

CS-3513

Programming Languages

Report

Kularathne W H S - 210301L
Mduranga H. D.E - 210352R

Introduction

The project aims to develop a robust software system capable of processing programs written in the RPAL programming language. This endeavor involves the implementation of key components including a lexical analyzer, parser, Abstract Syntax Tree (AST) conversion algorithm, and a Control Stack Environment (CSE) machine. RPAL, with its defined lexical rules and grammar, poses a challenging yet rewarding task for software engineers to tackle.

Objective

The primary objective of this project is to construct a programming language that can effectively analyze, parse, and execute RPAL programs. This involves translating RPAL source code into structured representations such as ASTs and ultimately executing them using a CSE machine. By achieving this objective, we aim to deepen our understanding of programming language theory and implementation while delivering a robust software tool for RPAL development.

Scope

The scope of the project encompasses the complete lifecycle of processing RPAL programs, from lexical analysis to execution. This includes the following key components:

Lexical Analyzer: Responsible for tokenizing RPAL source code according to the specified lexical rules.

Parser: Constructs a parse tree or AST from the tokens generated by the lexical analyzer, adhering to the grammar rules of RPAL.

Abstract Syntax Tree (AST) Conversion: Converts the parse tree or AST into a standardized representation suitable for subsequent processing.

CSE Machine: Implements a Control Stack Environment machine capable of executing RPAL programs by traversing the AST and performing the necessary computations.

Input Format

Here is a simple explanation about rpal programming language.

Expressions

Constants: Constants can be integers, truth values (boolean), strings, tuples, functions, or dummy values.

Function Application: Functions are applied to arguments by juxtaposition, where the function is followed by its arguments.

Conditional Expression: A conditional expression evaluates a condition and returns one of two possible results based on whether the condition is true or false.

Definitions

Constants: Constants are defined with a value, such as `let X = 3`.

Functions: Functions are defined with a parameter and a body, such as `let Abs = fn X. X is 0 -> -X | X`.

Recursion

Recursive Functions: Recursive functions are defined with the `rec` keyword, such as `let rec Fact N = N eq 1 -> 1 | N * Fact(N-1)`.

Tuples

Tuple Construction: Tuples are constructed with elements separated by commas, such as `(1, 2, 3)`.

Tuple Extension: Tuples can be extended by adding elements to the end, such as `let T = (2, 3) in let A = T aug 4`.

Operator Precedence

Operators: RPAL supports various operators, such as arithmetic operators (+, -, *, /), relational operators (eq, ne, ls, gr, le, ge), and logical operators (or, &, not).

The project takes RPAL programs as input, which are written in the RPAL language. RPAL programs consist of expressions, definitions, and recursive functions, following specific syntax and grammar rules. These programs are read from input files and processed to generate Abstract Syntax Trees (ASTs) for further analysis and execution.

Output Format of -ast switch

Example Program

```
let rec sum N =  
    N eq 1 -> 1 | N + sum (N-1)  
in  
Print(sum 10)
```

Output

```
PS E:\semester4\programming_languages\Scanner\cse_macheine> ./rpal -ast test.txt  
let  
  .rec  
  ..function_form  
  ...<ID:sum>  
  ...<ID:N>  
  ...->  
  ....eq  
  .....<ID:N>  
  .....<INT:1>  
  .....<INT:1>  
  ....+  
  .....<ID:N>  
  .....gamma  
  .....<ID:sum>  
  .....-  
  .....<ID:N>  
  .....<INT:1>  
  .gamma  
  ..<ID:Print>  
  ..gamma  
  ...<ID:sum>  
  ...<INT:10>  
55  
PS E:\semester4\programming_languages\Scanner\cse_macheine> █
```

Testing and Validation

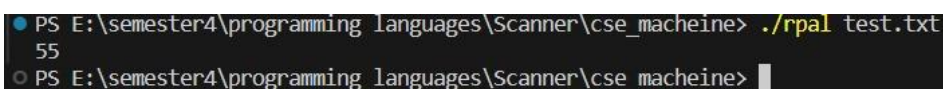
The testing process involves unit testing each component (lexical analyzer, parser, AST conversion), integrating them to test interactions, and executing RPAL programs with varied syntax and complexity for functional validation. Boundary testing ensures proper handling of edge cases, while regression testing confirms existing functionality post-modifications. Validation testing compares system-generated outputs against expected results for correctness. Additionally, error handling testing validates the system's ability to identify and report errors accurately. Through iterative application of these strategies, the correctness and robustness of the implemented components are thoroughly validated, ensuring reliable performance of the RPAL language interpreter.

