Project Checkpoint 1: Lexer

50 Points Possible

2/11/2026

Attempt 1

In Progress

NEXT UP: Submit Assignment

Add Comment

Unlimited Attempts Allowed

1/28/2026 to 2/14/2026

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 groups these characters together into tokens for use in the parser. These tokens form the foundational building blocks of our language's grammar.

Setup/Submission

- Setup: **PlcProject (Lexer).zip (https://ufl.instructure.com/courses/555493/files/103969638?wrap=1)** **(https://ufl.instructure.com/courses/555493/files/103969638/download?download_frd=1)** contains the provided project structure and source files. See **Project Setup (IntelliJ/Gradle) (https://ufl.instructure.com/courses/555493/pages/project-setup-intellij-slash-gradle)** for details on setting up the project (and IntelliJ) correctly.
  - **This is a new project, hence you will need to reconfigure these settings again.**
- Submission: You will submit a zip of **only** the `src` folder, which includes both sources and tests. Using `Right Click src` + `Compress to Zip` should have the expected behavior, or alternatively `zip -r submission.zip src` from the command line. The name of the file is irrelevant.
  - The test submission is **Sunday, February 8**: **Project Checkpoint 1: Lexer (TEST) (https://ufl.instructure.com/courses/555493/assignments/6905751)**
  - The final submission is **Wednesday, February 11**.

Lexer Overview

Recall that lexing 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 `1234`. More complex input, such as `LET name = 1234`, will produce multiple tokens like `[LET, name, =, 1234]`. Note that whitespace is not included in tokens, but rather plays a role in separating them: `LET name` is `[LET, name]` while `LETname` is `[LETname]`.

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.

Architecture Overview

The provided code contains three files for the lexer:

- `Lexer.java`, containing the `Lexer` class with `lex` methods to implement and a `CharStream` class that helps manage the use of input characters and making lexing decisions. See Crafting Interpreters for a similar (but not identical) approach introducing `peek` vs `match`.
- `Token.java`, containing the `Token` record and `Token.Type` enum.
- `LexException.java`, containing the `LexException` exception used for lexer errors.

Additionally, a `Main.java` file implementing a basic REPL for interacting with the lexer has been provided. You should review this file and be comfortable using it to test input as it will help immensely for manually testing later parts of the project.

Provided APIs & Compatibility

Additionally, it is your responsible to make the necessary changes to implement the project successfully without breaking the API compatibility of the provided code. Watch for public vs private visibility - public methods are part of the public API and thus should not be changed, while private methods are part of your implementation and can be modified more freely (e.g., adding exceptions to necessary methods...). That said, you should make an effort to stick to the provided architecture to better match lecture content and avoid pitfalls you may not be aware of that we have intentionally design around.

Grammar

An EBNF 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.

Ambiguity & Errors

A grammar can often be ambiguous on how specific input should be lexed/parsed or whether it should be considered an error (and if so where). Here are some concepts and approaches to addressing this:

- **Lexing/Parsing Grammars** are often written in a way to reduce ambiguity. Our grammar uses a single `number` rule even though we have separate `INTEGER` / `DECIMAL` tokens because they both start with an integer. This better represents the control flow needed to implement the corresponding lexing/parsing.
- **Maximal Munch ("Greedy" Lexing)** refers to prioritizing the longest valid input for a single token. For example, `1.5` is a single `DECIMAL` and not three tokens `[1, ., 5]` because we give priority to extending the initial integer into a complete decimal. This is how separate `integer` / `decimal` rules in the grammar would be handled.
- **Precedence** refers to prioritizing certain rules over others (typically via order) to indicate a single valid derivation. This is primarily used as the parser level rather than the lexer, but as a hypothetical example the `operator` rule's `[^A-Za-z_0-9'" \b\n\r\t]` simply represents "any input character that wouldn't have been handled by a rule above", effectively an "else" catchall.
- **Committing & Eager Errors** refers to the idea of determining when a rule in the grammar is expected to match and, if it does not, quickly erroring at the point of failure to provide a more descriptive error message. For example, `1.` is two tokens `[1, .]` rather than an error because we do *not* commit to being a decimal. However, `"unterminated` is an error because we *do* commit to being a string. The behavior here typically balances whether the input is valid using other grammar rules and what the most likely intention of the user is.

If an error should be thrown, you must use the provided `LexException` class with the correct character `index` at the point of failure (which should almost always be `chars.index`). We will check for both the `LexException` and `index` in tests, however the `String`

`message` is simply a descriptive message for your own debugging and will not be used in tests.

Rules / Tokens

The following table lists all rules, a description of token/error behavior beyond the provided grammar, and example inputs. As above, **pay particular attention to errors** and how potentially-ambiguous inputs should be lexed, especially with multiple token types.

Rule / Token	Description	Examples
`whitespace`	Whitespace characters that are skipped by the lexer.	• `" "` (Java string) • `"\n"` (Java string) • `"  \n  "` (Java string)
`comment`	A textual comment that is skipped by the lexer.	• `//` • `//comment`
`identifier` `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
`number` `INTEGER`	An integer literal, one part of `number`. The absence of a decimal part determines the type. The presence of `e` does not "commit" there to be a decimal/exponent part, and thus these are not errors.	• `1` • `123` • `1e10` • `1e`: not integer; missing exponent digits
`number` `DECIMAL`	A decimal literal, one part of `number`. The presence of a decimal part determines the type. The presence of `.` / `e` does not "commit" there to be a decimal/exponent part, and thus these are not errors.	• `1.0` • `123.456` • `1.0e10` • `1.`: not decimal; missing decimal digits
`character` `CHARACTER`	A character literal, which supports escapes. The initial `'` "commits" to a character literal (as there are no other valid uses in our grammar) and thus any subsequent non-matches should be treated as an error.	• `'c'` • `'\n'` • `'u`: error; unterminated • `'abc'`: error; too many characters
`string` `STRING`	A string literal, which supports escapes. The initial `"` "commits" to a string literal (as there are no other valid uses in our grammar) and thus any subsequent non-matches should be treated as an error.	• `""` • `"string"` • `"newline\nescape"` • `"invalid\escape"`: error; invalid escape
`escape`	An escape character, shared by character/string literals.	See character/string.
`operator` `OPERATOR`	An operator character. This is effectively a "catchall" token type for all characters not covered by the above rules.	• `+` • `<=` • ` `: not operator; empty • `"`: not operator; unterminated string

Changelog

2/3	• Added definitions for redacted number/character/string/escape rules.

Choose a submission type