Lexer Design Report

1. Introduction

Authors: Janri, Megan
Course: COS 341 – Semester Project

The **lexer (tokenizer)** is the **first phase of the compiler** for the SPL language. Its role is to:

1. Read the raw source code.
2. Discard unnecessary characters such as whitespace and comments.
3. Convert the remaining input into a **sequential stream of tokens**.

This token stream serves as input for the parser, which builds the Abstract Syntax Tree (AST) in the next compiler phase.

2. Master Token List

An **exhaustive list of tokens** was defined to cover all grammar symbols of SPL:

- **Keywords** (reserved words):
  `glob`, `proc`, `func`, `main`, `var`, `local`, `return`, `halt`, `print`,
  `while`, `do`, `until`, `if`, `else`, `neg`, `not`, `eq`, `or`, `and`,
  `plus`, `minus`, `mult`, `div`, `fdef`, `pdef`, `algo`.

- **Operators**:
  Assignment `=`, comparison `>`.
  Arithmetic operations are represented by keyword tokens (`plus`, `minus`, `mult`, `div`) instead of symbols, consistent with SPL's grammar.

- **Delimiters / Punctuation**:
  Braces `{ }`, parentheses `( )`, semicolon `;`.

- **Identifiers**:
  User-defined names matching `[a-z][a-z]*[0-9]*`, beginning with a lowercase letter, optionally followed by more lowercase letters and digits. They cannot conflict with keywords.

- **Literals**:
  - **Numbers**: Integers, either `0` or a non-zero digit followed by more digits (`(0|[1-9][0-9]*)`).

  - **Strings**: Double-quoted sequences of lowercase letters and digits, length up to 15 (`"[a-z0-9]{0,15}"`).

✅ This list covers **keywords, operators, identifiers, numbers, strings, and delimiters** exactly as required.

3. Regular Expressions (Regex) for Each Token

Every token type is defined using a regex to ensure precise recognition:

- **Identifier**: `[a-z][a-z]*[0-9]*`
- **Number**: `(0|[1-9][0-9]*)`
- **String**: `"[a-z0-9]{0,15}"`
- **Operators**: `=`, `>`
- **Whitespace**: `\s+` (skipped, not tokenized)
- **Comments**: `//.*(?:\n|$)` (skipped, not tokenized)
- **Keywords**: Initially matched as identifiers, then upgraded to `KEYWORD` if found in the reserved word list.

✅ This matches the **requirement that each token type must have a precise regex**.

4. Whitespace and Comment Handling

The lexer ensures **no token contains blank spaces**:

- Whitespace characters (spaces, tabs, newlines) are matched by `\s+` and discarded.
- Comments, defined as `//` followed by any characters until end-of-line, are skipped and not turned into tokens.

✅ This satisfies the requirement to **filter out whitespace and comments completely**.

5. Token Stream Creation

The lexer produces a **sequential token stream** where each token has:

- **Type**: e.g., `KEYWORD`, `IDENTIFIER`, `NUMBER`, `STRING`, `LBRACE`, `EQUALS`.
- **Value**: For identifiers (`Token(IDENTIFIER, 'x')`), numbers (`Token(NUMBER, '42')`), and strings (`Token(STRING, '"hello123"')`). Keywords also store their lexeme. Symbol tokens such as `{` or `;` do not store values.
- **EOF token**: Marks the end of input.

Example input:

```
main { var { w } print ( x plus y ) }
```

Token stream output:

```
Token(KEYWORD, 'main')
Token(LBRACE)
Token(KEYWORD, 'var')
Token(LBRACE)
Token(IDENTIFIER, 'w')
Token(RBRACE)
Token(KEYWORD, 'print')
Token(LPAREN)
Token(IDENTIFIER, 'x')
Token(KEYWORD, 'plus')
Token(IDENTIFIER, 'y')
Token(RPAREN)
Token(RBRACE)
Token(EOF)
```

✅ This confirms that the lexer **produces a clean, sequential token stream** with types and values.

6. Conclusion

The lexer design **fully addresses the project requirements**:

- ✔ **Master Token List**: Exhaustive, covering keywords, operators, delimiters, identifiers, numbers, and strings.
- ✔ **Regex Definitions**: Every token type has a precise regex.
- ✔ **Whitespace and Comments**: Both are skipped, ensuring no token contains spaces.

- ✔ **Token Stream**: Sequential, typed, includes values for identifiers/literals, ends with `EOF`.

Additionally, the implementation includes a **warning mechanism for unrecognized input**, improving robustness. This design provides a solid foundation for the parser in the SPL compiler.