RPAL Language Interpreter

Project Report

CS 3513 - Programming Languages

Group Members

- 220277B - Jayathunga R.D.S.S
- 220507H – Ranathunga W.K.P

Table of Contents

3. Introduction

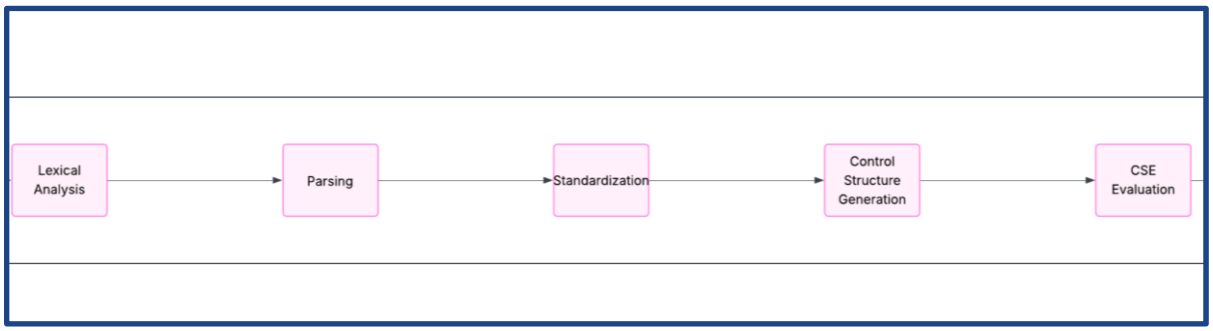
3.1 Purpose of the project

The RPAL Language Interpreter project aims to develop a complete interpreter for the Right-Reference Pedagogical Algorithmic Language (RPAL), a functional programming language designed for educational purposes. The primary objective is to process RPAL source code and produce correct output by implementing a pipeline that includes lexical analysis, parsing, standardization, control structure generation, and evaluation using a Control-Stack-Environment (CSE) machine. This project provides a practical understanding of compiler design principles, including tokenization, abstract syntax tree (AST) construction, tree transformation, and runtime evaluation, while adhering to the formal specifications of RPAL as defined in the course materials.

The interpreter can be used in a number of ways, such as producing the final evaluated output (default mode), a standardized tree (-st flag), or an AST (-ast flag). This flexibility makes it easier for users to debug and comprehend the compilation process by enabling them to examine intermediate representations of the program. In order to ensure precise execution of complex expressions, the project also attempts to handle RPAL's functional constructs, including let bindings, lambda abstractions, and predefined functions like Print and Order.

3.2 Scope

Lexical Analysis → Parsing (AST) → Standardization (ST) → Control Structure → CSE Evaluation



1. **Lexical Analysis**: Converts RPAL source code into a stream of tokens, identifying keywords, identifiers, operators, and literals while handling whitespace and comments appropriately. This is implemented in myrpal.py.

2. **Parsing**: Constructs an Abstract Syntax Tree (AST) from the token stream using a recursive descent parsing strategy based on the RPAL grammar (RPAL_Grammar.pdf). This is implemented in parser.py.

3. **Standardization**: Transforms the AST into a Standardized Tree (ST) by applying transformation rules (e.g., converting let to gamma and lambda) to simplify evaluation. This is handled in standardizer.py.

4. **Control Structure Generation**: Generates control structures (e.g., Lambda, Delta, Tau) from the standardized tree, setting up the environment for evaluation. This is implemented in structure.py and Environment.py.

5. **CSE Evaluation**: Executes the program using a CSE machine, managing control, stack, and environment to produce the final output. This is also implemented in csemachine.py

3.3 Language and tools used

The interpreter is implemented entirely in **Python 3**, leveraging its flexibility and robust standard library for file handling, regular expressions, and data structure management. No external lexical analyzer or parser generators (e.g., lex or yacc) were used, ensuring a fully custom implementation that provides deeper insight into the compilation process. Key Python modules utilized include:

- re - for regular expression-based tokenization in myrpal.py.
- sys - for command-line argument handling and file input in myrpal.py.
- Dataclasses - for defining the Token class.

To manage the AST, control structures, stack operations, and environment handling, respectively, the project uses custom classes defined in ASTnode.py, structure.py, stack.py, and environment.py. Compatibility with standard RPAL programs is ensured by the implementation's adherence to the RPAL language specification and grammar given in the course materials.

4. Program Structure

Lexical analysis, parsing, standardization, control structure generation, and CSE machine evaluation are the various components of the interpretation pipeline that are handled by the modular collection of Python files that make up the RPAL interpreter. Data structures like the AST, stack, and environments are managed by supporting utilities. Each file's description, goal, and essential elements are listed below, followed by succinct function prototypes or class diagrams that show how they are organized.

4.1 File descriptions

1. **ASTnode.py**:
    - **Purpose**: Defines the ASTnode class, used to represent nodes in the Abstract Syntax Tree (AST) and Standardized Tree (ST).
    - **Key Components**: The ASTnode class stores the node type, value (for terminal nodes), and a list of child nodes, supporting both terminal (e.g., <ID:x>) and non-terminal (e.g., let) nodes.
    - **Usage**: Used by the parser (parser.py) to build the AST, by the standardizer (standerdizer.py) to transform it into an ST, and by the CSE machine (csemachine.py) for control structure generation.
2. **parser.py**:
    - **Purpose**: Implements the recursive descent parser, converting a token stream into an AST based on the RPAL grammar.
    - **Key Components**: The Parser class contains methods like parse_E, parse_D, and buildTree, which recursively process tokens and construct the AST using a global stack.
    - **Usage**: Called by myrpal.py to parse the token stream, producing an AST for further processing.
3. **standerdizer.py**:
    - **Purpose**: Transforms the AST into a Standardized Tree (ST) by applying transformation rules (e.g., converting let to gamma and lambda).
    - **Key Components**: The standardize function recursively processes ASTnode objects, handling constructs like let, where, rec, and fcn_form.
    - **Usage**: Invoked by myrpal.py when the -st flag is used or before CSE evaluation to prepare the ST.
4. **csemachine.py**:
    - **Purpose**: Generates control structures from the ST and evaluates them using the CSE machine to produce the final output.
    - **Key Components**: Functions like generate_control_structure, apply_rules, and run_cse_machine manage control structure creation and evaluation, with support for built-in functions (e.g., Print, Order).
    - **Usage**: Called by myrpal.py for default execution mode to evaluate the program.
5. **Environment.py**:
    - **Purpose**: Defines the Environment class, managing variable bindings and scoping during CSE evaluation.
    - **Key Components**: Stores environment number, name (e.g., e_0), variables, parent, and children, enabling hierarchical lookup.
    - **Usage**: Used by csemachine.py to track variable bindings during lambda applications and evaluations.

