SCANNER

Prepared by Rene Andre B. Jocsing August 31, 2025

For this laboratory activity, your task is to build a **lexical scanner**. A programming language needs three components, at the very least, so that your machine can run it: a scanner, a parser, and an interpreter. Of course, like in most things, we will begin with the first one.

Source code is typically stored in text files. A computer machine needs some way to transform this source code into machine code. The scanner is our first stop in the pipeline. It takes some text and attempts to turn the letters and numbers in it into **tokens** that are relevant to some programming language. You can think of a scanner as the "frontend" of our source-to-machine code pipeline.

Tokens

Before we can build a scanner, we must first decide what to scan.

Consider the following line of source code from Kotlin:

```
1    var myVariable = 4
```

This string can be broken up into a countless number of permutations. You can form substrings like “my”, “Vari”, or “able = 4”, but none of these fragments have any meaning in the context of Kotlin. However, if we break the string into meaningful chunks, we get something different:

```
1    var | myVariable | = | 4
```

Each of these chunks is called a **lexeme**. A lexeme is the smallest sequence of characters that still represents something meaningful in the language’s grammar. In our example, `var` is a keyword for declaring variables, `myVariable` is an identifier, `=` is an assignment operator, and `4` is a number literal.

The process of grouping characters into lexemes is called **lexical analysis** or **tokenization**. Our scanner performs this analysis by examining the raw source code character by character and determining where one lexeme ends and the next one begins.

From Lexemes to Tokens

While lexemes are the raw substrings from our source code, we need additional information to make them useful for the later phases of our programming language implementation. This is where **tokens** come in. A token bundles a lexeme together with other useful data.

1. **Token Type**: This categorizes what kind of lexeme we have. Is it a keyword like `var`? An operator like `=`? A literal value like `4`? Later, the parser would need to know not just the raw text, but what category of language construct it represents.
2. **Literal Value**: For tokens that represent values (like numbers and strings), we convert the textual representation into the actual runtime value that our interpreter will use later.
3. **Location**: To provide meaningful error messages, we track where each token appears in the source code, typically recording at least the line number.

For our Kotlin example, the tokens might look like this:

- Token: Type=KEYWORD_VAR, Lexeme="var", Literal=null, Line=1
- Token: Type=IDENTIFIER, Lexeme="myVariable", Literal=null, Line=1
- Token: Type=EQUALS, Lexeme="=", Literal=null, Line=1
- Token: Type=NUMBER, Lexeme="4", Literal=4, Line=1

The Scanning Process

The heart of our scanner is a simple loop. Starting from the first character of the source code, the scanner:

1. Examines the current character to determine what kind of lexeme it might be starting
2. Consumes additional characters that belong to the same lexeme
3. When it reaches the end of the lexeme, it creates and emits a token
4. Moves to the next character and repeats the process

This continues until the scanner reaches the end of the input. Some lexemes are easy to recognize—single characters like parentheses and operators. Others require more work, like string literals that continue until a closing quote, or numbers that might include decimal points.

The rules that determine how characters are grouped into lexemes form what we call the **lexical grammar** of our language. For most programming languages, including the one you'll implement, this grammar is simple enough to be classified as a "regular language"—the same mathematical concept that underlies regular expressions.

However, since we're doing this from scratch like the cavemen of the computer science world once did, **you're not allowed to use regular expressions in your implementation**.

Laboratory Deliverables

Lexical Scanner

Implement a complete lexical scanner that can take source code written in a simple programming language and convert it into a stream of tokens. Your scanner will need to handle:

- Single-character tokens (parentheses, operators, punctuation)
- Multi-character operators (like == and <=)
- String literals enclosed in quotes
- Numeric literals (both integers and decimals)
- Identifiers and keywords
- Comments (which should be ignored)
- Whitespace handling and error reporting

From now on, your source code for our incremental labs must live in a GitHub repository. **Your GitHub repository will serve as your submission across all future labs.** Please invite me as a collaborator so I can see and review your code. Remember to write meaningful commits!

Language Specification

At this point, you must start thinking about the design of your programming language. How is a variable declared? What keywords are reserved by the programming language? Is whitespace significant? Or is it bracketed like the C family of programming languages? How are comments styled? Are nested or docstring-style comments allowed? How does it do loops and other common language constructs? Why is it this way? What is the motivation behind your choices?

You will document this information at the landing page of your GitHub repository. To do so, **edit your README.md** and fill in this template. The only hard and fast rules that you must adhere to when designing your programming language are: **the language must be dynamically typed** and **the prototype of the language must be feasible in the span of three months.**

Don't worry about making mistakes at this stage. Modifying your scanner later down the line if you change your mind about a design decision is easy, **as long as you understand the code that you've written**. In fact, shifting requirements and specifications are common in large, long-term software engineering projects. As developers, our task is to understand source code so we can scale, debug, and maintain it in the future (this is why vibe coders will never replace software engineers!).

