CS 3513: Programming Languages

Programming Project 01 - RPAL Interpreter

G.M.A.M Abhayawickrama - 220011G

1. Introduction

This project implements an interpreter for the RPAL programming language as part of CS 3513. It includes a lexical analyzer, a parser to generate the Abstract Syntax Tree (AST), a standardizer to convert the AST into a Standardized Tree (ST), and a CSE Machine for evaluation. The interpreter is written in Python, does not need any specific libraries and replicates the output format of rpal.exe. It accepts input from a text file and supports flags like -ast, -st to display things as ASTs and STs.

2. Project Structure

The RPAL interpreter is implemented in Python using a modular design. Each module corresponds to a core phase of interpretation: lexical analysis, parsing, standardization, and execution.

2.1 File Structure

```
myrpal.py                   # Main entry point of the interpreter
utils/
├── file_io.py              # Handles reading input files and writing outputs
└── node.py                 # Defines the Node class used in AST/ST representation
Lexer/
└── lexer.py                # Lexical analyzer for tokenizing RPAL code
Parser/
└── parser.py               # Recursive descent parser that generates AST
Standardizer/
└── standardizer.py         # Converts AST to a standardized tree (ST)
flattener/
└── flat.py                 # Flattens the ST into CSE-compatible control structures
CSE_Machine/
└── cse_machine.py          # Simulates the CSE Machine for evaluating the program
```

3. Module Documentation

3.1 Lexical Analyzer

A. Overview

- The lexical analyzer is implemented in Lexer/lexer.py. It processes the raw RPAL source code obeying rpal lexical grammar and breaks it into a sequence of tokens, skipping over comments and whitespace. The recognized tokens follow the RPAL lexical specifications, including identifiers, keywords, integers, strings, operators, and punctuation.
- The lexer uses regular expressions to define token patterns and the re module to scan the source code. Tokens are returned as instances of the MyToken class, each labeled with a TokenType.

B. Function Prototypes

```
class TokenType(Enum):
    KEYWORD = 1
    IDENTIFIER = 2
    INTEGER = 3
    STRING = 4
    OPERATOR = 5
    PUNCTUATION = 6
    END_OF_TOKENS = 7

class MyToken:
    def __init__(self, token_type, value):
        if not isinstance(token_type, TokenType):
            raise ValueError("Token_Type not recgnized")
        self.type = token_type
        self.value = value

def tokenize(code: str):
    tokens = []
    for mo in master_pattern.finditer(code):
        kind = mo.lastgroup
        value = mo.group()

        if kind in ('SPACES', 'COMMENT'):
            continue  # Skip ignored tokens

        if kind not in TokenType.__members__:
            raise ValueError(f"Unknown token kind: {kind}")

        token_type = TokenType[kind]
        tokens.append(MyToken(token_type, value))

    tokens.append(MyToken(TokenType.END_OF_TOKENS, '$'))
    return tokens
```

C. Implementation Details

- Token Types :
    - The TokenType enum classifies all possible token categories used in RPAL.
- MyToken Class :
    - A custom token object that holds the token's type and string value.

- Token_specification :
    - A list of regular expressions matching RPAL's lexical components including keywords, identifiers, integers, strings, operators, and punctuation.
- tokenize() Function :
    - Input: A string containing RPAL source code.
    - Output: A list of MyToken objects, ending with a special END_OF_TOKENS token.
    - Ignored: Comments (//) and whitespace are skipped.
    - Error Handling: Raises a ValueError if an unknown token is encountered.

D. Example Tokenization

- **For input:** let x = 5 in x + 1
- **The tokenize() function would return:**[KEYWORD: let, IDENTIFIER: x, OPERATOR: =, INTEGER: 5, KEYWORD: in, IDENTIFIER: x, OPERATOR: +, INTEGER: 1, END_OF_TOKENS: $]

3.2 Parser

A. Overview

- The parser, implemented in Parser/parser.py, performs **recursive descent parsing** on the token stream generated by the lexical analyzer. It constructs a hierarchical Abstract Syntax Tree (AST) based on the grammar rules defined in RPAL_Grammar.pdf.
- The parser operates directly on MyToken objects and recursively matches grammar productions. It handles precedence and associativity using separate methods for each grammar non-terminal, enabling accurate construction of the AST.