6. **stack.py**:
    - **Purpose**: Implements the Stack class, used for both AST construction (parser) and CSE evaluation.
    - **Key Components**: Supports push, pop, is_empty, and indexing operations, with error handling for empty stacks.
    - **Usage**: Used by parser.py for AST building and by csemachine.py for evaluation stack management.
7. **structure.py**:
    - **Purpose**: Defines classes for control structures (Delta, Tau, Lambda, Eta) used in CSE evaluation.
    - **Key Components**: Each class represents a specific control structure, storing attributes like number, body, or bounded variables.
    - **Usage**: Used by csemachine.py to create control structures during generation.
8. **myrpal.py**:
    - **Purpose**: Serves as the main entry point, performing lexical analysis and orchestrating the interpretation pipeline.
    - **Key Components**: The tokenize function converts source code to tokens, and the main function coordinates parsing, standardization, and evaluation based on command-line flags (-ast, -st, or default).
    - **Usage**: Executed directly to run the interpreter (python myrpal.py [flag] file_name).

5. Lexical Analyzer

5.1 Objective

The lexical analyzer is responsible for converting RPAL source code into a stream of tokens, which serves as the input for the parser. This process involves scanning the input code, identifying lexical units such as keywords, identifiers, operators, integers, strings, and punctuation, and producing a structured token stream that adheres to the RPAL language specification. The lexical analyzer ensures that the subsequent parsing phase receives well-formed tokens, enabling accurate syntax analysis and AST construction.

5.2 Description of rules used

The lexical analyzer, implemented in myrpal.py, follows the tokenization rules outlined in the RPAL language specification. It uses regular expressions to define token patterns in priority order, ensuring correct classification of lexical units. The token specifications are defined as follows.

- **Comments**: Single-line comments starting with // are ignored (r'**//.***').
- **Whitespace**: Spaces, tabs, and newlines are skipped (r'**[\t\r\n]+**').
- **Strings**: Sequences enclosed in single quotes, supporting escaped characters like \', \\, \t, and \n (r"**'(\\'|\\\\|\\t|\\n|[^'])*'**").

- **Identifiers**: Alphanumeric sequences starting with a letter, possibly including underscores (r'**[A-Za-z][A-Za-z0-9_]***').
- **Integers**: Sequences of digits (r'**\d+**').
- **Operators**: Symbols such as +, -, *, /, <, >, =, @, etc. (r'**[\+\-*\/<>&\.@:=~|$!#%^_\[\]{}\"'\?]+**').
- **Punctuation**: Characters like (,), ,, and ; (r'[(),;]').
- **Mismatches**: Any unrecognized characters trigger an error (r'**.**').

A predefined set of RPAL keywords (e.g., let, fn, where, true, false, nil) is maintained to distinguish them from identifiers. The regular expressions are combined into a single pattern using Python's re module, and the token_re.finditer method processes the input code to generate tokens.

5.3 Output format of tokens

The lexical analyzer produces a list of Token objects, each defined using Python's dataclasses module in myrpal.py. Each Token has two attributes:

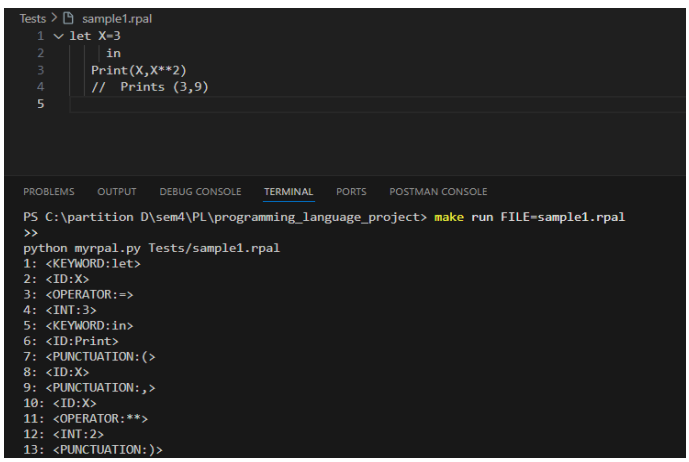
- type: A string indicating the token category (e.g., KEYWORD, ID, INT, STR, OPERATOR, PUNCTUATION).
- value: The actual string value of the token (e.g., let for a keyword, x for an identifier, 42 for an integer).

The string representation of a token is formatted as <type:value>, for example,

- <KEYWORD:let> for the keyword let.
- <ID:x> for the identifier x.
- <INT:42> for the number 42.
- <STR:'hello'> for the string 'hello'.

This format is used for debugging and can be printed to verify the token stream before parsing.

Example Code And The Relevent Output -



5.4 Challenges and handling of whitespace/comments/strings

1. **Whitespace and Comments** -
    - Whitespace (spaces, tabs, newlines) and single-line comments (//) are skipped to avoid cluttering the token stream. The regular expressions r'[\t\r\n]+' and r'//.*' ensure these are ignored without generating tokens.
    - **Challenge** - Ensuring comments do not interfere with token recognition, especially when they appear mid-line. The priority order in token_specification ensures comments are matched before other tokens.
2. **String Handling** -
    - Strings in RPAL are enclosed in single quotes and may contain escaped characters (e.g., \n, \t). The regular expression r"'(\\'|\\\\|\\t|\\n|[^'])*'" captures the entire string, including its delimiters, while preserving escaped sequences.
    - **Challenge**: Correctly handling escaped quotes (\') and backslashes (\\) within strings. The regex pattern accounts for these by explicitly including them in the match, preventing premature termination of string tokens.
3. **Keyword vs. Identifier Differentiation** -
    - Identifiers that match RPAL keywords (e.g., let, fn) must be classified as KEYWORD tokens rather than ID. This is handled by checking the token value against the KEYWORDS set in myrpal.py.
    - **Challenge**: Ensuring efficient lookup to avoid misclassifying keywords as identifiers. The use of a Python set for KEYWORDS provides constant-time lookup.
4. **Unexpected Characters** -
    - Any character not matching a defined pattern is caught by the MISMATCH rule, raising a RuntimeError with a descriptive message (e.g., Unexpected character: #).
    - **Challenge**: Providing clear error messages to aid debugging. The error includes the offending character and its position, making it easier to locate issues in the source code.

5.5 Function prototype

The core lexical analysis functionality is encapsulated in the tokenize function in myrpal.py. Its prototype is,

```
def tokenize(code: str) -> list[Token]:
```

- **Input**: A string containing the RPAL source code.
- **Output**: A list of Token objects, each representing a lexical unit with type and value attributes.

- **Implementation Details**:
    - Uses re.compile to create a single regular expression from token_specification.
    - Iterates over matches using token_re.finditer, processing each match based on its group (e.g., COMMENT, ID).
    - Skips COMMENT and SKIP matches, converts identifiers to keywords when applicable, and raises errors for mismatches.

