CS 3513: Programming Languages

Programming Languages Project

220023U : Amarasinghe A.A.D.H.S.

220573E : Sandiw K.B.I.

1. Project Overview

This project is an RPAL (Right-reference Programming Algorithmic Language) Interpreter implemented in Python. The interpreter reads RPAL source code, performs lexical analysis (scanning), syntactic analysis (parsing), builds abstract and standardized syntax trees, generates control structures, and finally executes the code using a Control Stack Environment (CSE) machine. The system is modular, with each phase handled by a dedicated Python module, and supports outputting the AST, standardized tree, or control structures for debugging and educational purposes.

2. Technologies Used

- **Programming Language**: Python 3
- **External Libraries**: None
- **Execution Style**: Command-line interface (CLI) and Makefile-based automation

The implementation avoids third-party dependencies and implements all components (scanning, parsing, control structure generation, and evaluation) as self-contained modules, enabling full control over the entire compilation pipeline.

3. Interpreter Architecture

The interpreter is designed as a modular pipeline, consisting of five main stages:

1. **Lexical Analysis** using a custom scanner that tokenizes input.
2. **Syntactic Analysis** using recursive descent parsing to build an AST.
3. **Standardization** that rewrites the AST into a simpler, canonical form (ST).
4. **Control Structure Generation** that linearizes ST into deltas for execution.
5. **Evaluation** using a CSE (Control Stack Environment) machine, which simulates functional execution using a stack and scoped environments.

4. Development Details

4.1 Lexical Analysis (Scanner)

The scanner reads the source file character by character, identifying tokens such as identifiers, integers, keywords, operators, punctuation, and strings. It uses a recognizer

to classify character types and a reader to access file content and track line numbers. Comments (//), whitespace, and tabs are ignored. Any unrecognized characters trigger lexical errors through the ErrorHandler.

4.2 Parsing Mechanism

Parsing is implemented using recursive descent parsing directly from the RPAL grammar. The parser builds the AST from the token stream, handling precedence and associativity rules. Grammar ambiguities, particularly in distinguishing similar starting productions like Vl vs. Vb+, are resolved using LL(2)-style lookahead. Syntax errors are reported with line and character information, powered by the ErrorHandler.

4.3 AST to ST Standardization

The AST is transformed into a standardized tree (ST) using pattern-based rewrites. Constructs like let, where, within, fcn_form, and rec are systematically converted into simpler forms using lambda, gamma, and tau. This step prepares the tree for deterministic execution and simplifies the logic of the evaluation engine.

4.4 Control Structure Generation

The ST is traversed to generate a series of control structures, often referred to as deltas. These represent the program in a linearized instruction list format. The builder assigns each lambda or delta block an index and converts nested expressions into a flat sequence of evaluable components.

4.5 CSE Machine Execution

The control structures are executed using a Control Stack Environment (CSE) machine. It simulates lexical scoping, closures, and tuple-based computation using a stack and a dynamic environment list. Function application is handled using gamma, conditionals with beta, and recursion using the Y* combinator. The engine supports built-in functions such as Print, Order, Conc, Stern, and Stem.

4.6 Supporting Modules

Several utility modules underpin the interpreter:

- Token.py: Represents individual tokens with type and value.
- Node.py: Used for tree nodes in AST/ST.
- Stack.py: Stack operations for the CSE machine.
- Environment.py: Lexical scoping model using environment chaining.
- Recognizer.py: Character classification logic.
- Reader.py: Reads files and tracks line positions.
- ErrorHandler.py: Centralized error reporting.

