Programming Languages Project

Group 41

Lishan S.D. 210339J
Nanayakkara A.H.M. 210404F

Table of Contents

1. Introduction

This report documents the implementation of a lexical analyzer and parser for the RPAL language. The project required the creation of an Abstract Syntax Tree (AST) from an input RPAL program and its transformation into a Standardized Tree (ST). Additionally, a CSE machine was developed to execute the RPAL programs.

2. Problem Description

The goal of this project was to develop a lexical analyzer and a parser for the RPAL language according to the specifications provided in RPAL_Lex.pdf and RPAL_Grammar.pdf. The parser was required to output an Abstract Syntax Tree (AST) for the given input program. Additionally, we needed to create an algorithm to convert the AST into a Standardized Tree (ST) and implement a Control Structure Environment (CSE) machine. The program was to be tested by ensuring its output matched that of the reference program "rpal.exe".

3. Objectives

The primary objectives of this project were:

- To develop a lexical analyzer that tokenizes RPAL code.
- To implement a parser that constructs an AST based on the provided RPAL grammar.
- To convert the AST into a Standardized Tree (ST).
- To implement a CSE machine to execute the Standardized Tree.
- To ensure the output matches that of a reference implementation (rpal.exe).

4. Solution Overview

Scanner

The scanner.py module handles the lexical analysis. It defines various token types, reserved keywords, and punctuation marks. The tokenizer reads the input program character by character, recognizing tokens and skipping whitespace and comments. The Screener class processes the tokens to merge multi-character operators and filter out comments.

Key Components:

- Token Types: Defined using an Enum to categorize different tokens.
- Tokenizer: Reads the input and generates tokens.
- Screener: Merges specific tokens and removes comments.

Parser

The myrpal.py module implements the parser. It reads the tokens produced by the scanner and constructs the AST according to the RPAL grammar. The parser employs recursive descent parsing techniques, with functions corresponding to different non-terminal symbols in the grammar.

Key Components:

- ASTNode: Class to represent nodes in the AST.
- Parser: Contains methods to parse expressions (procE), terms (procT), factors (procF), and so on.
- Tree Building: Methods to construct the AST (buildTree).

Abstract Syntax Tree (AST)

The AST represents the syntactic structure of the input RPAL program. Each node in the AST corresponds to a construct in the language (e.g., operators, literals, identifiers). The tree is constructed during parsing and can be traversed for further transformations or execution.

Standardized Tree (ST)

The ST is a transformed version of the AST, adhering to a standard form that simplifies execution. The transformation ensures that all constructs are represented in a uniform manner, facilitating the generation of control structures.

Control Structure Generator

The control structure generator translates the ST into control structures, which are sequences of instructions that the CSE machine can execute. This step bridges the gap between the tree representation of the program and its execution semantics.

CSE Machine

The CSE machine executes the control structures generated from the ST. It simulates the runtime environment of the RPAL language, handling function calls, variable bindings, and expression evaluations.

Key Components:

- Control Structures: Represent program logic in an executable form.
- Execution Engine: Executes the control structures, producing the final output.

5. Implementation

5.1 Scanner.py

Overview

The scanner.py module is responsible for lexical analysis of the RPAL language. It reads the input source code, identifies tokens based on predefined rules, and produces a stream of tokens for further processing by the parser.

Classes

1. Token

Represents a token in the input source code.

Attributes:

- type: The type of the token (e.g., identifier, integer, keyword).
- value: The value of the token (e.g., actual string or number).

Methods:

- __init__(self, type, value): Initializes a token with a given type and value.

2. Type

An enumeration of different token types.

Members:

- Various token types such as RESERVED_KEYWORD, ID, INT, COMMENT, PLUS, MINUS, MUL, DIV, etc.

3. State

Maintains the current state of the tokenizer, including the current character, line number, and column number.

Attributes:

- current_char: The current character being processed.
- column_num: The current column number in the input.
- line_num: The current line number in the input.

Methods:

- __init__(self): Initializes the state with default values.

4. Tokenizer

Performs lexical analysis by reading characters from the input and generating tokens.

Attributes:

- text: The input source code as a string.
- pos: The current position in the input text.
- state: The current state of the tokenizer.

Methods:

- __init__(self, text): Initializes the tokenizer with the given text and sets the initial state.
- error(self): Raises an exception for handling errors during tokenization.
- advance(self): Moves the current position to the next character and updates the state.
- skip_space(self): Skips over whitespace characters, updating line and column numbers for newlines.

- integer(self): Reads a sequence of digits and returns it as an integer token.
- identifier(self): Reads a sequence of alphanumeric characters and underscores to identify an identifier token.
- comment(self): Reads characters until the end of the line to identify a comment token.
- string(self): Reads characters within single quotes to identify a string token.
- next_token(self): Main method to get the next token from the input text.

5. Screener

Processes the list of tokens to apply further transformations and filtering.

Attributes:

- tokens: A list of tokens to be processed.