The Token class, defined using @dataclass, is:

```
@dataclass
class Token:
    type: str
    value: str
```

This class provides a clean and efficient way to represent tokens, with a __str__ method for debugging output.

6. Parser

6.1 Objective

An essential part of the RPAL interpreter, the parser is in charge of converting a stream of tokens generated by the lexical analyzer into an Abstract Syntax Tree (AST) that depicts the RPAL program's syntactic structure. The generated AST correctly reflects the hierarchical relationships between language constructs like expressions, declarations, and operators because the parser complies with the RPAL grammar as defined in RPAL_Grammar.pdf. The AST that is produced is the starting point for the following stages of standardization and assessment.

6.2 Parsing strategy

The parser employs a **recursive descent parsing** strategy, a top-down approach that directly implements the RPAL grammar through a set of recursive functions. Each function corresponds to a non-terminal in the grammar (e.g., E, T, D), systematically consuming tokens and constructing the AST by pushing nodes onto a stack. The recursive descent method was chosen for its clarity and alignment with the hierarchical structure of the RPAL grammar, allowing straightforward handling of nested expressions and declarations.

The parser, which is implemented in parser.py, keeps a global stack (stack) to hold ASTnode objects and initializes with a token list. It makes sure the tree reflects the grammar's production rules by using a buildTree method to create AST nodes with the right number of children and a read method to consume tokens. Error handling is integrated to provide informative error messages with token positions and detect syntax errors, such as unexpected token types or values.

6.3 Example input and generated AST

Consider the following RPAL program as an example input,

```
let x = 5 in x + 3
```

Running the interpreter with the -ast flag (python myrpal.py -ast test.rpal) produces the following AST, printed using the print_ast function in myrpal.py.

```
let
.x
..<ID:x>
..<INT:5>
.+
..<ID:x>
..<INT:3>
```

Explanation:

- The root node is let, representing the let-expression.
- The first child is an = node, with children <ID:x> and <INT:5>, representing the binding x = 5.
- The second child is a + node, with children <ID:x> and <INT:3>, representing the expression x + 3.

This AST captures the structure of the let-expression, with proper nesting of declarations and expressions, and is ready for standardization.

6.4 Core parser functions

The parser's functionality is implemented in the Parser class in parser.py, with key functions corresponding to the RPAL grammar's non-terminals. The primary functions include:

- parse_E(self): The entry point for parsing expressions, handling constructs like let, fn, and other expressions via parse_Ew. It constructs nodes for let and lambda forms.
- parse_D(self): Parses declarations, supporting constructs like within, and, and recursive definitions (rec).
- parse_T(self): Handles tuple expressions, building tau nodes for comma-separated expressions.
- parse_Bp(self): Processes boolean and comparison operations (e.g., gr, eq), creating nodes for relational operators.
- parse_A(self): Manages arithmetic expressions, handling operators like +, -, *, /, and **.
- parse_R(self): Parses applications, constructing gamma nodes for function applications.
- parse_Vb(self): Handles variable bindings, supporting single identifiers or parenthesized lists.

- buildTree(self, token, numberOfChildren): Constructs an AST node with the specified type and number of children, popping children from the stack and reversing their order to match the grammar.
- read(self, expected=None): Consumes the current token, validates it against an expected type or value, and creates terminal nodes for identifiers, integers, strings, and keywords.

These functions work together to recursively traverse the token stream, building the AST by pushing terminal nodes (e.g., <ID:x>) and constructing non-terminal nodes (e.g., let, +) based on grammar rules.

6.5 AST node class structure

The AST is represented using the ASTnode class defined in ASTnode.py, which provides a flexible structure for both terminal and non-terminal nodes. The class is defined as follows:

```
class ASTnode:

    def __init__(self, type):
        self.type = type
        self.value = None
        self.child = []


    def __str__(self):
        return f"ASTnode(type={self.type}, value={self.value})"
```

- **Attributes**:
    - type: The node type (e.g., let, lambda, ID, +), indicating the construct or token type.
    - value: The value for terminal nodes (e.g., x for an identifier, 5 for an integer). Non-terminal nodes typically have None.
    - child: A list of child ASTnode objects, representing the hierarchical structure of the AST.

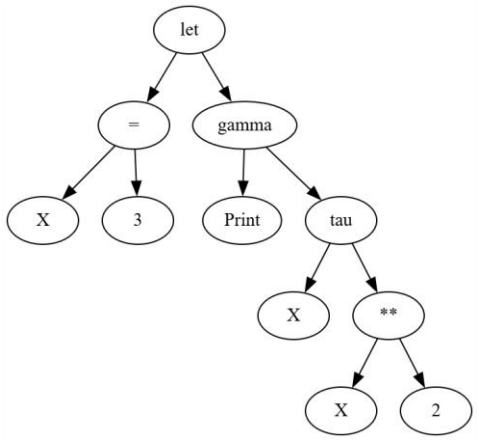
The ASTnode class supports the construction of a tree where terminal nodes (e.g., <ID:x>) store token values, and non-terminal nodes (e.g., let) organize the program's structure. The __str__ method provides a debugging representation, though the print_ast function in myrpal.py formats the tree for readable output with indentation.

6.6 Screenshot or diagram of an AST tree

When it gives example input,

let X=3 in Print (X,X**2)

The diagram of an AST tree would look like this,




```
Tests > sample1.rpal
1  let X=3
2      in
3    Print(X,X**2)
4    //  Prints (3,9)
5
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  POSTMAN CONSOLE

PS C:\partition D\sem4\PL\programming_language_project> make  ast FILE=sample1.rpal
>>
python myrpal.py -ast Tests/sample1.rpal
let
.=
..<ID:X>
..<INT:3>
.gamma
..<ID:Print>
..tau
...<ID:X>
...**
....<ID:X>
....<INT:2>
PS C:\partition D\sem4\PL\programming_language_project>
```

7. Standardization

7.1 Goal Transform AST → Standardized Tree (ST)

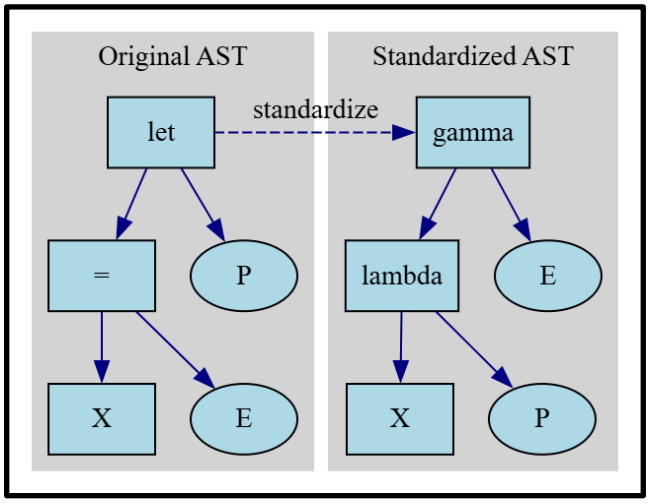
The standardization phase transforms the Abstract Syntax Tree (AST) generated by the parser into a Standardized Tree (ST) to simplify the subsequent control structure generation and evaluation by the CSE machine. This process applies a set of transformation rules to convert complex RPAL constructs (e.g., let, where, rec, fcn_form) into a uniform representation using lambda, gamma, and other constructs. The standardized tree ensures that the RPAL program is expressed in a form that is easier to evaluate, reducing the complexity of the CSE machine's execution logic.

