Department of Computer Engineering
University of Peradeniya
CO523 - Programming Languages

Lab 03: Implement simple Lexical Analyzer

Objectives

- Understand the role of a **Lexical Analyzer** in the compiler pipeline.
- Apply Regular Expressions (Regex) to define language patterns.
- Implement a tokenization loop using Python's `re` module.
- Manage pattern priority to correctly distinguish between keywords, identifiers, and operators.

1. Lexical Analysis

Lexical analysis, also known as scanning, is the first phase of a compiler. In this phase, the compiler reads the source program character by character from left to right and groups them into meaningful units called **tokens**. These tokens serve as the input for the next phase of compilation, known as syntax analysis.

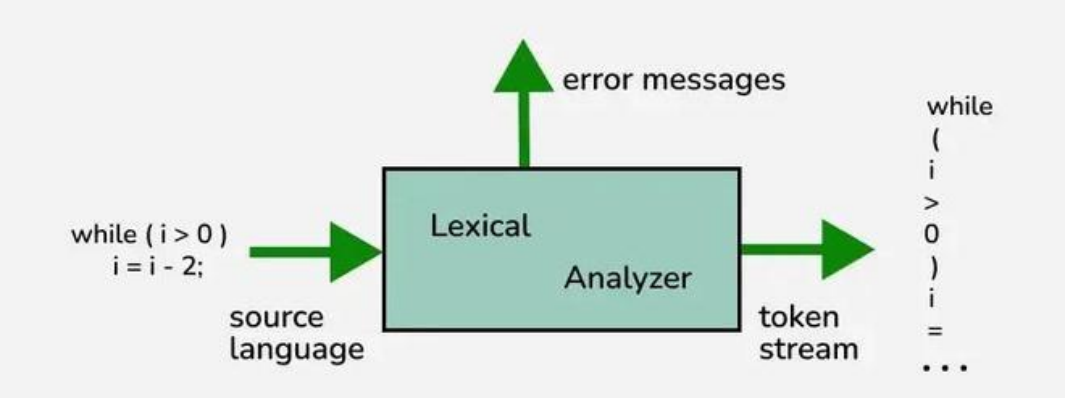

Figure 1: Lexical Analysis

1.1. Lexeme

The actual sequence of characters in the source code that matches a pattern (e.g., 12.5, count, if).

1.2. Token

The abstract category for a lexeme (e.g., FLOAT, ID, KEYWORD).

Tokens in a programming language can be described using regular expressions.

2. Lexical Analyzer

A Lexical Analyzer, also known as a scanner, is responsible for reading the source code character by character and converting it into meaningful tokens. These tokens are then used by the next stage of the compiler for further processing.

e.g:

Consider the following line of code in C:

```
int num = 10;
```

The Lexical Analyzer breaks this into tokens such as:

- int → Keyword
- num → Identifier (variable name)
- = → Operator
- 10 → Numeric Constant
- ; → Special Symbol

Lexical Analyzer helps simplify and structure the input for the next phase of compilation. It also removes whitespace, comments, and detects basic syntax errors such as invalid characters.

3. Simple Lexical Analyzer Implementation

The following Python program implements a basic Lexical Analyzer that can recognize numbers, identifiers, operators, and statement terminators.

```python
import re

def simple_lexer(code):
    # 1. Define the Token Specification
    token_specification = [
        ('NUMBER',   r'\d+'),       # Integer
        ('ID',       r'[A-Za-z]+'),    # Identifiers
        ('OPERATOR',       r'[+=]'),      # Operators (only + and =)
        ('END',      r';'),         # Statement Terminator
        ('SKIP',     r'[ \t]+'),    # Spaces and Tabs
        ('UNKNOWN',  r'.'),         # Any other character (error catch)
    ]

    # 2. Compile patterns into one master Regular Expression
    tok_regex = '|'.join(f'(?P<{name}>{pattern})'
        for name, pattern in token_specification)

    print(f"{'TOKEN TYPE':<15} | {'VALUE'}")
    print("-" * 30)
```

```
    # 3. The Scanning Loop
    for match in re.finditer(tok_regex, code):
        kind = match.lastgroup
        value = match.group()

        if kind == 'SKIP':
            continue
        elif kind == 'UNKNOWN':
            print(f"Error: Unexpected char '{value}'")
        else:
            print(f"{kind:<15} | {value}")

# Test Case
code_sample = "count = 10 + 5;"
simple_lexer(code_sample)
```

Run the program and observe the output for: `count = 10 + 5;`

Notice how the code is broken into tokens such as:

- `ID`
- `NUMBER`
- `OPERATOR`
- `END`

3.1 Understanding the Regular Expressions

Here are the main regex patterns used:

Pattern	What it matches
`\d+`	One or more digits
`[A-Za-z]+`	Words made of uppercase and lowercase letters
`[+=]`	`+` or `=`
`;`	Semicolon
`[ \t]+`	Spaces and tabs
`.`	Any single character

3.2 The Token Specification

The `token_specification` is a list of tuples. Each tuple contains a Label and a Pattern.