Results and Comparison

Example Program 1

```
let rec sum N =  
    N eq 1 -> 1 | N + sum (N-1)  
in  
Print(sum 10)
```

Output Image

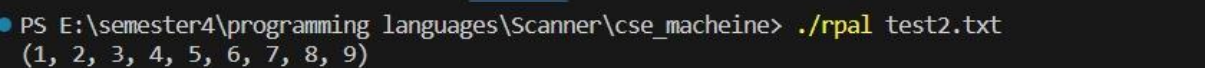


```
PS E:\semester4\programming languages\Scanner\cse_macheine> ./rpal test.txt  
55  
PS E:\semester4\programming languages\Scanner\cse_macheine> █
```

Example Program 1

```
let rec firstNo P C = (P/10) Is 10 -> ((P/10),C)
| (firstNo (P/10) (C+1))
in
let sternN Q = (Q - (firstNo Q 1 1) * 10**(firstNo Q 1 2))
in
let rec lastNo L = L Is 10 -> L | lastNo (sternN L)
in
let rec isPal S = S Is 10 -> 1 | (firstNo S 1 1) ne (lastNo S) -> 0 |
isPal ((sternN S - lastNo S) / 10)
in
let rec PalList Sno Lno pl = Sno eq Lno -> pl | isPal Sno eq 1 -> (PalList (Sno +1) Lno (pl
aug Sno)) |
(PalList (Sno +1) Lno pl )
in
let calc A B = (PalList A B nil)
in
Print(calc 1 10)
```

Output Image



```
PS E:\semester4\programming languages\Scanner\cse_macheine> ./rpal test2.txt
(1, 2, 3, 4, 5, 6, 7, 8, 9)
```

Simple Overview of Content

Lexical Analyzer

The provided code snippet represents a lexical analyzer for the RPAL language. Its primary task is to tokenize input RPAL code into distinct lexical elements such as identifiers, strings, operators, integers, keywords, punctuation, and undefined tokens.

- **Tokenization:** The scan function reads characters from the input file stream and identifies the type of token based on the characters encountered.
- **Identifier Tokenization:** The *readIdentifierToken* function reads characters until it encounters a character that doesn't belong to an identifier (such as digits, letters, or underscores). It distinguishes keywords from regular identifiers by checking against a list of keywords.
- **Integer Tokenization:** The *readIntegerToken* function reads characters until it encounters a character that isn't a digit, tokenizing the integer.
- **Operator Tokenization:** The *readOperatorToken* function identifies and tokenizes sequences of operator characters.
- **String Tokenization:** The *readStringToken* function tokenizes string literals, handling escape characters and ensuring the string is correctly terminated.
- **Punctuation Tokenization:** The *readPunctuationChar* function identifies and tokenizes punctuation characters.
- **Comment Handling:** The *resolveCommentOrOperator* function determines whether a sequence starting with '/' is a comment or part of an operator.
- **Whitespace Handling:** Whitespace characters are ignored unless they're part of a string literal.

Parser

The provided code represents a parser for the RPAL language. Here's a breakdown of its functionality and structure:

- **Parsing Procedure Functions:**

- (1) Functions like *fn_E*, *fn_T*, *fn_B*, etc., represent parsing procedures for different non-terminals of the RPAL grammar.
- (2) Each function corresponds to a specific non-terminal and recursively parses the input according to the rules defined in the RPAL grammar.

- **Building Abstract Syntax Tree (AST):**

- (1) The `buildTree` function constructs nodes of the abstract syntax tree (AST) based on the parsed input.
- (2) It takes the label of the node being constructed and the number of subtrees to pop from the parser stack to make children of the new node.

