Design and Implementation of a C-like Language Interpreter

1. Introduction

An **interpreter** is a software tool that reads and directly executes instructions written in a programming language without first converting them into machine code. It parses the source code line by line and performs actions as specified.

Why Use an Interpreter?

- **Portability**: Source code can be run on any system with the interpreter.
- **Ease of Debugging**: Errors are caught and reported during execution, making debugging easier.
- **Educational Value**: Great for learning how programming languages are structured and executed.
- **Rapid Prototyping**: Enables testing and running code quickly without compilation.

---

2. Project Overview

This project implements a **extensible interpreter** for a **minimalistic, imperative, C-like language**. The system is divided into three core components:

- **Lexer**
- **Parser**
- **Interpreter**

It demonstrates how lexing, parsing, and execution are performed in a language environment. This interpreter can be used for educational purposes or as a foundation for building more complex language tools.

---

3. Components and Functionality

3.1 Lexer (Lexical Analysis)

Role

The lexer converts raw source code into a sequence of **tokens**. Tokens represent meaningful elements like keywords (if, else), operators (+, -, &&), identifiers (a, b), and literals (100, -15).

How It Works

- The lexer reads characters one by one from a memory-mapped file (PROT_READ).
- Each token is identified using **token functions**, which:
  - Maintain an internal state.
  - Return one of:
    - STS_ACCEPT: Token is complete and accepted.
    - STS_HUNGRY: Need more characters.
    - STS_REJECT: Token doesn't match.
- The lexer follows a **maximal munch algorithm**, which prefers the longest possible valid token.
- When all token functions return STS_REJECT, the lexer finalizes the last accepted token and resets its state to continue lexing.

Output

- Tokens are printed in **alternating colors**, making boundaries clear for debugging and visualization.

3.2 Parser (Syntax Analysis)

Role

The parser takes the sequence of tokens from the lexer and produces a **parse tree** that represents the syntactic structure of the program.

How It Works

- Implements a **shift-reduce, bottom-up parser**.
- The **parse stack**:
  - **Shift**: Push tokens/non-terminals onto the stack.
  - **Reduce**: Replace a recognized sequence of stack elements with a higher-level non-terminal based on grammar rules.
- Grammar is statically defined as an array of structs. Each struct represents a rule with:
  - Left-hand side (LHS): The resulting non-terminal.
  - Right-hand side (RHS): The sequence that triggers the reduction.
- **Operator precedence** and **control flow constructs** (e.g., if-elif-else) require additional hacks for correct parsing due to the stateless parser.

Output

- The **parse stack** is printed after every shift/reduce step, which helps visualize the parsing process.

3.3 Interpreter (Execution Engine)

Role

The interpreter walks the parse tree and **executes** the program by evaluating expressions and performing operations as per the language semantics.

How It Works

- Evaluates expressions (arithmetic, relational, logical).
- Executes control structures:
  - if, elif, else
  - while, do-while (supported but not demonstrated here)
- Supports:
  - Variable assignment.
  - Array access and modification.
  - print statements (supports printing both strings and evaluated expressions).
- Warnings and runtime messages are output to **stderr** prefixed with warn:.

---

4. Language Features

Your language, designed to be simple but expressive, includes:

Control Flow

- if-elif-else
- while and do-while loops (optional extensions)

Variables and Arrays

- Scalars: a = 10;
- Arrays: arr[0] = 5;

Expressions

- Binary operators: +, -, *, /, %, ==, !=, <, >, <=, >=, &&, ||
- Unary operators: -, +, !
- Ternary operator: Expr ? Expr : Expr

Printing

- print "Hello World";
- print "Value is " a;

Comments

- Single line: // comment
- Block: /* multi-line comment */

---

5. Sample Program and Execution

Input Program (Smallest of Three Numbers)

```
a = 100;

b = 20;

c = -15;


if (a < b && a < c) {

    print "Smallest number is " a;
```

```
} elif (b < a && b < c) {
    print "Smallest number is " b;
} else {
    print "Smallest number is " c;
}
```

Lexing Stage Output

Tokens are printed with clear separation:

a = 100 ;

b = 20 ;

c = -15 ;

if (a < b && a < c) { ... }

Each token is printed in alternating colors (useful for visual debugging).

Parsing Stage Output

Shift/reduce operations printed step by step:

Shift: ^ a

Shift: ^ a =

Shift: ^ a = 100

Red19: ^ a = Atom

Red20: ^ a = Expr

Shift: ^ a = Expr ;

Red05: ^ Assn

Red02: ^ Stmt

...

ACCEPT Unit

Every parser action (shift, reduce, rule applied) is displayed to illustrate how the source program is parsed into an abstract syntax tree.

Execution Stage Output

The program outputs the result of the interpreted code execution:

Smallest number is -15

---

6. Advantages of the Design

- **Hackable and Extensible**: The modular design allows easy addition of new features (e.g., functions, custom types).
- **Educational Value**: Demonstrates core concepts of interpreters (lexing, parsing, execution) with clear outputs.
- **Visualization**: Prints the parsing and execution process, making it easier to debug and understand.

---

7. Implementation Details

- **Language**: C
- **Memory Management**: Uses mmap for reading source files.

- **State Machines**: Each token function in the lexer operates as a mini state machine.
- **Grammar Handling**: Static grammar definition allows fast lookups during reductions.
- **Error Reporting**: Warnings and runtime errors are clearly flagged.

---

8. Conclusion and Future Work

This project demonstrates a functioning **interpreter for a minimalistic, C-like language** with lexer, parser, and execution stages.

Future Enhancements

- **Functions and Recursion**: Support for user-defined functions.
- **Type Checking**: Basic static type checking during parsing.
- **Better Error Handling**: More informative error messages with line/column info.
- **Optimizations**: Improve parsing efficiency and interpreter performance.

---

9. References

- Dragon Book (Compilers: Principles, Techniques, and Tools)
- Crafting Interpreters by Bob Nystrom
- Flex/Bison documentation (for inspiration, not used)

Appendix: Sample Parse Rules (Excerpt)

Rule ID	Production
01	Unit → Stmt Stmt Stmt $
02	Stmt → Stmt Stmt
03	Stmt → Ctrl
05	Assn → Name = Expr ;
08	Prnt → print "Str" Expr ;
13	Cond → if Pexp { Stmt }
14	Elif → elif Pexp { Stmt }
15	Else → else { Stmt }