5. Program Structure

5.1 Main Entry Point (myrpal.py)

- **Function:** Coordinates the entire workflow of the RPAL interpreter.
- **Input:**
  - Command-line arguments specifying the input file and optional output mode (-ast, -st, -cs).
- **Output:**
  - Depending on the mode, prints the Abstract Syntax Tree (AST), Standardized Tree (ST), Control Structures (CS), or the final evaluation result of the RPAL program.
- **Parameter Parsing:**
  - Parses command-line arguments to determine which mode to run (AST, ST, CS, or default evaluation).
- **Error Handling:**
  - Checks for errors after parsing and after execution; prints detailed error messages using ErrorHandler.
- **Return Values:**
  - None (outputs are printed to the terminal; exits with error code on failure).
- **Internals:**
  - Imports and initializes all major modules: Scanner, Parser, ControlStructureBuilder, and CSEMachine.
  - Manages the flow:
    1. Scans and parses the input file.
    2. Handles errors if present.
    3. Depending on flags, prints AST, ST, CS, or executes the program.
    4. Handles and prints runtime errors if they occur.

```
import sys
from Parser import Parser
from Scanner import Scanner
from ControlStructures import ControlStructureBuilder
from CSEMachine import CSEMachine

# Parse command-line arguments
# AST, ST, CS: Flags to determine which output mode to use
file_name = sys.argv[1]
AST = True if len(sys.argv) == 3 and sys.argv[2] == "-ast"
else False
ST = True if len(sys.argv) == 3 and sys.argv[2] == "-st" else
False
CS = True if len(sys.argv) == 3 and sys.argv[2] == "-cs" else
False
```

```
# Initialize parser and parse input
p = Parser(Scanner(file_name))
p.parse()

# Error handling after parsing
if p.errors.error_status:
    p.errors.print()
    sys.exit(1)
else:
    # Print the AST, ST, or CS based on the flags
    if AST:
        p.printAST()
    else:
        if ST:
            p.printST()
        else:
            p.standardize()
            c = ControlStructureBuilder(p.ST)
            if CS:
                c.printCS()
            else:
                c.linerize()
                a = CSEMachine(c.control_structures, p.errors)
                a.apply_rules()
                if a.errors.error_status:
                    p.errors.print()
                    sys.exit(1)
                else:
                    a.print()
```

5.2 Lexical Analyzer (Scanner.py)

- **Input:** Raw source code string, obtained via Reader.py.
- **Output:** A list of Token objects, each with a type (key) and value (val).
- **Parameter Parsing:** None (reads the entire input from the file passed via myrpal.py).
- **Error Handling:**
  - Detects illegal/unrecognized characters.
  - Flags malformed strings and unsupported symbols.
  - Reports errors via ErrorHandler.py, including file and line information.
- **Return Values:** get_tokens() → List[Token]
- **Internals:**
  - Uses Recognizer.py for character classification (letters, digits, operators).
  - Strips comments, tabs, and whitespace.
  - Builds tokens one character at a time, storing them for the parser.

```
class Scanner:
    def __init__(self, file_name)
    def tokenize(self)
    def handle_identifier(self)
    def handle_integer(self)
    def handle_operator(self)
    def handle_comment(self)
    def handle_space(self)
    def handle_string(self)
    def handle_punctuation(self)
    def screen(self)
    def print(self, print_token=True)
    def get_tokens(self) -> list
```

5.3 Syntax Analyzer / Parser (Parser.py)

- **Input:** List of Token objects from the Scanner.
- **Output:** Abstract Syntax Tree (AST), built as a tree of Node objects.
- **Parameter Parsing:**
  - Implements recursive descent parsing based on RPAL grammar.
  - Handles ambiguity (e.g., distinguishing between Vl and Vb+) using LL(2) lookahead.
- **Error Handling:**
  - Detects unexpected tokens, missing symbols (e.g., missing) or in).
  - Invokes ErrorHandler to show line and token context.
- **Return Values:** AST stored in self.stack[0] or returned via parse().
- **Internals:**
  - Grammar rules are implemented as methods: E(), T(), B(), etc.
  - Uses build_tree() to push/pull nodes from a stack and form the tree hierarchy.

5.4 AST Generation (inside Parser.py)

- **Input:** Tokens from the scanner (via Parser.parse()).
- **Output:** An Abstract Syntax Tree (AST) representing the full program structure.
- **Parameter Parsing:** None beyond token stream.
- **Error Handling:** Same as parser – malformed grammar results in parse errors.
- **Return Values:** Tree rooted at self.stack[0].
- **Internals:**
  - Uses the Node class to define tree structure.
  - AST is printed using printAST() with recursive traversal.

5.5 Standardized Tree (ST) Generation (inside Parser.py)

- **Input:** AST root node (self.stack[0]).
- **Output:** Standardized Tree (ST) with reduced constructs.

