2/13/2025

Details

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.

Submission

You will submit a Zip file of the entire `src` folder (containing just the single `src` folder), which should contain both your implementation (`src/main/java`) and tests (`src/test/java`). On Windows, right-click on the src folder and "Compress to ZIP file" should have the expected behavior here.

- The test submission is **Thursday, February 6**: **Project Checkpoint 1: Lexer (TEST) (https://ufl.instructure.com/courses/523541/assignments/6419670)** .
- The final submission is **Monday, February 10**.

Project Setup

See the following links for provided code and setup instructions:

- **PlcProject.zip (https://ufl.instructure.com/courses/523541/files/94402171?wrap=1)** **(https://ufl.instructure.com/courses/523541/files/94402171/download?download_frd=1)**
- **Project Setup (IntelliJ & Gradle) (https://ufl.instructure.com/courses/523541/pages/project-setup-intellij-and-gradle)**

Lexer Overview

The Lexer is a common preliminary step in the parsing process that breaks up the source input into **tokens**. These tokens group together categories of inputs, such as integers, that can then be used by the parser without having to deal with individual characters. For example, the input `1234` has four characters - `['1', '2', '3', '4']` - that represent a single `INTEGER` token. More complex input,

such as `LET x = 5`, will produce multiple tokens of varying types to be handled by the Parser later on.

Crafting Interpreters

The **Scanning (https://www.craftinginterpreters.com/scanning.html)** section of Crafting Interpreters provides a good overview of the lexing process. The architecture used is similar to our project, but not strictly identical. 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 (https://www.bottlecaps.de/rr/ui)**: Paste into "Edit Grammar", then "View Diagram".
  - **Important**: Escape characters (e.g. `\n` for newline) aren't supported by the tool and will display weirdly.

```
tokens ::= (skipped* token)* skipped*

//these rules do not emit tokens; input is skipped by the lexer
skipped := whitespace | comment
whitespace ::= [ \b\n\r\t]+
comment ::= '/' '/' [^\n\r]*

//these rules do emit tokens
token ::= identifier | number | character | string | operator
identifier ::= [A-Za-z_] [A-Za-z0-9_-]*
number ::= [+-]? [0-9]+ ('.' [0-9]+)? ('e' [+-]? [0-9]+)?
character ::= ['] ([^'\n\r\\] | escape) [']
string ::= '"' ([^"\n\r\\] | escape)* '"'
escape ::= '\' [bnrt'"\]
operator ::= [<>!=] '='? | [^A-Za-z_0-9'" \b\n\r\t]
```

Notice that each rule corresponds to one of the provided `lex` methods. 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. Maintaining this separation of concerns will reduce the likelihood of bugs in your lexer.

Token Types

The following table lists all token types and example inputs. **Pay particularly attention to error behavior**: A grammar can often be ambiguous whether input should be considered an error (and if so where) or simply lexed/parsed in a different way. For example:

- `1.` could be considered an invalid decimal, but could also be lexed as the tokens `INTEGER 1`

and `OPERATOR .`. Our language will use the later to allow inputs like `1.toString()`.
- `'char` could be considered an invalid character literal, but could also be lexed as the tokens `OPERATOR '` (if `'` were allowed to be an operator by our grammar) and `IDENTIFIER char`. Our language will use the former to provide a more descriptive error message to uses.

If an error should be thrown, this should use the provided `LexException` class. This class takes a `String message` - this should be a descriptive message to help identify the issue on your side; tests will not use this message.

Token Type	Description	Examples
`IDENTIFIER`	An identifier in our language, such as a variable name or keyword.	• `getName` • `iso8601` • `-five`: not identifier; leading hyphen • `1fish2fish`: not identifier; leading digit
`INTEGER`	An integer literal. Note that this token type is handled by `lexNumber`, and the presence of a decimal part or not determines the token type.	• `1` • `123` • `1e10` • `1e`: not integer; missing exponent digits (but contains an integer)
`DECIMAL`	A decimal literal. The presence of `.`/`e` does not "commit" there to being a decimal or exponent value, and thus these are not considered errors. Note that this token type is handled by `lexNumber`, and the presence of a decimal part or not determines the token type.	• `1.0` • `123.456` • `1.0e10` • `1.`: not decimal; missing decimal digits
`CHARACTER`	A character literal, which supports escapes. Once a character literal begins with `'`, any non-matches to the grammar is an error (e.g. an unterminated literal or invalid escape). There is no other valid use of `'` in our grammar.	• `'c'` • `'\n'` • `'u`: error; unterminated • `'abc'`: error; too many characters
`STRING`	A string literal, which supports escapes. Once a string literal begins with `"`, any non-matches to the grammar is an error (e.g. an unterminated literal or	• `""` • `"string"` • `"newline\nescape"`

	invalid escape). There is no other valid use of `"` in our grammar.	• `"invalid\escape"`: error; invalid escape
`OPERATOR`	An operator character. This is a "catchall" token type for all characters not covered by the above rules.	• `+` • `<=` • `)`: not operator; empty • `"`: not operator; unterminated string

Changelog

2/3	• Added productions for redacted rules (number/character/string/escape). • Clarified that whitespace/comments are skipped by the lexer and do not emit tokens. • Clarified that `1e`, while not itself an integer, does *contain* an integer. • Fixed spec incorrectly saying `LexException` takes a `Token.Type` argument.
2/9	• Updated grammar to include top-level tokens rule (for multi-token lexing) and use precise characters for operators. • Clarified that non-matches in character/string literals are considered errors (as now clear from the grammar).
2/11	• Adding missing sign to exponents (`(e [+-]? [0-9]+)?`) to match the RegexTesting solution as originally intended. ◦ We will NOT grade these cases for this assignment, but may on the resubmission at the end of the semester.

Test Cases

Note: Unprintable characters, such as newlines, may be represented with printable **Unicode control symbols (https://www.compart.com/en/unicode/block/U+2400)** (`␤`) or as a Java string (`"\n"`). The input `[ ␉␍␊␍␤]`, for example, is the Java string `"[ \b\n\r\t]"`.

▸ Lexer - Final (200):