7.2 Grammar transformation rules

The standardization process, implemented in standerdizer.py, follows transformation rules derived from the RPAL language specification, which can be visualized as transformations of the AST's structure. These rules convert high-level RPAL constructs into equivalent forms using lambda and gamma nodes, ensuring functional equivalence while simplifying the tree. The key transformation rules, as implemented in the standardize function, are:

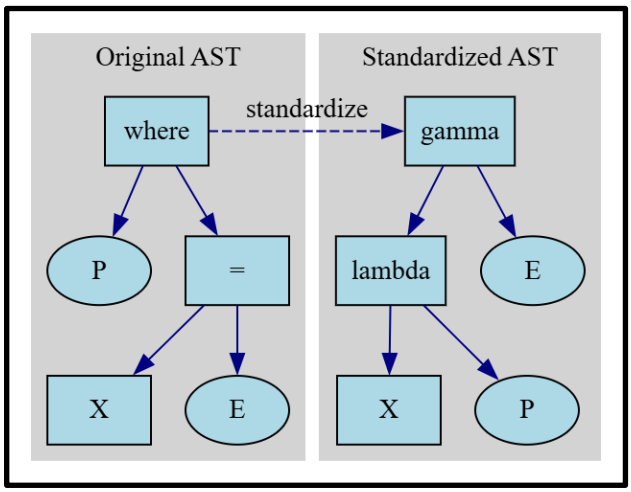
1.Let Transformation:

Rule: let (= X E) P → gamma (lambda X P) E
Description: A let expression is transformed into a function application (gamma) where the bound variable X and body P form a lambda expression applied to the expression E.



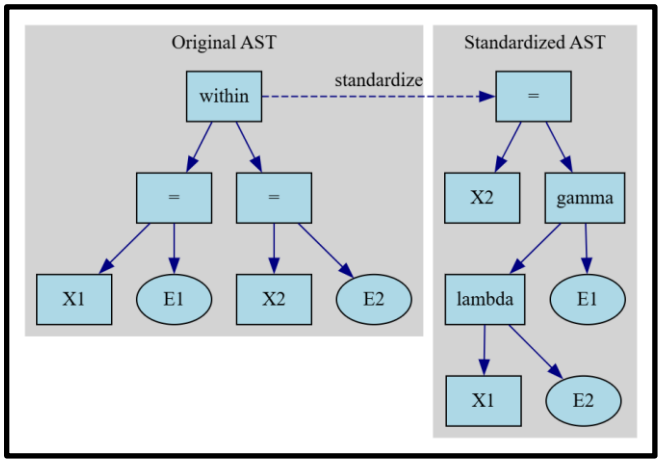
2.Where Transformation:

Rule: where P (= X E) → gamma (lambda X P) E
Description: Similar to let, a where expression reverses the order of the body and binding, transforming into a gamma node with a lambda encapsulating X and P.



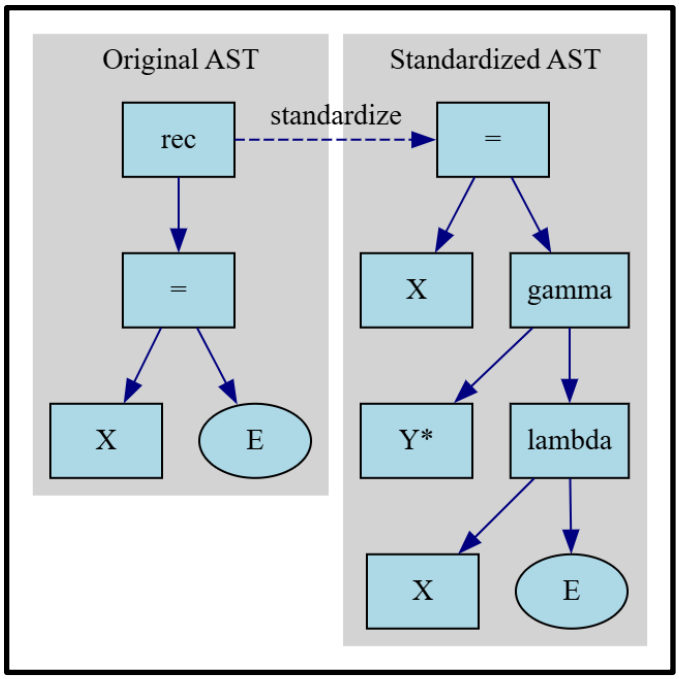
3.Within Transformation:

Rule: within (= X1 E1) (= X2 E2) → (= X2 (gamma (lambda X1 E2) E1))
Description: A within construct nests one binding inside another, transforming into an assignment where X2 is bound to the application of a lambda function (binding X1 to E2) to E1.



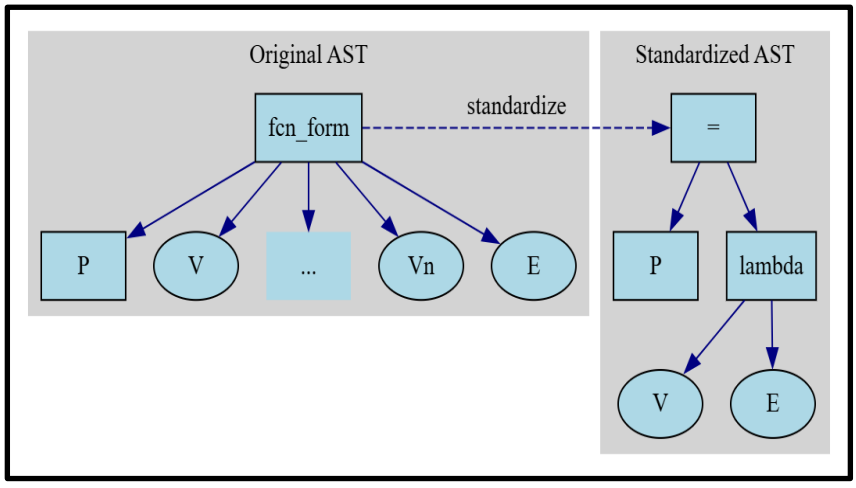
4.Rec Transformation:

Rule: rec (= X E) → (= X (gamma Y* (lambda X E)))
Description: A recursive definition is transformed into an assignment using the fixed-point operator Y* to enable recursive evaluation.