- **Parameter Parsing:** None; standardization logic is hard-coded.
- **Error Handling:**
  - Minimal; assumes valid AST.
  - Malformed nodes or unexpected constructs could lead to traversal errors.
- **Return Values:** self.ST (standardized tree).
- **Internals:**
  - Uses buildST() to transform constructs:
    - let → gamma + lambda
    - fcn_form → nested lambdas
    - rec → Y* combinator logic
    - Handles within, @, and other constructs.
  - Standardized tree is printed via printST().

```
class Parser:
    def __init__(self, screener)
    def printAST(self)
    def printST(self)
    def build_tree(self, transduction, n)
    def read(self, token)
    def parse(self)
    # Grammar rule methods (examples):
    def E(self)
    def Ew(self)
    def T(self)
    def Ta(self)
    def Tc(self)
    def B(self)
    def Bt(self)
    def Bs(self)
    def Bd(self)
    def D(self)
    def Da(self)
    def Dr(self)
    def Db(self)
    def Vb(self)
    def Vl(self)
    def standardize(self)
    def buildST(self, node)
    def pre_order(self)
    def printPreOrder(self, node)
```

5.6 Control Structure Generation (ControlStructures.py)

- **Input:** Standardized Tree (ST).
- **Output:** List of control structures (called “deltas”) for execution.
- **Parameter Parsing:** Tree traversal; no user input required.
- **Error Handling:**
  - Unhandled constructs or missing children may raise runtime errors.
- **Return Values:**
  - List of control structures, each indexed by function/environment ID.
- **Internals:**
  - Uses Lambda, Delta, Tau, Eta from CSComponents.py.
  - Recursively builds flat representations of the tree for interpretation.

```
class ControlStructureBuilder:
    def __init__(self, SI_Tree)
    def linerize(self)
    def printCS(self)
    def pre_order(self, root, index)
```

5.7 CSE Machine (CSEMachine.py)

- **Input:** List of control structures (deltas).
- **Output:** Final evaluation result (printed or returned).
- **Parameter Parsing:** None (internal control loop).
- **Error Handling:**
  - Stack underflows.
  - Type errors (e.g., applying + to strings).
  - Division by zero.
  - Unbound variable references.
- **Return Values:** Final value pushed onto the stack, printed if Print is used.
- **Internals:**
  - Uses a stack (Stack.py) and dynamic environments (Environment.py).
  - Supports:
    - Function application via gamma.
    - Conditionals via beta.
    - Recursion via Y*.
    - Built-in functions like Print, Order, etc.

```
class CSEMachine:
    def __init__(self, control_structure, errors)
    def lookup(self, var)
    def apply_rules(self)
    def print(self)
```

5.8 Supporting Modules

Recognizer.py

- **Function:** Classifies characters (is_letter, is_digit, is_operator, etc.).
- **Used In:** Scanner.
- **Return Values:** Boolean values for character type checks.

```
class Recognizer:
    def __init__(self)
    def is_digit(self, char)
    @staticmethod
    def is_space(char)
    def is_punctuation(self, char)
    def is_letter(self, char)
    def is_operator(self, char)
    @staticmethod
    def is_eol(char)
    @staticmethod
    def is_underscore(char)
    @staticmethod
    def is_slash(char)
    @staticmethod
    def is_back_slash(char)
    def is_after_back_slash(self, char)
    @staticmethod
    def is_double_quote(char)
    @staticmethod
    def is_single_quote(char)
    @staticmethod
    def is_ht(char)
```

Reader.py

- **Function:** Reads the source file as a string and tracks line/char positions.
- **Input:** File path.
- **Output:** Full file string and list of lines.
- **Used In:** Scanner.
- **Error Handling:** File not found / read errors.

```
class Reader:
    def __init__(self, filename)
    def read_whole(self)
    def find_line(self, index)
    def read_lines(self)
    def get_line(self, line_number)
```

Token.py

- **Function:** Represents lexical tokens.
- **Structure:** Token(key, val) where key = type, val = literal value.
- **Used In:** Scanner → Parser.

```
class Token:
    def __init__(self, typeClass, value)
    def __str__(self)
    def __repr__(self)
```