Methods:

- __init__(self, tokens): Initializes the screener with a list of tokens.
- merge(self): Merges specific sequences of tokens into single tokens.
- screen_reserved(self): Identifies tokens that match reserved keywords and updates their type.
- remove_comments(self): Removes all comment tokens from the list of tokens.
- screen(self): Runs the merging, comment removal, and reserved keyword identification in sequence and returns the processed list of tokens.

5.2 Node.py

Overview:

The ASTNode class represents nodes in the Abstract Syntax Tree (AST) of an RPAL program. It provides methods for standardizing the AST, printing the tree structure to the console, and creating a copy of a node.

Methods:

1. **standarize(root):**
    - Description: This method performs standardization of the AST. It recursively processes each node and its children to match AST node types and standardize them accordingly.
    - Parameters:
        - root: The root node of the AST to be standardized.
    - Returns: The standardized AST.

2. **__init__(type):**
    - Description: Constructor method to initialize an ASTNode object with the given type.
    - Parameters:
        - type: The type of the AST node.
    - Attributes:
        - type: The type of the AST node.
        - value: The value associated with the node (default is None).
        - sourceLineNumber: The line number in the source file where the node is located (default is -1).
        - child: Reference to the first child node (default is None).
        - sibling: Reference to the next sibling node (default is None).
        - previous: Reference to the previous sibling node (default is None).
        - indentation: The level of indentation for printing (default is 0).
3. **print_tree():**
    - Description: Recursively prints the tree structure to the console.
4. **print_tree_to_cmd():**
    - Description: Prints the tree structure to the console with proper indentation.
5. **createCopy():**
    - Description: Creates a deep copy of the ASTNode object.
    - Returns: A new ASTNode object with the same attributes as the original.

5.3 Env.py

Overview:

The Environment class represents an environment in an RPAL program. It provides methods for setting variable bindings, retrieving variable values, and managing the environment index.

Attributes:

- index: An integer representing the index of the environment.
- vars: A dictionary storing variable bindings (key-value pairs).
- parent: A reference to the parent environment.

Methods:

1. **__init__(index):**
   - Description: Initializes the environment with a given index and an empty variables dictionary.
   - Parameters:
     - index: An integer representing the index of the environment.
2. **set_parameters(parent_environment, key, value):**
   - Description: Sets variable bindings in the environment.
   - Parameters:
     - parent_environment: The parent environment.
     - key: The key (variable name) to bind.
     - value: The value associated with the key.
   - Note: This method also updates the parent reference if provided.
3. **get_index():**
   - Description: Returns the index of the environment.
4. **get_value(key):**
   - Description: Retrieves the value associated with a variable key.
   - Parameters:
     - key: The key (variable name) to retrieve the value for.
   - Returns: The value associated with the key if found, otherwise None.

5.4 controlStructure.py

Overview

The controlStructure.py module is responsible for generating control structures for an Abstract Syntax Tree (AST) of an RPAL program. This involves traversing the AST, creating control structures for lambda expressions, arrow expressions, gamma expressions, and tau expressions, and managing a mapping of these control structures.

Classes

1. Tau

Represents a Tau expression in the AST.

Attributes:

- n: The number of elements in the Tau expression.

Methods:

- __init__(self, n): Initializes a Tau expression with the given number of elements.

2. Beta

Represents a Beta expression in the AST.

Attributes:

- None

Methods:

- __init__(self): Initializes a Beta expression.

3. ControlStructure

Represents a control structure in the AST.

Attributes:

- index: The index of the control structure.
- delta: The list of expressions or sub-structures within the control structure.

Methods:

- __init__(self, index, delta): Initializes a control structure with a given index and delta.

4. LambdaExpression

Represents a lambda expression in the AST.

Attributes:

- environmentIndex: The index of the environment.

- lambdaIndex: The index of the lambda expression.
- item: The token or list of tokens representing the lambda variables.

Methods:

- __init__(self, environmentIndex, lambdaIndex, token): Initializes a lambda expression with the given environment index, lambda index, and token.
- print_lambda_expression(self): Prints the lambda expression (currently incomplete and not fully implemented).

5. ControlStructureGenerator

Generates control structures by performing a pre-order traversal of the AST.

Attributes:

- current_index_delta: The current index for delta structures.
- queue: A queue for managing nodes during traversal.
- map_control_structs: A mapping of control structures by their index.
- current_delta: The current list of delta structures being processed.

Methods:

- __init__(self): Initializes the control structure generator with default values.
- generate_control_structures(self, root): Generates control structures starting from the root of the AST and returns a mapping of these structures.
- pre_order_traversal(self, root, delta): Performs a pre-order traversal of the AST, generating control structures based on the node types encountered.