- **Tokenization:** The *readNextToken* function ensures that the next token read from the input file matches the expected token. If not, it raises an error.

- **Parsing Non-Terminals:**

- (1) Each parsing function handles a specific non-terminal symbol from the RPAL grammar.

*For example, *fn_E* handles the *E* non-terminal, *fn_T* handles the *T* non-terminal, and so on.*

- (2) These functions recursively call each other based on the grammar rules until the entire input is parsed.

- **Error Handling:** The parser includes error handling mechanisms to handle cases where the input does not conform to the grammar rules. Errors are detected when the expected token does not match the actual token read from the input.
- **AST Construction:** As the input is parsed, nodes representing the abstract syntax tree are constructed and pushed onto a stack. Non-terminals are represented as interior nodes, while terminals are represented as leaf nodes.
- **Integration with Lexical Analyzer:** The parser interacts with the lexical analyzer by calling the `scan` function to read the next token from the input file.

CSE Machine

The provided code implements a variant of the CSE (Compiled stack Environment) machine, a virtual machine designed to execute programs written in Scheme, a dialect of Lisp. Here's a simplified description of how this CSE machine works:

Initialization: The *initializeCSEMachine()* function sets up the initial state of the CSE machine, including initializing stacks and other data structures.

Processing Control Stack: The *processCSEMachine()* function is the core of the machine. It iterates through the control stack, which contains instructions for the machine to execute.

Rule-Based Execution: The machine operates based on a set of rules defined for various scenarios encountered during execution. These rules correspond to different Lisp constructs, such as lambda expressions, built-in functions, tuple operations, conditionals, etc.

Stack Manipulation: The machine manipulates two main stacks:

Control Stack (*cseMachineControl*): Contains instructions for the machine to follow.

Data Stack (*cseMachineStack*): Contains data operands being manipulated during execution.

Execution Flow: The machine processes each instruction from the control stack, applying the appropriate rule based on the type of instruction encountered.

Rules handle operations like function application, arithmetic and logical operations, tuple operations, conditionals, etc. Depending on the rule applied, data may be pushed or popped from the data stack, and control flow may be altered.

Handling Errors: The machine includes error handling mechanisms to detect and handle various error conditions, such as invalid operand types, undefined behaviors, etc.

Challenges and Solutions

Several challenges may arise during the implementation of an RPAL lexical analyzer, parser, and AST conversion algorithm:

Grammar Complexity: RPAL's grammar can be intricate due to its functional nature, recursive definitions, and support for tuples. Handling these complexities while ensuring accurate parsing poses a challenge.

Error Handling: Effective error detection and reporting are crucial, especially in dynamically typed languages like RPAL. Designing robust error handling mechanisms to identify syntax errors and provide meaningful error messages is challenging.

AST Construction: Generating an accurate and efficient Abstract Syntax Tree (AST) from parsed tokens requires careful design. Ensuring the AST accurately represents the structure and semantics of the RPAL program is essential.

Language Features Support: RPAL may support various language features like recursion, conditionals, and function application. Ensuring that the lexer, parser, and AST conversion algorithm can handle these features correctly adds complexity to the implementation.

Addressing these challenges requires careful planning, design, and implementation strategies to develop a robust and efficient RPAL interpreter.

Conclusion

In conclusion, the implementation of the CSE (Control-Stack Environment) machine demonstrates its efficacy in evaluating lambda calculus expressions, showcasing functionalities such as lambda abstractions, function applications, binary and unary operators, conditionals, and tuple operations. The adherence to specific rules derived from lambda calculus semantics ensures accuracy and consistency throughout the computation process, complemented by error detection and handling mechanisms for addressing potential issues. While the focus has been on functionality and correctness, future optimizations could enhance performance and efficiency through techniques like memoization, lazy evaluation, or parallel processing. Moreover, potential extensions could explore support for higher-order functions, pattern matching, type inference, or integration with other programming paradigms, laying the groundwork for deeper exploration into functional programming concepts.

References

Stack Overflow:

URL: <https://stackoverflow.com/>

Simplilearn:

URL: <https://www.simplilearn.com/best-programming-languages-start-learning-today-article>