Node.py

- **Function:** Tree node representation for AST and ST.
- **Attributes:** data, children.
- **Used In:** Parser, Standardizer, CS builder.

```
class Node:
    def __init__(self, data)
    def add_child(self, child)
    def add_child_end(self, child)
    def remove_child(self, child)
    def __str__(self)
    def __repr__(self)
```

Stack.py

- **Function:** Implements stack operations (push, pop, peek).
- **Used In:** CSEMachine for managing evaluation stack.

```
class Stack:
    def __init__(self)
    def push(self, item)
    def pop(self)
    def is_empty(self)
    def size(self)
    def peek(self)
    def print_stack(self)
    def get_items(self)
```

Environment.py

- **Function:** Manages variable scopes and function closures.
- **Structure:** Linked environments with parent-child access.
- **Used In:** CSEMachine during lambda application.

```
class Environment:
    def __init__(self, number, parent)
    def add_child(self, child_env)
    def add_variable(self, key, value)
    def __str__(self)
    def __repr__(self)
```

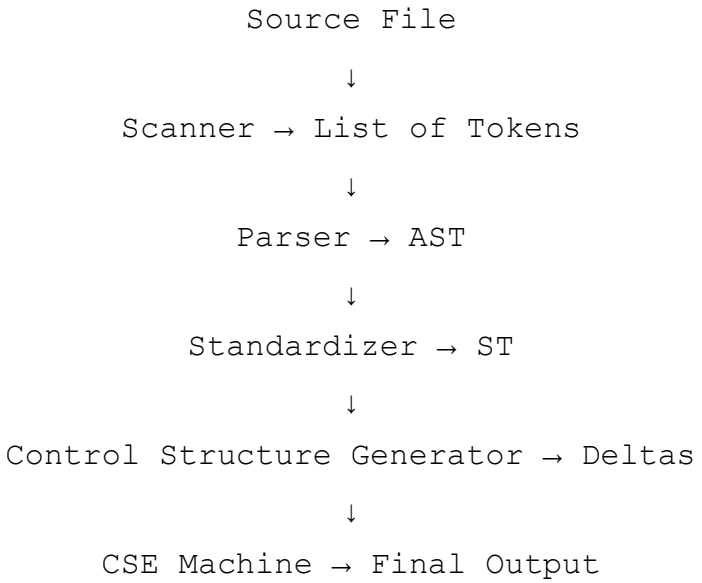
ErrorHandler.py

- **Function:** Centralized error reporting.
- **Features:**
  - Colored CLI output (ANSI).
  - Detailed messages for lexical, parsing, and runtime errors.
- **Used In:** Scanner, Parser, CSEMachine.

```
class ErrorHandler:
    def __init__(self, reader=None, source_list=None)
    def syntax_error(self, index)
    def unrecognized_error(self, index)
    def parse_error(self, error)
    def unsupported_operands(self, operation, types)
    def zero_division_error(self, operand1)
    def print(self)
```

6. Execution Flow

The system follows the high-level flow shown below:



7. How to Use

7.1 CLI Usage

```
# Executes the RPAL program
python3 myrpal.py filename.txt
# Outputs the Abstract Syntax Tree
python3 myrpal.py filename.txt -ast
# Outputs the Standardized Tree
python3 myrpal.py filename.txt -st
# Outputs the Control Structures
python3 myrpal.py filename.txt -cs
```

7.2 Makefile Usage

```
run:
    python3 myrpal.py $(file)

# Print Abstract Syntax Tree
ast:
    python3 myrpal.py $(file) -ast

# Print Standardized Tree
st:
    python3 myrpal.py $(file) -st

# Print Control Structures
cs:
    python3 myrpal.py $(file) -cs
```

8. Conclusion

The RPAL interpreter successfully implements every major component of a functional language pipeline — from scanning and parsing to standardization and evaluation. The interpreter handles non-trivial expressions, function applications, recursion, and built-in functions, offering an educationally valuable and technically sound implementation of a real interpreter. The modular design and clean CLI output also make it suitable for debugging, extension, and demonstration purposes in future projects or academic contexts.