CS 3513: Programming Languages

Programming Languages Project

Group CodeCrafters

07/06/2025

Gallage A.H – 220172A

Wijesooriya J.M.I – 220724U

Contents

RPAL Interpreter Implementation

Introduction

For this project, we have to build a full implementation of an interpreter for the Right-reference Pedagogic Algorithmic Language (RPAL). A variety of important components make up the project, and each component helps with the interpretation.

This project includes the following components:

- Lexical analyzer
- Parser
- Standardizer
- CSE machine

Implementation Details

1. Technology
    - ➢ All the steps in this project were completed using python and no application-specific libraries had to be installed ahead of time.

2. Lexical Analyzer
    - ➢ The Lexical Analyzer creates a series of tokens and sends them on to the parser.

3. Parser
    - ➢ After the grammar from given document in moodle is used by the parser, it makes an Abstract Syntax Tree (AST) using the lexemes formed by the lexical analyzer. We made our own parser, instead of using one that was already constructed.

4. AST to ST Conversion (Standardizer)
    - ➢ We applied an algorithm to turn the AST into a ST, after it was ready. The program is organized into ST form for it to be executed by the CSE.

5. CSE Machine
    - A CSE machine was used to perform the programs in the RPAL language as written in the ST. The CSE machine works by going through the symbol table and doing the required operations.

6. Input and Output
    - The program looks at an input file that contains RPAL program code. It deals with the -ast option to display the Abstract Syntax Tree (AST) for the user's input program. In addition, by adding -std switch to the command, a standard tree of lambdas and gammas is printed.

Program Structure

1. Lexical Analyzer

```
class Token:
    def __init__(self, value, type):
        self.value = value
        self.type = type


himadreee, 2 hours ago | 2 authors (himadreee and one other)
class RPAL_Scanner:

    punction = [")", "(", ";", ","]

    # List of operator symbols in RPAL language
    operator_symbol = [
        "+", "-", "*", "<", ">", "&", ".", "@", "/", ":", "=", "~", "|", "$",
        "!", "#", "%", "^", "_", "[", "]", "{", "}", '"', "`", "?"
    ]

    # Reserved keywords in RPAL
    # These keywords cannot be used as identifiers
    RESERVED_KEYWORDS = [
        "fn", "where", "let", "aug", "within", "in", "rec", "eq", "gr", "ge",
        "ls", "le", "ne", "or", "@", "not", "&", "true", "false", "nil",
        "dummy", "and", "|"
    ]

    # Elements that can be part of comments
    comment_elements = ['"', "\\", " ", "\t"]

    def __init__(self, file):
        self.file = file

    # Scanning function to tokenize the input file
    def Scanning(self):
        token_list = []
```

Data Formatting:

- Input: The tokenize function expects a string as input, representing the program code to be tokenized.

- Output: It provides a list of Tokens objects in which each token includes details about its type and value .
- Parameter Passing: The tokenize function uses a copy of the input string because it takes the string by value.
- Error Handling: When it meets an unrecognized token or cannot tokenize the input, the tokenize function issues an error message.
- Return Values: The return of the tokenize function is a list of tokens, which gives the calling code the ability to go through them and work with them more.

2. Parser

```
class ASTParser:
    def __init__(self, tokens):
        self.tokens = tokens          # List of tokens to parse
        self.current_token = None     # Current token being processed
        self.index = 0                # Index of current token in tokens list
        self.stack = []               # Stack used during AST construction
        self.prevToken = None         # Previous token processed (for error reporting)
        self.has_error = False       # Flag indicating parsing errors

    # Core parsing method that initiates the parsing process
    def parse_tokens(self, astFlag):
        self.current_token = self.tokens[0]
        self.parse_expression()  # Start with E production rule (top-level expression)

        # Handle parsing results based on flags and success
        if self.has_error:
            print("Parsing error")
        elif astFlag == "-ast":
            self.print_pre_order(self.stack[0])  # Print AST if requested
        elif astFlag == "":
            pass  # No output needed
        else:
            print("Input command incorrect.")

    # Error state check
    def isAnError(self):
        return self.has_error

    # Helper methods for parsing

    # Reads and consumes the next token, checking for expected values/types
    def read(self, value, type):
        self.current_token = self.tokens[self.index]
```

Data Formatting:

- Input: The method named parse takes a list of tokens that represents the lexemes produced by the lexical analyzer as input.
- Output: It creates an Abstract Syntax Tree (AST) that reflects the program after it has been parsed.