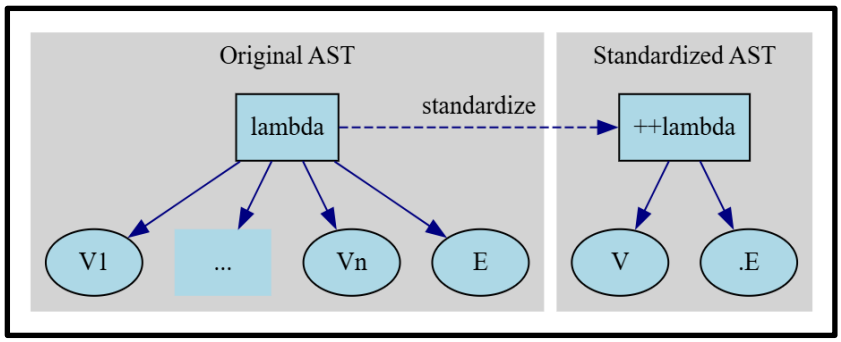
6.Function Form Transformation:

Rule: fcn_form P V+ E → (= P (lambda V (. E)))
Description: A function form is transformed into an assignment where P is bound to a nested lambda structure capturing multiple variables V and the expression E.



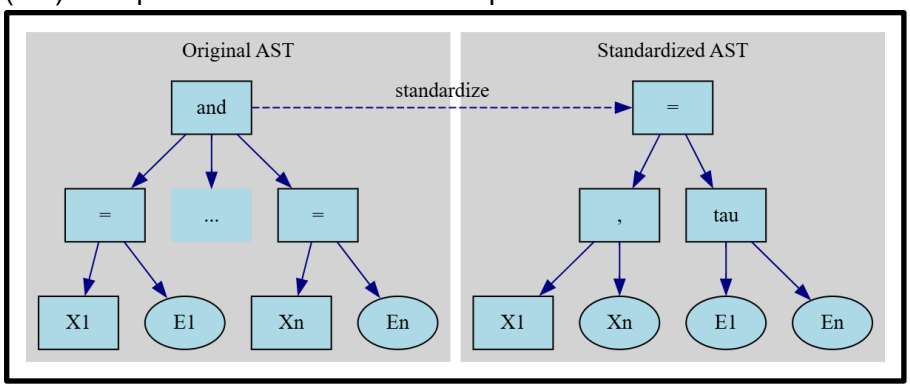
7.Multi-Variable Lambda Transformation:

Rule: lambda V++ E → lambda V (. E)
Description: A lambda with multiple variables is transformed into nested lambda nodes, each binding one variable.



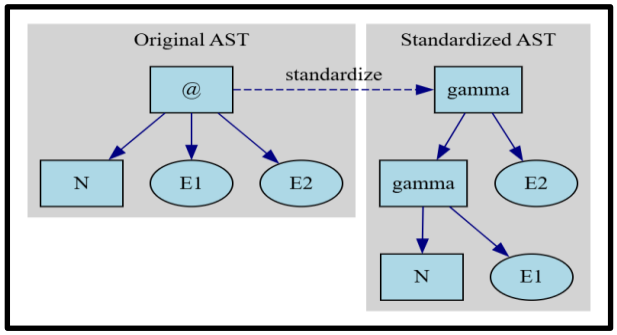
8.And Transformation:

Rule: and ((= X1 E1) ... (= Xn En)) → (= (X1, ..., Xn) (tau E1 ... En))
Description: Simultaneous bindings are transformed into a single assignment using a tuple (tau) of expressions and a comma-separated list of variables.



9.Infix Application Transformation:

Rule: @ N E1 E2 → gamma (gamma N E1) E2
Description: An infix operator application is transformed into nested gamma applications for evaluation.



These rules are applied recursively to the AST, ensuring that all nodes are transformed appropriately while preserving the program's semantics.

7.3 Important transformations

The most significant transformations, due to their frequency and impact on the evaluation process, include:

- **Let and Where**: These are common in RPAL programs and are transformed into gamma and lambda nodes to represent function application. This simplifies the evaluation of bindings in the CSE machine.
- **Function Form**: The transformation of fcn_form into nested lambda nodes supports RPAL's functional programming paradigm, enabling multi-argument functions through currying.
- **And**: The and transformation handles simultaneous bindings, crucial for programs with multiple parallel assignments, by creating tau and comma nodes.
- **Rec**: The recursive transformation introduces the Y* operator, enabling recursive function evaluation, which is essential for RPAL's functional constructs.

These transformations ensure that the standardized tree is optimized for the CSE machine, reducing the complexity of handling diverse syntactic forms.

7.4 AST vs ST

Consider the following RPAL program:

let X=3 in Print (X,X**2)

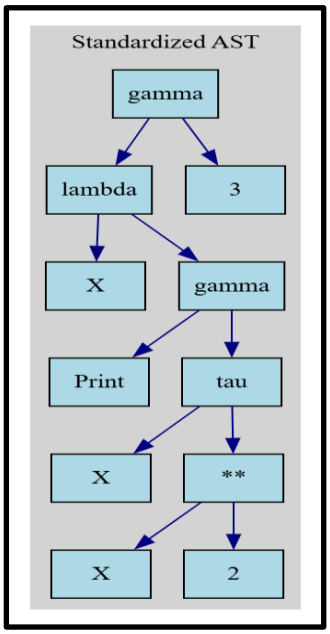
Before Standardization (AST)	**After Standardization (ST)**
let .= ..<ID:X> ..<INT:3> .gamma ..<ID:Print> ..tau ...<ID:X> ...** ....<ID:X> ....<INT:2>	gamma .lambda ..<ID:X> ..gamma ...<ID:Print> ...tau ....<ID:X> ....** .....<ID:X> .....<INT:2> .<INT:3>

Explanation:

- The let node is transformed into a gamma node.
- The = node's children (<ID:x> and <INT:5>) are reorganized into a lambda node (binding x to the body x + 3) applied to the expression 5.
- The + node and its children remain unchanged, as they are already in a standard form.

This transformation, performed by the standardize function, converts the let construct into a form suitable for CSE evaluation.

Diagram of ST Tree,



7.5 Code reference

The standardization process is implemented in the standardize function in standerdizer.py . Its prototype is:

def standardize(node: ASTnode) -> ASTnode:

- **Input**: An ASTnode representing a node in the AST.
- **Output**: An ASTnode representing the corresponding node in the standardized tree.
- **Implementation Details**:
    - Recursively processes each node, applying the appropriate transformation rule based on the node's type.
    - Handles base cases (nodes with no children) by returning them unchanged.
    - Constructs new ASTnode objects for transformed constructs (e.g., gamma, lambda, tau).
    - Ensures child nodes are standardized before building the parent node, maintaining the tree's integrity.
    - Includes debug print statements (commented out) for tracking transformations, useful for development and testing.