B. Function Prototypes

```
class ASTNode:
    def __init__(self, label, children=None):
        self.label = label
        self.children = children or []

    def __repr__(self):
        return f"<{self.label}>"

    def print_ast(self, level=0):
        print('.' * level + self.label)
        for child in self.children:
            child.print_ast(level + 1)
```

```python
class Parser:
    def __init__(self, tokens):
        self.tokens = tokens
        self.pos = 0

    def current_token(self):
        return self.tokens[self.pos] if self.pos < len(self.tokens) else None

    def peek_token(self, offset=1):
        target_pos = self.pos + offset
        return self.tokens[target_pos] if target_pos < len(self.tokens) else None

    def match(self, expected_type=None, expected_value=None):
        token = self.current_token()
        if token and ((expected_type is None or token.type == expected_type) and
                      (expected_value is None or token.value == expected_value)):
            self.pos += 1
            return token
        raise SyntaxError(f"Unexpected token: {token}, expected {expected_value or expected_type}")

    '''
    # Tuple Expressions ####################################
        T   -> Ta ( ',' Ta )+ => 'tau'
            -> Ta;
        Ta  -> Ta 'aug' Tc => 'aug'
            -> Tc;
    '''
    def parse_tuple(self):
        ta_node = self.parse_tuple_aug()
        # Check if there's a comma, indicating a tuple
        if self.current_token() and self.current_token().value == ',':
            tas = [ta_node]
            while self.current_token() and self.current_token().value == ',':
                self.match(TokenType.PUNCTUATION, ',')
                tas.append(self.parse_tuple_aug())
            return ASTNode('tau', tas)
        return ta_node
```

C. Implementation Details

The parser in Parser/parser.py is a recursive descent parser that builds an Abstract Syntax Tree (AST) from the list of tokens produced by the lexer. Each grammar rule is implemented as a separate method (e.g., parse_expr, parse_definition, parse_arithmetic), allowing the parser to handle RPAL syntax accurately.

- The Parser class manages the token stream and provides helper methods like match() and peek_token() for token consumption and lookahead.
- Operator precedence and associativity are handled through function layering (e.g., parse_arithmetic, parse_arithmetic_term, parse_arithmetic_factor).
- AST nodes are constructed using the ASTNode class, with clear labels (e.g., let, +, <ID:x>).
- The parser supports complex RPAL constructs including let, fn, rec, within, conditionals, and tuples.
- Error handling is included to detect unexpected or malformed syntax.
- Gamma expressions are formed for function application, supporting nested calls like gamma(gamma(f, a), b)..

D. Inputs and Outputs

- **Input**: A list of MyToken objects produced by the lexer. These represent the lexical tokens of an RPAL source program.

- **Output**: A tree of ASTNode objects, with each node representing a syntactic construct in the RPAL grammar.
- The final output is the root of the Abstract Syntax Tree (AST), which reflects the program structure and is passed to later stages like the standardizer.

3.3 Standerdizer

A. Overview

- The Standardizer, implemented in Standardizer/standardizer.py, transforms the Abstract Syntax Tree (AST) into a Standardized Tree (ST) based on RPAL's pictorial transformational grammar. It rewrites syntactic sugar and high-level constructs into a normalized form using only core RPAL primitives like lambda, gamma, and Y*.

B. Function Prototypes

C. Implementation Details

```python
def standardize(node: ASTNode) -> ASTNode:
    """
    Complete standardization function for RPAL AST based on the pictorial grammar.
    Transforms syntactic sugar into standard forms using lambda calculus primitives.
    """
    label = node.label
    children = node.children

    # Terminal nodes (IDs, INTs, STRs) - no transformation needed
    if not children:
        return ASTNode(label)

    def std(child):
        """Helper to recursively standardize child nodes"""
        return standardize(child)

    # let X = E1 in E2 => gamma(lambda X. E2, E1)
    if label == "let":
        binding = std(children[0])  # = X E1
        e2 = std(children[1])       # E2
        x = binding.children[0]     # X
        e1 = binding.children[1]    # E1
        lam = ASTNode("lambda", [x, e2])
        return ASTNode("gamma", [lam, e1])
```