- Parameter Passing: The list of tokens is passed to the parse method with a reference, so the method can make changes to it as the parsing occurs.
- Error Handling: Should parsing not work out, the parse method produces an error message and returns None to indicate that the process ended unsuccessfully.
- Return Values: The call to parse creates the AST, so the calling code can handle it or run other operations on the parsed code.

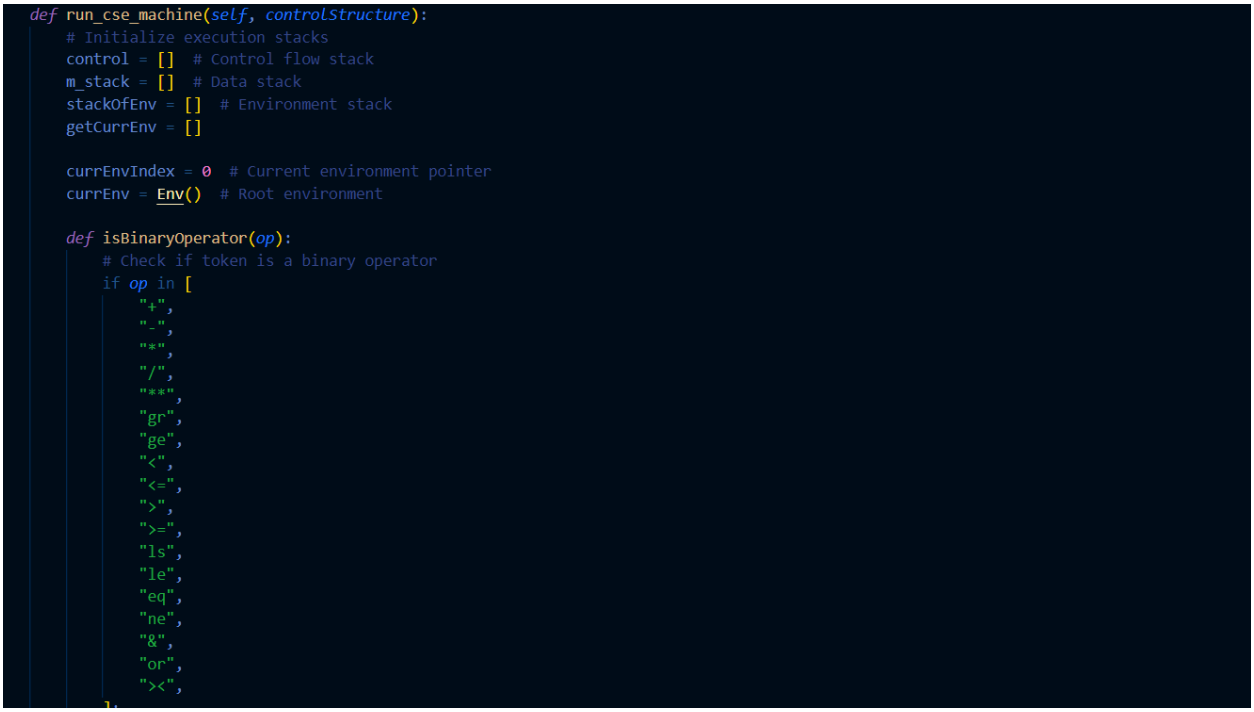
3. AST to ST Convention (Standardizer)

```
class standardizer:
    def __init__(self, tree):
        self.tree = tree
        self.ST = None

    def standardize_tree(self, x):
        self.transform_node(x)

    def copy_ast_node(self, x):
        # Create new node with same properties as input
        node = ASTNode(x.value, x.type)
        node.left = x.left  # preserve left subtree
        node.right = None  # initialize right child as empty
        return node

    def transform_node(self, node):
        if node is None:
            return None

        # Process children first (post-order traversal)
        self.transform_node(node.left)
        self.transform_node(node.right)

        # Handle different node types
        if node.get_label() == "let":
            if node.left.get_label() == "=":
                node.set_label("gamma")
                node.set_node_type("KEYWORD")
                P = self.copy_ast_node(node.left.right)
                X = self.copy_ast_node(node.left.left)
                E = self.copy_ast_node(node.left.left.right)
                node.left = ASTNode("lambda", "KEYWORD")
                node.left.right = E
```