The function systematically applies the transformation rules listed above, ensuring that the resulting ST is consistent with RPAL's evaluation semantics.

8. Control Structure Generation

8.1 Goal

The control structure generation phase transforms the Standardized Tree (ST) produced by the standardization phase into a set of control structures that serve as the input for the Control-Stack-Environment (CSE) machine. This process translates the abstract representation of the RPAL program into a sequence of executable units, such as Lambda, Delta, and Tau, which facilitate efficient evaluation. The generated control structures encapsulate the program's logic, including function definitions, conditionals, and tuple formations, while maintaining environment information for variable bindings.

8.2 How environments and lambdas are handled?

The control structure generation, implemented in csemachine.py, processes the standardized tree recursively to produce a list of control structures indexed by a global counter (count). Each control structure is a sequence of nodes or objects that represent operations or data to be processed by the CSE machine. The handling of environments and lambdas is critical to ensure correct scoping and function application:

- **Environments** - The Environment class (from Environment.py), which keeps track of variable bindings and upholds a parent-child hierarchy, is used to manage environments. Every lambda node in the ST is linked to the current environment index (current_env) during the creation of the control structure. This guarantees that the appropriate scope can be accessed by variable lookups during evaluation. New environments are added as needed during evaluation (e.g., for lambda applications) after the initial environment (e_0) is created at the beginning.
- **Lambdas** - In the ST, lambda expressions are transformed into Lambda objects (from structure.py), which hold the body of the lambda as a reference to another control structure, the environment index at definition time, and the bounded variable or variables. A new Lambda object with a distinct number (count) is generated for a lambda node, and its body is handled as an independent control structure. In order to support currying and tuple-based applications, the bounded variables are stored as a list if the lambda binds multiple variables (for example, from a and construct).

The generation process ensures that each node type in the ST is mapped to an appropriate control structure, preserving the program's semantics while preparing it for stack-based evaluation.

8.3 Class used

The following classes, defined in structure.py, are used to represent control structures:

- **Delta**: Represents a conditional branch (-> in the ST) or the body of a lambda. It has:
    - number: A unique identifier for the control structure.
    - body: A list of control structure elements (e.g., nodes, literals, or other structures).

- type: Set to "Delta".

```
class Delta:

    def __init__(self, number):

        self.number = number

        self.body = []  # Store control structure content

        self.type = "Delta"



    def __str__(self):

        return f"Delta({self.number})"
```

- **Tau**: Represents a tuple formation, used for comma-separated expressions. It has:
  - number: The number of elements in the tuple.
  - type: Set to "Tau".

```
class Tau:

    def __init__(self, number):

        self.number = number

        self.type = "Tau"



    def __str__(self):



        return f"Tau({self.number})"
```

- **Lambda**: Represents a lambda expression, supporting function definitions. It has:
  - number: A unique identifier.
  - bounded_variable: A single variable (string) or list of variables.
  - environment: The environment index at definition time.
  - body: A reference to the control structure for the lambda's body.
  - type: Set to "Lambda".

```
class Lambda:

    def __init__(self, number):

        self.number = number

        self.bounded_variable = None

        self.environment = None

        self.body = []  # Store control structure content

        self.type = "Lambda"



    def __str__(self):

        return f"Lambda({self.number},
{self.bounded_variable})"
```

- **Eta**: Represents a fixed-point operator application for recursion. It has:
  - number: A unique identifier.
  - bounded_variable: The bound variable(s).
  - environment: The environment index.
  - type: Set to "Eta".

```
class Eta:

    def __init__(self, number):

        self.number = number

        self.bounded_variable = None

        self.environment = None

        self.type = "Eta"

    def __str__(self):

        return f"Eta({self.number},
{self.bounded_variable})"
```

Additionally, the Environment class (from Environment.py) supports environment management, and the ASTnode class (from ASTnode.py) is used for literal nodes (e.g., <ID:x>, <INT:5>).

9. CSE Machine

9.1 Execution engine for control stack evaluation

The Control-Stack-Environment (CSE) machine is the execution engine of the RPAL interpreter, responsible for evaluating the control structures generated from the standardized tree to produce the final output of a RPAL program. Implemented in csemachine.py, the CSE machine operates by maintaining a control stack, an evaluation stack, and a set of environments, processing control structures according to predefined transition rules. This approach supports the functional semantics of RPAL, handling lambda applications, conditionals, tuples, and built-in functions efficiently.

9.2 stack, control, environment setup

The CSE machine uses three core components to manage execution:

Stack: Implemented as the Stack class in stack.py, the evaluation stack stores intermediate values, such as literals, lambda closures, tuples, and environment markers. The stack supports operations like push, pop, and is_empty, with error handling for unexpected empty states. It is initialized with type "CSE" to distinguish it from the parser's stack.
class Stack:

```
class Stack:
    def __init__(self, type):
        self.stack = []
        self.type = type
def push(self, item):
        self.stack.append(item)
def pop(self):
        if not self.is_empty():
            return self.stack.pop()
        else:
            if self.type == "CSE":
                print("Stack in CSE machine has become empty
unexpectedly.")
            else:
                print("Stack used for AST generation has become empty
unexpectedly.")
            exit(1)
```

- **Control**: The control stack is a list (control in csemachine.py) that holds the control structures to be executed. It is initialized with the root environment (e_0) and the first control structure (control_structures[0]). Elements are popped and processed sequentially, driving the evaluation process.

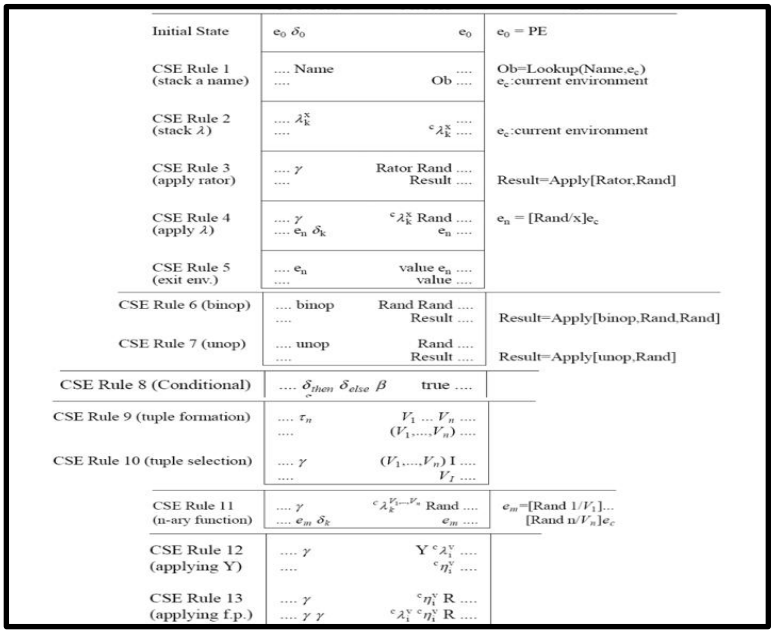
Environment: Environments are managed using the Environment class in Environment.py, which maintains variable bindings and a parent-child hierarchy. Each environment has a unique number, a name (e.g., e_0), a dictionary of variables, and references to its parent and children. The initial environment (e_0) is created at the start, and new environments are added during lambda applications to handle variable scoping.
class Environment.

```
    def __init__(self, number, parent):
        self.number = number
        self.name = "e_" + str(number)
        self.variables = {}
        self.children = []
        self.parent = parent
```