- The standardizer handles all RPAL constructs, including:
    - The standardize() function **recursively** transforms the AST into its **fully standardized** form.
    - All non-leaf nodes are converted into **lambda** or **gamma** expressions, aligning with RPAL's pictorial grammar.
    - Constructs like let, where, rec, within, fn, and conditionals (->) are rewritten using combinations of lambda, gamma, and <Y*>.
    - Function definitions are converted into nested lambda forms.
    - Binary and unary operators (e.g., +, *, not, neg) are transformed into curried gamma applications.

- This results in a **fully standardized** tree ready for evaluation by the CSE Machine.

D. Inputs and Outputs

- **Input**: A tree of ASTNode objects representing the RPAL Abstract Syntax Tree (AST).
- **Output**: A fully transformed ASTNode tree representing the Standardized Tree (ST), suitable for execution by the CSE Machine.

*Special Note on Full Standardization

Unlike the default rpal.exe interpreter, which does not fully standardize certain operations (e.g., it leaves + as a binary operator), this implementation applies all transformations strictly according to the RPAL pictorial standardization grammar.
As a result, the Standardized Tree (ST) produced by this interpreter may look more transformed or expanded than the one shown by the official tool.
This includes converting:

- All binary operators are converted into curried gamma expressions
- All unary operators into gamma form
- Function definitions into lambda structures
- Conditionals and recursive bindings into compositional gamma + lambda + Y* structures

3.3 Flattener

A. Overview

The flattener converts the Standardized Tree (ST) into a control structure—a linear sequence of operations—used by the CSE Machine for execution. This process is implemented in two versions:

- **STFlattener**: A basic version that flattens the ST following direct post-order traversal.
- **OptimizedFlattener**: A rule-based optimizer that applies special-case compressions as defined in CSE Rules 6–11 for improved execution efficiency.

Both flatteners follow the mechanical rules described in the RPAL CSE Machine documentation, including the use of control identifiers like γ, λ, τn, β, and δi.

B. Implementation Details

When the flatten() function is called with a Standardized Tree (ST) as input, the flattener performs the following steps:

- Initialization
  - A control structure dictionary is created (e.g., δ0, δ1, ...)
  - A counter is used to assign unique IDs to nested control blocks (lambda, then, else)
- Tree Traversal (Post-Order Reversal)
  - The flattener performs a post-order traversal of the tree, meaning:
    - It recursively processes children first

        - Then adds the current node's operation last
    - This order ensures the generated instructions can be pushed directly onto the stack for correct evaluation in the CSE Machine
- Node Conversion
    - Each node is converted based on its type:
- Building Control Structures
    - Each sub-expression (e.g., body of a lambda, then/else branch) gets its own control structure (δi) recursively.
    - These are then stored in a dictionary.

C. Function Prototypes

```
class OptimizedFlattener:
    def __init__(self):
        self.control_counter = 1
        self.control_structures = {}

    def flatten(self, node: ASTNode) -> dict:
        self.control_counter = 1
        self.control_structures = {}
        self.control_structures[0] = self._generate_control(node)
        return self.control_structures

def _generate_control(self, node: ASTNode) -> list:
        if not node:
            return []

        label = node.label
        children = node.children
        control = []

        # Terminal
        if not children:
            return [self._extract_terminal_value(label)]
        # Gamma
        if label == 'gamma':
            left = children[0]
            right = children[1]

            # Detect curried binary ops: gamma(gamma(op, x), y)
            if (left.label == 'gamma' and len(left.children) == 2 and
                left.children[0].label in {'+', '-', '*', '/', '**',
'eq', 'ne', 'gr', 'ge', 'ls', 'le', 'aug'}):
                op = left.children[0].label
                x = left.children[1]
                y = right
                control += self._generate_control(x)
                control += self._generate_control(y)
                control.append(op)
                return control
```

D. Inputs and Outputs

- **Input**: A fully standardized RPAL tree (ASTNode), where all constructs have been transformed into lambda, gamma, and related primitive forms.
- **Output**: A **dictionary of control structures**, where each control structure is a **list of instructions** in stack-execution order.