The expected, testable output for this laboratory activity is a Read-eval-print loop (REPL) in the commandline. This has your scanner implementation at its heart. The scanner must also provide some form of error handling for invalid or malformed lexemes. Refer to the following.

```
> This is the ktlox programming language
Token(type=IDENTIFIER, lexeme=This, literal=null, line=1)
Token(type=IDENTIFIER, lexeme=is, literal=null, line=1)
Token(type=IDENTIFIER, lexeme=the, literal=null, line=1)
Token(type=IDENTIFIER, lexeme=ktlox, literal=null, line=1)
Token(type=IDENTIFIER, lexeme=programming, literal=null, line=1)
Token(type=IDENTIFIER, lexeme=language, literal=null, line=1)
Token(type=EOF, lexeme=, literal=null, line=1)
> var someString = "I am scanning"
Token(type=VAR, lexeme=var, literal=null, line=1)
Token(type=IDENTIFIER, lexeme=someString, literal=null, line=1)
Token(type=EQUAL, lexeme==, literal=null, line=1)
Token(type=STRING, lexeme="I am scanning", literal=I am scanning, line=1)
Token(type=EOF, lexeme=, literal=null, line=1)
> var someNumber = 3.1415 + 6.9420
Token(type=VAR, lexeme=var, literal=null, line=1)
Token(type=IDENTIFIER, lexeme=someNumber, literal=null, line=1)
Token(type=EQUAL, lexeme==, literal=null, line=1)
Token(type=NUMBER, lexeme=3.1415, literal=3.1415, line=1)
Token(type=PLUS, lexeme=+, literal=null, line=1)
Token(type=NUMBER, lexeme=6.9420, literal=6.942, line=1)
Token(type=EOF, lexeme=, literal=null, line=1)
> // This is a comment, so it is ignored
Token(type=EOF, lexeme=, literal=null, line=1)
> var parenthesizedEquation = (1 + 3) - (2 * 5)
Token(type=VAR, lexeme=var, literal=null, line=1)
Token(type=IDENTIFIER, lexeme=parenthesizedEquation, literal=null, line=1)
Token(type=EQUAL, lexeme==, literal=null, line=1)
Token(type=LEFT_PAREN, lexeme=(, literal=null, line=1)
Token(type=NUMBER, lexeme=1, literal=1.0, line=1)
Token(type=PLUS, lexeme=+, literal=null, line=1)
Token(type=NUMBER, lexeme=3, literal=3.0, line=1)
Token(type=RIGHT_PAREN, lexeme=), literal=null, line=1)
Token(type=MINUS, lexeme=-, literal=null, line=1)
Token(type=LEFT_PAREN, lexeme=(, literal=null, line=1)
Token(type=NUMBER, lexeme=2, literal=2.0, line=1)
Token(type=STAR, lexeme=*, literal=null, line=1)
Token(type=NUMBER, lexeme=5, literal=5.0, line=1)
Token(type=RIGHT_PAREN, lexeme=), literal=null, line=1)
Token(type=EOF, lexeme=, literal=null, line=1)
> /* This is a C-style docstring */
Token(type=EOF, lexeme=, literal=null, line=1)
>
```

Rubric for Programming Exercises

Table 1: Rubric for Programming Exercises (50 pts)

Criteria (10 Points Each)	Excellent (9 – 10)	Good (6 – 8)	Fair (3 – 5)	Poor (0 – 2)
Program Correctness	Program executes correctly with no syntax or runtime errors, meets/exceeds specifications, and displays correct output	Program executes and outputs with minor errors, yet meets specifications	Program executes and outputs with major errors, yet somehow meets specifications	Program does not execute or does not meet specs
Logical Design	Program is logically well-designed with excellent structure and flow	Program has slight logic errors that do not significantly affect the results	Program has significant logic errors affecting functionality	Program logic is fundamentally incorrect
Code Mastery	Programmer demonstrates excellent mastery over the program's code	Programmer demonstrates adequate mastery over the program's code	Programmer demonstrates fair mastery over the program's code	Programmer demonstrates poor mastery over the program's code
Engineering Standards	Program is stylistically well designed from an engineering standpoint	Slight inappropriate design choices (i.e., poor variable names, improper indentation)	Severe inappropriate design choices (i.e., code repetition, redundancy)	Program is poorly written
Documentation*	Program is well-documented: comments exist for clarity, not redundancy	Missing one required comment or some redundant comments	Missing two or more required comments or many redundant comments	Most documentation missing or most documentation is redundant

Remember:* *"Code tells you how, comments tell you why."*** *— Jeff Atwood, co-founder of Stack Overflow and Discourse*