Data Formatting:

- Input: The nodes returned from the parser are changed into other nodes with more attributes and methods, with the help of “ASTParser,” before passing the root node to the standardize function.
- Output: In turn, it generates the standard AST structure, which consists of nodes.
- Parameter Passing: The root node that was returned by parsing through the ASTParser.
- Return Values: Standard AST object is built from scratch.

4. CSE Machine



```
def run_cse_machine(self, controlStructure):
    # Initialize execution stacks
    control = []  # Control flow stack
    m_stack = []  # Data stack
    stackOfEnv = []  # Environment stack
    getCurrEnv = []

    currEnvIndex = 0  # Current environment pointer
    currEnv = Env()  # Root environment

    def isBinaryOperator(op):
        # Check if token is a binary operator
        if op in [
            "+",
            "-",
            "*",
            "/",
            "**",
            "gr",
            "ge",
            "<",
            "<=",
            ">",
            ">=",
            "ls",
            "le",
            "eq",
            "ne",
            "&",
            "or",
            "><",
```

Data Formatting:

- Input: The procedure uses the control, stack, and environment returned by “cse_machine”, as they were at the time the “cse_machine” was called with the standardized AST as input.
- Output: It produces the final output result for the RPAL program code.
- Parameter Passing:
    - Control Stack: The control attribute holds the symbols and the instructions that are executed when using the CSE machine.
    - Stack: Stack in CSE means the storage area in the machine where temporary results are kept as the code runs.
    - Environment: The value of the environment attribute shows the environment of the CSE machine, which assigns values to the identifiers used.

- Return Values: run_cse_machine returns the end result after running the CSE machine.

Usage

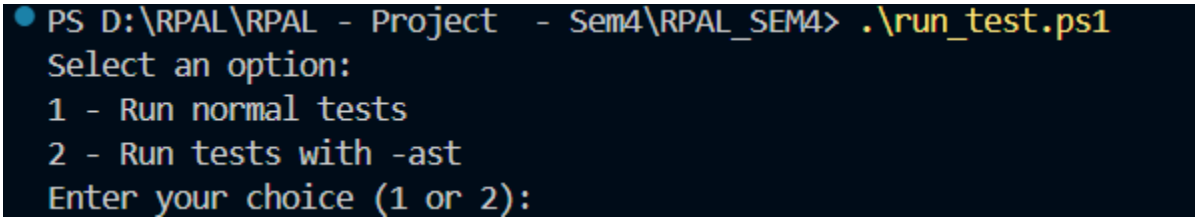
The interpreter can be run by two methods.

1. Using Direct Python Commands
2. Using Makefile

1. Using direct python commands
   1.1. Open the terminal and navigate to the directory where the interpreter is located.
   1.2. Run the following commands for output, ast and standardized ast respectively.
      - python myrpal.py input.txt
      - python myrpal.py input.txt
      - python myrpal.py -ast input.txt

2. Using Makefile
   2.1. Open the terminal and navigate to the directory where the interpreter is located.
3. Run using powershell file(run_test.ps1)
   - Paste your all test cases in to “pass” folder and then run the powershell file.

You can get normal answer and ast answers by selecting 1 or 2



Conclusion

All in all, we managed to build a lexical analyzer, parser, and turn the AST into ST. An algorithm for converting RPAL is called a conversion algorithm. The program we offer helps you handle the processing of RPAL code, make ASTs, and run the programs with the CSE use smart technologies, and ensure the outcomes are correct.

To find the repository for this project, use the link given below:

https://github.com/Muniya-64bit/RPAL_SEM4.git