Programming Paradigms and Pragmatics (CS202)

LAB - 3

Kaleidoscope is a toy programming language designed to introduce the basics of compiler and language design. The language is procedural, supporting functions, conditionals, loops, and mathematical expressions. The purpose of Kaleidoscope is to keep the syntax simple while demonstrating key concepts of a language frontend, including tokenization, parsing, abstract syntax trees (AST), and interpretation or compilation.

Key Features of Kaleidoscope:

1. Minimal Syntax: Only one data type – a 64-bit floating-point number (double in C).
2. No Type Declarations: All values are implicitly of type double.
3. Procedural: Supports function definitions, conditionals (if/then/else), loops (for), and standard operators.
4. Extensibility: The language can be easily extended to include user-defined operators, JIT compilation, and more.

Example Code in Kaleidoscope:

```
# Compute the x th Fibonacci number
def fib(x)
  if x < 3 then
    1
  else
    fib(x-1) + fib(x-2)

# Calculate the 40th Fibonacci number
fib(40)
```

The language focuses on simplicity and teaching the foundations of building a compiler or interpreter. In this tutorial, you will implement parts of Kaleidoscope starting with tokenization, the process of breaking input code into meaningful units called tokens.

For more details, refer to the official LLVM tutorial:
[LLVM Kaleidoscope Tutorial]

1) Implement a Tokenizer

You need to implement the get_tok function. The program should:

1. Read the input code (either line-by-line from the user or from a file).
2. Recognize and categorize the following tokens:
    - def and extern keywords.
    - Identifiers (e.g., fib, x).
    - Numbers (e.g., 1, 3.14).
    - Operators (e.g., +, -, <).
    - Comments (e.g., lines starting with #).
3. Print the token type and associated value (if applicable, such as the name of an identifier or the value of a number).

Input Example:

```
def fib(x)
  if x < 3 then
    1
  else
    fib(x-1) + fib(x-2)
```

2) Extend the Lexer to Handle Error Checking for Numbers

Extend the functionality of the lexer to identify and handle invalid numeric inputs.

Modify the get_tok function to:

a) **Detect invalid numeric formats**, such as:
- Numbers with multiple decimal points (e.g., 1.23.45.67).
- Numbers containing consecutive dots or starting with invalid sequences (e.g., ..1).

b) **Provide error handling**:
- Display a clear error message whenever an invalid numeric input is encountered.

Input Example:

```
def calculate (x)
        x = 1.23.45.6
```

3) Add Support for String Literals and Logical Expressions in the Lexer

Modify the get_tok function to:

- Recognize string literals enclosed in double quotes (" ").
    - String literals can contain alphanumeric characters, spaces, and special characters like !, ?, and ..
    - Report an error for invalid strings, such as unclosed quotes or invalid escape sequences.
- Add support for logical NOT (!) and parentheses (,) for grouping expressions.

- Update the lexer to tokenize expressions containing both strings and logical expressions.
- Test the lexer with a Kaleidoscope program using these constructs.

Input Example:

```
def greet(name)
  if name == "Alice" || name == "Bob" then
    print("Hello, " + name + "!")
  else
    print("Unknown user: " + name)

  if !(name == "Admin") then
    print("Access denied.")
```

4) Add Support for Multi-Character Operators & Compound Assignment

Extend the lexer to support multi-character operators and compound assignment operators like:

- +=, -=, *=, /=, %=
- ==, !=, <=, >=
- &&, ||

Requirements:

1. Modify/Write code to include new token types for these operators.
2. Extend the get_tok() function to correctly identify multi-character operators without confusion.

Input Example:

```
x += 3 * 4 == 12 && y != 10;
```

5) Lexer Extension for Comments

Modify the lexer to:

- Skip single-line comments (starting with // and ending at a newline).
- Skip multi-line comments (starting with /* and ending with */).

Input Example:

```
x += 3 * (4 + 5) /*IGNORE THIS*/
x += 3 * (4 + 6) //IGNORE THIS TOO
```