The CSE machine initializes with an empty stack, a control stack containing the initial environment and control structure, and a single environment (e_0). As execution progresses, the machine updates the current environment (current_environment) and extends the control stack with new structures as needed.

9.3 Transition rules supported

The CSE machine implements transition rules to process control structures. These rules, implemented in the apply_rules function in csemachine.py, handle various constructs:



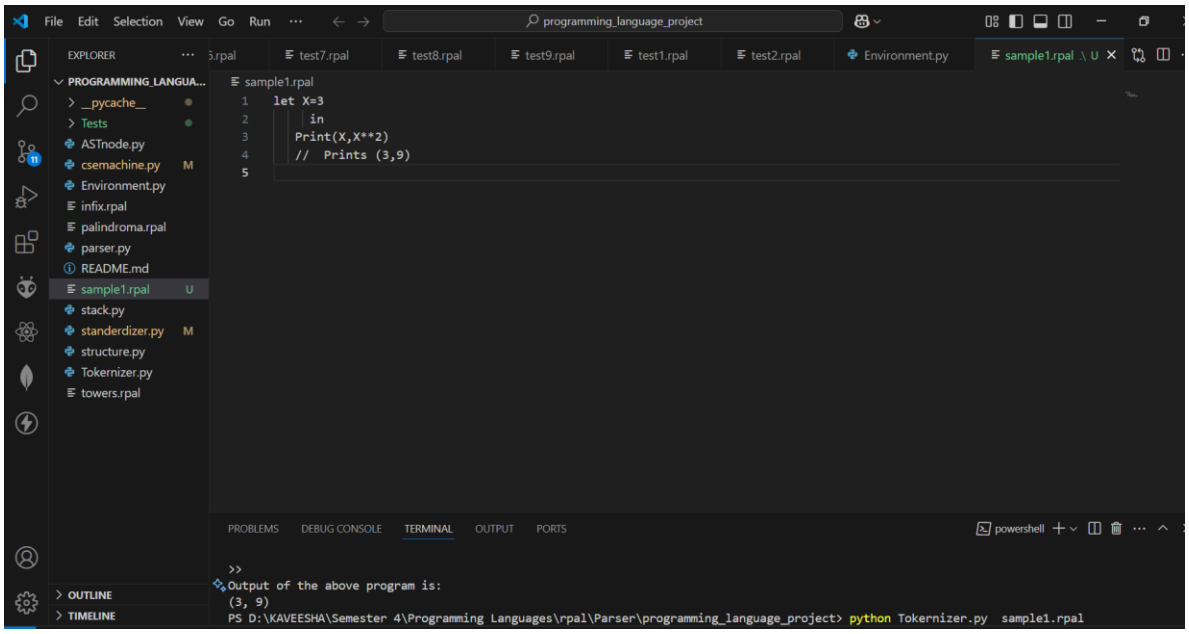
9.4 Predefined functions handled

The CSE machine supports a set of predefined functions, implemented in the built_in function in csemachine.py. The supported functions include:

- **Order**: Returns the size of a tuple (len(argument)).
- **Print, print**: Outputs the argument, handling strings with escape sequences (e.g., \n, \t). Sets a print_present flag to track usage.
- **Conc**: Concatenates two strings, popping an additional argument from the stack.
- **Stern**: Returns the string after the first character (argument[1:]).
- **Stem**: Returns the first character of a string (argument[0]).
- **Isinteger**: Checks if the argument is an integer (isinstance(argument, int)).
- **Istruthvalue**: Checks if the argument is a boolean (isinstance(argument, bool)).
- **Isstring**: Checks if the argument is a string (isinstance(argument, str)).
- **Istuple**: Checks if the argument is a tuple (isinstance(argument, tuple)).
- **Isfunction**: Checks if the argument is a built-in function.
- **ItoS**: Converts an integer to a string (str(argument)).
- **neg**: Negates an integer (-argument).

These functions handle common operations in RPAL, ensuring compatibility with the language's specification.

9.5 Final output of RPAL execution



10. Execution

10.1 How to run?

The RPAL interpreter is executed via the command line using the myrpal.py script, which serves as the entry point for processing RPAL source files. The interpreter supports three modes of operation, controlled by optional command-line flags. The usage is as follows:

- **Print AST**: Displays the Abstract Syntax Tree (AST) generated by the parser.
  python myrpal.py -ast file_name
- **Print Standardized Tree (ST)**: Displays the standardized tree after applying transformation rules.
  python myrpal.py -st file_name
- **Run CSE Machine (Default)**: Executes the program and prints the final output.
  python myrpal.py file_name

Requirements:

- Python 3.x must be installed.
- The input file (file_name) must contain valid RPAL code.
- The supporting files (ASTnode.py, parser.py, standerdizer.py, csemachine.py, stack.py, structure.py, Environment.py) must be in the same directory as myrpal.py.

The interpreter reads the input file, tokenizes it, parses it into an AST, standardizes the AST, generates control structures, and evaluates them using the CSE machine, depending on the specified flag.