5.5 cseMachine.py

Overview

The CSEMachine.py module defines a CSE (Control Structure Execution) machine to interpret and execute control structures in a functional language context. Here's an overview and some clarifications on the key parts of the code:

Key Components

1. **CSEMachine Class**:
    - Initialization (__init__): Initializes environments and control structures. The root environment is created and added to both the stack and control.
    - Execution (executeCSEMachine): The main execution loop that processes the control stack, performs operations, and manages environments.
2. **Operations**:
    - Binary Operations (binaryOperation): Handles operations like addition (+), subtraction (-), multiplication (*), division (/), power (**), logical operations (&, or), and comparisons (>, <, >=, <=, ==, !=). The results are created as Node.ASTNode objects.
    - Unary Operations (unaryOperation): Handles unary operations like negation (neg) and logical NOT (not).
3. **Special Functions**:
    - Print: Outputs a string or a tuple by recursively printing its elements.
    - Conc: Concatenates two strings.
    - Stem: Retrieves the first character of a string.
    - Stern: Retrieves the rest of the string after the first character.
    - Null: Checks if a list is empty or if an ASTNode is of type nil.
    - ItoS: Converts an integer to a string.
    - Type Checking Functions: Isinteger, Istruthvalue, Isstring, Istuple, Isdummy check the type of the given node.
    - Order: Returns the length of a list or zero if the element is not a list.
4. **Environment Management**:
    - Environment (Environment): Manages variable bindings and scopes. When a lambda expression is evaluated, a new environment is created.
    - Eta: Represents the environment and lambda expression to facilitate recursion.
5. **Control Structures**:
    - LambdaExpression: Represents a lambda function with its environment.
    - Beta: Handles conditional branching.
    - Tau: Handles tuple creation.

Execution Flow

- **Control Stack Processing**: The control stack is processed from top to bottom. Depending on the type of the control structure at the top of the stack, different actions are taken.
- **Lambda Expressions**: When a lambda expression is encountered, it is associated with the current environment and pushed onto the stack.
- **ASTNode Operations**: Depending on the type of node (binary, unary, special function), appropriate operations are performed.
- **Conditional Branching**: Handled using Beta nodes which check the top of the stack for true/false values and branch accordingly.

- **Environment Management**: Environments are managed and switched as needed for lambda evaluations.

Detailed Analysis

- **Binary and Unary Operations**: These are central to the functional execution, handling arithmetic and logical operations with careful type checks.
- **Special Functions**: These extend the language capabilities, allowing for string manipulation and type checking.
- **Environment Management**: Ensures that variable scopes are properly managed, especially during lambda evaluations.

5.6 myrpal.py

Overview

myrpal.py is a Python script that serves as an interpreter for the Rpal programming language. It takes an Rpal program file as input, tokenizes and screens it, parses the tokens to build an Abstract Syntax Tree (AST), standardizes the AST, generates control structures, and executes them using a Control Structure Evaluation (CSE) Machine.

Usage

```
python .\myrpal.py path_to_the_file -ast
```

- **path_to_the_file**: The path to the RPAL program file.
- **-ast**: Optional flag to print the Abstract Syntax Tree (AST) to the terminal.

Components

1. **Parser Class**: Contains methods for parsing tokens and building an AST.
2. **ControlStructureGenerator Class**: Generates control structures from the AST.
3. **CSEMachine Class**: Executes control structures using a Control Structure Evaluation (CSE) Machine.
4. **ASTNode Class**: Represents nodes in the Abstract Syntax Tree (AST) and provides methods for tree manipulation and printing.
5. **Scanner Module**: Tokenizes the input Rpal program.
6. **Screener Module**: Screens tokens to remove comments and whitespace.
7. **controlStructure Module**: Contains the ControlStructureGenerator class.
8. **cseMachine Module**: Contains the CSEMachine class.

Main Program Flow

1. Read the input Rpal program file.
2. Tokenize the program using the Scanner module.
3. Screen the tokens to remove comments and whitespace.
4. Parse the tokens to build an Abstract Syntax Tree (AST) using the Parser class.
5. Standardize the AST using the ASTNode class.
6. Generate control structures from the AST using the ControlStructureGenerator class.
7. Execute the control structures using the CSEMachine class.
8. Optionally, print the AST to the console if the **-ast** flag is present.

Output

- The output of the input RPAL program will be printed in the terminal.
- If the **-ast** flag is present, the AST is also printed to the terminal.

6. Testing

Testing involved running various RPAL programs through the implemented system and comparing the output with the reference implementation (rpal.exe). The tests included programs with different language constructs to ensure comprehensive coverage. The test cases are included in the file "test_cases".

Input Format

1. To execute the program

- Only to get the output
```
python .\myrpal.py path_to_the_file
```
- To get the abstract syntax tree
```
python .\myrpal.py path_to_the_file -ast
```

2. To execute with Makefile

- Only to get the output
```
make run ARGS="path_to_the_file"
```
- To get the abstract syntax tree
```
make run ARGS="path_to_the_file -ast"
```

Output

- Without the “-ast” switch, the output of the input RPAL program will be printed to the terminal.
- With the “-ast” switch, the program prints the abstract syntax tree to the terminal.

7. Conclusion

This project enhanced our understanding of language interpretation by delving into parsing, tokenization, and control structure evaluation. By implementing a lexical analyzer, parser, and CSE machine, we gained practical experience in creating complex systems that adhere to strict language specifications. This hands-on experience was invaluable in equipping us with the skills necessary for programming language interpreter implementation.