Important: The order matters! The Lexical Analyzer checks patterns from top to bottom. If you have two patterns that could match the same text, the one higher in the list wins.

Lab Task

Your task is to modify the given python implementation, `simple_lexer` to handle a subset of the C programming language.

Your code must support the following:

• Preprocessor directives: `#include`, `#define`
• C keywords: `int`, `float`, `char`, `if`, `else`, `while`, `for`, `return`, `void`, `include`, `double`
• Built-in functions: `printf`, `scanf`, `main`, `exit`, `sqrt`
• String literals: `"Hello World"`
• Floating-point numbers: `3.14`, `10.5`
• Integer numbers: `10`, `250`
• Single-line comments: `// this is a comment`
• Comparison operators: `==`, `!=`, `<=`, `>=`
• Assignment operator: `=`
• Statement terminator: `;`
• Dot notation: `.` (eg: in `stdio.h` identify `stdio` and `h` as Identifiers for simplicity and . as DOT)
• Identifiers: letters or _ followed by letters, digits, or _ (e.g., `count`, `total_sum1`)
• Arithmetic and relational operators: `+`, `-`, `*`, `/`, `%`, `<`, `>`
• Curly braces: `{`, `}`
• Parentheses: `(`, `)`
• Comma: `,`
• Whitespace: spaces and tabs (ignored by the lexer)
• Any invalid or unexpected character should be reported as an error

Hints:

- A pattern like `@\w+` could match something similar like `@custom`
- `\b(start|stop)\b` Matches `start` or `stop` **but not** `starter` or `stopped`.
  `\b` make sure the word is separate, not part of a longer word.
- `\d+` matches one or more digits.
  Idea: `\d` stands for a digit (0–9) and `+` means one or more repetitions.
- `"[^"]*"` matches everything inside double quotes, except the quote itself.

Expected Output

After successful implementation, your program should behave as follows:

Test Case 1:

```
int a = 5 / 2;
```

Output:

```
TOKEN TYPE      | VALUE
-----------------------------------
KEYWORD         | int
ID              | a
ASSIGN          | =
NUMBER          | 5
OPERATOR        | /
NUMBER          | 2
END             | ;
=======================================
```

Figure 2: Expected output for Test Case 1

Test Case 02:

```
float x = 10.5; if (x >= 10) { x = x + 1; }
```

Output:

```
TOKEN TYPE      | VALUE
-----------------------------------
KEYWORD         | float
ID              | x
ASSIGN          | =
FLOAT           | 10.5
END             | ;
KEYWORD         | if
LPAREN          | (
ID              | x
COMP            | >=
NUMBER          | 10
RPAREN          | )
LBRACE          | {
ID              | x
ASSIGN          | =
ID              | x
OPERATOR        | +
NUMBER          | 1
END             | ;
RBRACE          | }
=======================================
```

Figure 3: Expected output for Test Case 2

Test Case 03:

```c
#include <stdio.h>
            int main() {
                printf("Hello World");
                return 0;
            }
```

Output:

```
TOKEN TYPE      | VALUE
-----------------------------------
PREPROC         | #include
OPERATOR        | <
ID              | stdio
DOT             | .
ID              | h
OPERATOR        | >
KEYWORD         | int
BUILTIN         | main
LPAREN          | (
RPAREN          | )
LBRACE          | {
BUILTIN         | printf
LPAREN          | (
STRING          | "Hello World"
RPAREN          | )
END             | ;
KEYWORD         | return
NUMBER          | 0
END             | ;
RBRACE          | }
=======================================
```

Figure 4: Expected output for Test Case 03

Submission

Rename your Python script to CO523_Lab03_EXXYYY.py, where EXXYYY represents your E-number.

Upload your answer script to the FEeLS by the given deadline.