Special Note on Flatenner

- In my implementation, the **OptimizedFlattener** supports all enhancements from the RPAL CSE Machine's mechanical evaluation rules (as documented), and generates more efficient control structures. These include reductions for unary and binary operators, conditional blocks, and tuple handling.
- This flattened output may differ slightly from basic gamma-style expansion, but is **functionally equivalent** and better aligned with how the CSE Machine operates in practice.

3.3 CSE Machine

A. Overview

The CSEMachineExecutor, implemented in CSE_Machine/cse_machine.py, evaluates the flattened control structures generated from the standardized tree. It uses a Control-Stack-Environment (CSE) model to simulate functional execution using closures, tuples, conditionals, and built-in functions.

B. Function Prototypes

```python
class Environment:
    def __init__(self, index=0, parent=None):
        self.index = index
        self.bindings = {}
        self.parent = parent
        self.is_removed = False

    def lookup(self, var):
        if var in self.bindings:
            return self.bindings[var]
        elif self.parent:
            return self.parent.lookup(var)
        raise NameError(f"Unbound identifier: {var}")

    def extend(self, var, value):
        self.bindings[var] = value

    def set_removed(self, removed):
        self.is_removed = removed

    def get_removed(self):
        return self.is_removed

class Closure:
    def __init__(self, params, delta_id, env_index):
        self.params = params  # list of variable names
        self.delta_id = delta_id
        self.env_index = env_index  # Store environment index instead of reference

    def __repr__(self):
        return f"<Closure λ{','.join(self.params)}^{self.delta_id}>"

class Eta:
    def __init__(self, closure):
        self.closure = closure  # The original closure from Y*

    def __repr__(self):
        return f"<Eta {self.closure}>"
```

```python
class CSEMachineExecutor:
    def __init__(self, control_structures):
        self.control_structures = control_structures
        self.stack = []
        self.environments = [Environment(0)]  # List of environments
        self.current_env = self.environments[0]  # Current active environment
        self.control = list(reversed(control_structures[0]))  # Start from δ0
        self.env_counter = 1
        self.trace_log = []
        self.builtins = {
            'Print', 'Isinteger', 'Isstring', 'Istuple', 'Isdummy',
            'Istruthvalue', 'Isfunction', 'Stem', 'Stern', 'Conc',
            'Order', 'Null',
        }

    def run(self):
        steps = 0
        MAX_STEPS = 100000  # Increased for deep recursion
        self.trace_log.clear()

        while self.control:
            steps += 1
            if steps > MAX_STEPS:
                print("🔁 Execution stopped: exceeded maximum steps (possible infinite loop).")
                print(f"Top of stack: {self.stack[-1] if self.stack else 'empty'}")
                break

            instr = self.control.pop()
            # self.print_state(instr)  # Debug info
            self.record_state(instr)

            if instr.isdigit():
                self.stack.append(instr)

            elif instr.startswith('λ'):
                lambda_header = instr[1:]  # e.g. x,y^1
                param_part, delta_part = lambda_header.split('^')
                params = param_part.split(',')
                delta_id = int(delta_part)
                closure = Closure(params, delta_id, self.current_env.index)
                self.stack.append(closure)
```

C. Implementation Details

- Initialization:
  - The machine starts by reversing δ0 and pushing it onto the control list. An initial environment (e0) is created.
- Execution Loop:
  - The machine pops instructions from the control list and processes them.
  - Maintains a stack and an environment chain.
  - Supports:
    - γ (gamma): for function application
    - λ (lambda): creates closures
    - β (beta): handles conditionals
    - τn (tau): constructs tuples
    - δi (delta): jumps to stored control blocks
    - Built-ins like Print, Isinteger, Order, Conc, etc.
- Closures and Recursion:
  - Lambdas are turned into closures with captured environments.
  - Recursion is supported via the <Y*> combinator, which generates an Eta node that self-applies.
- Tuple Support:
  - Tuples are simulated using lists.

