# Exercise 01: A Handwritten Scanner

Compiler Construction (Fall '25)

Discussion: Sept, 26th

1. **Regular Expressions**
   Declare a regular expression which describes exactly the following languages:
   (a) the set of all strings $w \in \{a, b\}$ where each $a$ is stated after at minimum two $b$'s.
   (b) the set of Java `DecimalIntegerLiterals`. They are either `0` or one or more digis without a `0` in the beginning, and optionally terminated with `L` or `l`.

2. **Implementing a Scanner for SPL**
   In the course of this lecture and exercise series, we implement the interpreter for a simple programming language. The language is designed to be handy for our small Compiler Construction Course but as expressive as to be Turing-complete.

   In fact, the language does not need much concepts to be Turing-complete: It needs to be able to perform arithmetic operations, simple control flows, and to allocate memory. For the moment, we choose a structured programming style and call the language *SPL* (SPL, because it is a Structured Programming Language).

   SPL is based on the Lox language used by Robert Nystrom in his book "Crafting Interpreters", with some modifications and simplifications. Particularly, for the moment, we do neither include functions nor classes or objects.

## The key SPL concepts

Let's get familiar with SPL. Here, is what you need to know for now.

### Data Types

We assume SPL to be dynamically typed, and we support Boolean, Number, and String as data types. The following code snippet demonstrates the types available in SPL.

```
var a = true;  // A boolean
var b = 123;   // A number
var c = 12.3;  // Another number
var d = "123";    // This is a string, not a number
```

### Expressions

SPL features basic arithmetic, comparison, and logical operators which you may know, for instance, from Java. The only lexical difference is that we use the keywords `and` and `or` to represent the basic Boolean operators. We do not support bitwise operators so there is no need for a cryptic syntax.

```
i + d; // 135.3
1 == 2; // false
!true; // false
true || false; // true
var average = (min + max) / 2;
```

### Statements

We saw now some kinds of statements: Simple expression statements and variable declarations. Additionally, SPL has a special `print`-statement to communicate with the world. Furthermore, SPL can group statements in a block(-statement):

```
{
  print "Hello, world!";
  print "Hello, SPL Prime world!";
}
```

### Control Flow

For expressing control flow, SPL features `if`- and the `while`-statements.

```
if (condition) {
  print "yes";
} else {
  print "no";
}
```

```
var a = 1;
while (a < 10) {
  print a;
  a = a + 1;
}
```

## Lexical Specification of SPL

In summary, the lexical specification of SPL comes with the following token types:

- **Operators** One or two consecutive characters that match: `+`, `-`, `*`, `/`, `=`, `==`, `!=`, `>`, `<`, `>=`, `<=`, `&&`, `||`
- **Special characters** For instance parentheses: `;`, `(`, `)`, `{`, `}`
- **Keywords** Exact matches of: `true`, `false`, `var`, `print`, `if`, `else`, `while`
- **Strings** Double quotation followed by none or more characters and a double quotation; e.g., `"hello, world!"` and `""`
- **Numbers** One or more numeric characters followed by an optional decimal point and one or more numeric characters. Such as: `15` and `3.14`
- **Identifiers** An alphabetical character followed by zero or more alphanumeric characters; e.g., `x` and `x1`

**Note:**

- Different kinds of white spaces as well as line comments (starting with `//`) are not specified as tokens. The scanner shall consume them without passing them to the parser (i.e., ignore them).
- Conversely, though not strictly needed, it is common practice and good style to report a special end-of-file token (EOF) to the parser once we have reached the end of an input program.

(a) Specify regular expressions for Strings, Numbers and Identifiers used in SPL. Thus, we can get rid of potential ambiguities which are inherent to natural language specifications. This is a theoretic task, as it is not strictly required for the following implementation task.

(b) Implement a scanner for SPL in the non-naive way presented in the lecture. Essentially, our scanner receives an input String and returns a list of tokens. A `Token` shall comprise its `type`, `lexeme`, `literal`, and `line` number and a customized String method:

```java
public class Token {
  public final TokenType type;
  public final String lexeme;
  public final Object literal;
  public final int line;

  public String toString() {
    return "<" + type + "," + lexeme + "> " + "Literal: " + literal + ",
    Line: " + line;
  }
}
```

**Notes:**

- We use publicly visible fields in our `Token` class which is not a good object-oriented style. However, the tokens properties are immutable, so the object state cannot be manipulated.
- You can ignore the `literal` information for now. It becomes relevant when implementing the interpreter.
- You may use the SPL code snippets above to test the implementation of your scanner.
- A Java code template is provided along with the exercise sheet on ITSLearning. It offers boilerplate code to read SPL programs from an external source file.