Implementation of Domain-Specific Language Using Rust

Ali Dikme
Mehmet Reşit Çağan

Sunday 26th May, 2024

Abstract

This project report details the implementation of a domain-specific language (DSL) using the Rust programming language. The project involves the development of a Rust-based interpreter for a simplified programming language. The interpreter, constructed step-by-step, demonstrates the fundamental processes of lexical analysis, parsing, and code execution using abstract syntax trees (ASTs). From tokenization to the handling of variables and functions, the report illustrates how Rust's language features empower the creation of a robust interpreter. Through practical examples and insights, this project illuminates the journey from language specification to a functional interpreter, offering valuable lessons in systems programming and language implementation.

1 Introduction

In our previous report, we explored the Rust programming language, its unique features, and its growing popularity in the technology community. Building on this foundation, we embarked on an ambitious project to create a domain-specific language (DSL) using Rust. This report documents the development of this DSL, providing a comprehensive overview of the implementation process, challenges encountered, and solutions devised.

2 Motivation

Our motivation for this project stems from Rust's safety, speed, and flexibility. Rust's design principles, such as ownership and type safety, make it an excellent choice for implementing interpreters and compilers. The language's growing popularity in systems programming and its ability to prevent common programming errors further inspired us to choose Rust for our DSL project.

3 Design and Implementation

3.1 Lexical Analysis (Lexing) and Tokenization

The first step in creating our DSL was lexical analysis, where the source code is converted into a sequence of tokens. Tokens are the basic building blocks of the language, representing keywords, identifiers, operators, and other symbols. We implemented a lexer in Rust to scan the input source code and generate tokens.

```
enum Token {
    Identifier(String),
    Number(i32),
    Plus,
    Minus,
    // other tokens...
}

fn lex(input: &str) -> Vec<Token> {
    let mut tokens = Vec::new();
    let mut chars = input.chars().peekable();
    while let Some(&ch) = chars.peek() {
        match ch {
            '0'..='9' => {
                let mut number = String::new();
                while let Some(&c) = chars.peek() {
                    if c.is_digit(10) {
                        number.push(c);
                        chars.next();
                    } else {
                        break;
                    }
                }
                tokens.push(Token::Number(number.parse().unwrap()))
    ;
            }
            '+' => {
                tokens.push(Token::Plus);
                chars.next();
            }
            '-' => {
                tokens.push(Token::Minus);
                chars.next();
            }
            // other cases...
            _ => chars.next(),
        }
    }
    tokens
}
```

3.2 Parsing Techniques and Abstract Syntax Trees (ASTs)

After tokenization, the next step was parsing, where tokens are transformed into a structured representation called an abstract syntax tree (AST). The AST

represents the grammatical structure of the source code and is used for further processing, such as code generation and execution.

```
enum ASTNode {
    BinOp(Box<ASTNode>, Token, Box<ASTNode>),
    Number(i32),
    // other nodes...
}

fn parse(tokens: &[Token]) -> ASTNode {
    // implementation of parsing logic to create AST
}
```

3.3 Code Evaluation and Execution

The final step involved evaluating the AST to execute the code. This step involves interpreting the AST nodes, managing variable assignments, function calls, and control structures.

```
fn evaluate(ast: &ASTNode) -> i32 {
    match ast {
        ASTNode::BinOp(left, Token::Plus, right) => evaluate(left)
    + evaluate(right),
        ASTNode::BinOp(left, Token::Minus, right) => evaluate(left)
     - evaluate(right),
        ASTNode::Number(value) => *value,
        // other cases...
    }
}
```

3.4 Utilization of Rust Language Features

Rust's features such as pattern matching, ownership, and lifetimes played a crucial role in the implementation of our DSL. These features helped in managing memory safely and efficiently, ensuring the robustness of the interpreter.

4 Additional Project: WiFi Tool Implementation

In addition to the DSL, we developed a WiFi tool using Rust. This tool lists WiFi profiles on a Windows system and retrieves their stored passwords. The implementation leverages Rust's `std::process::Command` for executing system commands and handling their output.

```
use std::process::Command;
use std::str;

fn main() {
    // Retrieve WiFi profiles
    let output = Command::new("netsh")
```

```
        .args(&["wlan", "show", "profiles"])
        .output()
        .expect("Failed to execute command");

    let output_str = str::from_utf8(&output.stdout).expect("Invalid
     UTF-8");

    // Retrieve passwords for each profile
    for line in output_str.lines() {
        if line.contains("All User Profile") {
            let profile_name = line.split(':').nth(1).map(|s| s.
    trim()).unwrap_or("");

            let key_output = Command::new("netsh")
                .args(&["wlan", "show", "profile", profile_name, "
    key=clear"])
                .output()
                .expect("Failed to execute command");

            let key_output_str = str::from_utf8(&key_output.stdout)
    .expect("Invalid UTF-8");

            // Extract the password
            let key = key_output_str
                .lines()
                .filter(|line| line.contains("Key Content"))
                .map(|line| line.split(':').nth(1).map(|s| s.trim()
    ).unwrap_or(""))
                .next()
                .unwrap_or("");

            println!("{:<30}|  {:<}", profile_name, key);
        }
    }

    // Wait for user input before closing
    println!("Press Enter to exit...");
    let mut input = String::new();
    std::io::stdin()
        .read_line(&mut input)
        .expect("Failed to read line");
}
```

5 Conclusion

This project demonstrated the feasibility and advantages of using Rust for implementing a domain-specific language and a practical WiFi tool. Rust's safety, speed, and modern features greatly facilitated the development process, resulting in robust and efficient applications. Our experience with Rust has been highly educational, providing us with valuable insights into systems programming and language implementation.