    - Supports indexed access using gamma with numeric arguments.
- Built-in Function Application:
    - Built-in names are stored on the stack and handled in gamma when applied to arguments.
    - Includes type-checkers (Isstring, Istuple), string manipulators (Stem, Stern), and logical functions.
- Environment Management:
    - New environments are created when lambdas are applied.
    - Environments are removed after evaluation using special env_remove_i instructions.
- Trace and Debugging:
    - The machine supports internal tracing (trace_log) to record each step for debugging or visualization.

This implementation fully supports deep recursion, tuples, multi-argument lambdas, and all built-in RPAL functions. It also correctly handles eta expansion for Y*-based recursion and maintains a flexible environment stack using Environment objects.

D. Inputs and Outputs

- **Input**: A dictionary of control structures (δ0, δ1, ...) generated by the flattener. Ex:

```
-{
   0: ['5', 'λx^1', 'γ'],
   1: ['x', '1', '+']
}
```

- **Output**: The final result of executing the RPAL program — typically a number, a boolean, a string, or a tuple.
    - Also prints intermediate results if Print is used.
    - Returns the top of the stack after completion.

3.3 Main Program - myrpal.py

E. Overview

The myrpal.py script serves as the **entry point** of the RPAL interpreter. It coordinates all phases: **tokenization, parsing, standardization, flattening, and execution using the CSE Machine**. It also supports command-line flags for debugging and visualization.

F. Function Prototypes

G. Execution Workflow

- **Lexical Analysis**: Reads the input file and tokenizes it using Lexer/lexer.py.
- **Parsing**: Builds the AST using the recursive descent Parser.

- **Standardization**: Converts the AST into a normalized standardized tree via the standardizer.
- **Flattening**: Generates an optimized control structure.
- **Execution**: Passes the optimized control structure to the CSEMachineExecutor, which interprets the RPAL program.
- **Optional Flags**:
    - ➢ -ast: Print Abstract Syntax Tree
    - ➢ -st: Print Standardized Tree
    - ➢ -flat: Show unoptimized control structure
    - ➢ -optflat: Show optimized control structure
    - ➢ -cse: Show detailed CSE Machine execution trace
    - ➢ -allt: Print both AST and ST
    - ➢ -h: Show help message

H. Inputs and Outputs

- **Input**: A fully standardized RPAL tree (ASTNode), where all constructs have been transformed into lambda, gamma, and related primitive forms.
- **Output**: A **dictionary of control structures**, where each control structure is a **list of instructions** in stack-execution order.

4. How to Run

The interpreter can be run in two ways:

1. Using Python Command

i. Open terminal and navigate to the project directory.
ii. Run the following commands:

```
python myrpal.py input.txt            # Execute program
python myrpal.py input.txt -ast       # Show AST
python myrpal.py input.txt -st        # Show Standardized Tree
```

2. Using Makefile

1. Open a terminal and navigate to the project directory.
2. Use the following:

```
make run file=input.txt      # Run program
make ast file=input.txt      # Show AST
make st file=input.txt       # Show Standardized Tree
```

5. Testing

- A custom test script test_rpal.py is included to automate validation of the RPAL interpreter. It supports detailed output, filtering, single test execution, and automatic report generation.You can edit the test_cases dictionary at the top of the script to add your own test programs.
- Navigate to the project directory and use one of the following commands:

```
python test_rpal.py                  # Run all tests (summary only)
python test_rpal.py --details        # Run all tests with AST, ST, and control view
python test_rpal.py --single <name>  # Run a single named test
python test_rpal.py --filter <key>   # Run tests matching keyword(s)
python test_rpal.py --report         # Run and generate a full test report
```

6. Conclusion

This project successfully implements a complete RPAL interpreter from scratch, covering lexical analysis, parsing, AST standardization, control structure flattening, and execution via a CSE Machine. By following the RPAL grammar and evaluation rules closely, the system accurately simulates functional language behavior, including recursion, tuples, and built-in operations. The modular design, along with support for both standard and optimized execution paths, ensures clarity, extensibility, and performance.