10.2 Example input and output

Consider the following RPAL program saved in sample1.rpal,

```
let rec length S = S eq '' -> 0 | 1 + length (Stern S)

in Print ( length('1,2 ,  -  3'), length (''), length('abc'))
```

❖ AST Output (python myrpal.py -ast test.rpal)

```
let
.rec
..fcn_form
...<ID:length>
...<ID:S>
...->
```

```
....eq
.....<ID:S>
.....<STR:''>
....<INT:0>
....+
.....<INT:1>
.....gamma
......<ID:length>
......gamma
.......<ID:Stern>
.......<ID:S>
.gamma
..<ID:Print>
..tau
...gamma
....<ID:length>
....<STR:'1,2 ,  -  3'>
...gamma
....<ID:length>
....<STR:''>
...gamma
....<ID:length>
....<STR:'abc'>
```

- Standardized Tree Output (python myrpal.py -st test.rpal)

```
gamma
.lambda
..<ID:length>
..gamma
...<ID:Print>
...tau
....gamma
.....<ID:length>
.....<STR:'1,2 ,  -  3'>
....gamma
.....<ID:length>
.....<STR:''>
....gamma
.....<STR:'abc'>
.gamma
..Y*
..lambda
```

```
...<ID:length>
...lambda
....<ID:S>
....->
.....eq
......<ID:S>
......<STR:''>
.....<INT:0>
.....+
......<INT:1>
......gamma
.......<ID:length>
.......gamma
........<ID:Stern>
........<ID:S>
```

- CSE Machine Output (python myrpal.py test.rpal)

```
Output of the above program is:
(11, 0, 3)
```

11. Conclusion

The Right-Reference Pedagogical Algorithmic Language (RPAL), a functional programming language intended for educational use, has a fully functional interpreter thanks to the RPAL Interpreter project. A pipeline of lexical analysis (myrpal.py), parsing (parser.py), standardization (standerdizer.py), control structure generation, and evaluation using a Control-Stack-Environment (CSE) machine (csemachine.py) is used by the interpreter, which is implemented in Python 3. Strong data structures are offered by the supporting modules (ASTnode.py, stack.py, environment.py, and structure.py) for handling environments, stacks, trees, and tokens.

11.1 Achievements

- **Correctness**: The interpreter accurately processes RPAL programs, producing expected outputs for constructs like let, lambda, tuples, and built-in functions (e.g., Print, Order). Comparisons with a reference interpreter (e.g., rpal.exe) confirmed consistent results for test cases like let x = 5 in x + 3 (output: 8).
- **Flexibility**: Support for command-line flags (-ast, -st) allows users to inspect intermediate representations (AST and Standardized Tree), aiding debugging and education.

- **Modularity**: The codebase is organized into distinct modules, each handling a specific phase of interpretation, enhancing maintainability and readability.
- **Robustness**: Error handling for invalid tokens, syntax errors, and runtime issues (e.g., undeclared identifiers) ensures reliable operation.

The project provided valuable insights into compiler design, including lexical analysis, recursive descent parsing, tree transformations, and stack-based evaluation, deepening understanding of functional programming concepts like currying and closures.

11.2 Strengths and limitations

- **Complex Transformations**: Implementing standardization rules (e.g., let to gamma) required careful handling of nested constructs to preserve semantics, addressed by recursive processing in standerdizer.py.
- **CSE Machine Rules**: Coordinating the stack, control, and environment in csemachine.py to implement all transition rules (e.g., lambda application, tuple indexing) was intricate, resolved through systematic rule application.
- **Debugging**: Ensuring correct AST and ST representations required extensive testing, facilitated by debug print statements and intermediate output modes.

These challenges were overcome through iterative development, rigorous testing, and adherence to the RPAL specification.

11.3 Summary

The RPAL Interpreter project produced a reliable and adaptable tool for running RPAL programs, demonstrating the useful application of compiler design principles. It is a useful teaching tool because of its modular design, thorough error handling, and support for intermediate outputs. Building on the strong foundation already in place, future improvements could further enhance its performance and usability.

12. Appendices

12.1 CSE Machine Rules

Initial State	$e_0\ \delta_0$	e_0	e_0 = PE
CSE Rule 1 (stack a name)	.... Name	.... Ob	Ob=Lookup(Name,e_c) e_c:current environment
CSE Rule 2 (stack λ)	.... λ_k^x	.... $^c\lambda_k^x$	e_c:current environment
CSE Rule 3 (apply rator)	.... γ	Rator Rand .... Result	Result=Apply[Rator,Rand]
CSE Rule 4 (apply λ)	.... γ .... $e_n\ \delta_k$	$^c\lambda_k^x$ Rand .... e_n	e_n = [Rand/x]e_c
CSE Rule 5 (exit env.)	.... e_n	value e_n .... value	
CSE Rule 6 (binop)	.... binop	Rand Rand .... Result	Result=Apply[binop,Rand,Rand]
CSE Rule 7 (unop)	.... unop	Rand .... Result	Result=Apply[unop,Rand]
CSE Rule 8 (Conditional)	.... $\delta_{then}\ \delta_{else}\ \beta$	true	
CSE Rule 9 (tuple formation)	.... τ_n	V_1 ... V_n .... $(V_1,...,V_n)$	
CSE Rule 10 (tuple selection)	.... γ	$(V_1,...,V_n)$ I .... V_I	
CSE Rule 11 (n-ary function)	.... γ .... $e_m\ \delta_k$	$^c\lambda_k^{V_1,...,V_n}$ Rand .... e_m	e_m=[Rand 1/V_1]... [Rand n/V_n]e_c
CSE Rule 12 (applying Y)	.... γ	Y $^c\lambda_i^v$ .... $^c\eta_i^v$	
CSE Rule 13 (applying f.p.)	.... γ .... $\gamma\ \gamma$	$^c\eta_i^v$ R .... $^c\lambda_i^v\ ^c\eta_i^v$ R	

12.2 RPAL's Phrase StructureGrammar

```
# Expressions ###############################################

E    -> 'let' D 'in' E                          => 'let'
     -> 'fn'  Vb+ '.' E                         => 'lambda'
     -> Ew;
Ew   -> T  'where' Dr                           => 'where'
     -> T;

# Tuple Expressions #########################################

T    -> Ta ( ',' Ta )+                          => 'tau'
     -> Ta ;
Ta   -> Ta 'aug' Tc                             => 'aug'
     -> Tc ;
Tc   -> B '->' Tc '|' Tc                        => '->'
     -> B ;

# Boolean Expressions #######################################

B    -> B 'or' Bt                               => 'or'
     -> Bt ;
Bt   -> Bt '&' Bs                               => '&'
     -> Bs ;
Bs   -> 'not' Bp                                => 'not'
     -> Bp ;
Bp   -> A ('gr' | '>' ) A                       => 'gr'
     -> A ('ge' | '>=') A                       => 'ge'
     -> A ('ls' | '<' ) A                       => 'ls'
     -> A ('le' | '<=') A                       => 'le'
     -> A 'eq' A                                => 'eq'
     -> A 'ne' A                                => 'ne'
     -> A ;

# Arithmetic Expressions ####################################

A    -> A '+' At                                => '+'
     -> A '-' At                                => '-'
     ->   '+' At
     ->   '-' At                                => 'neg'
     -> At ;
At   -> At '*' Af                               => '*'
     -> At '/' Af                               => '/'
     -> Af ;
Af   -> Ap '**' Af                              => '**'
     -> Ap ;
Ap   -> Ap '@' '<IDENTIFIER>' R                 => '@'
     -> R ;

# Rators And Rands ##########################################

R    -> R Rn                                    => 'gamma'
     -> Rn ;
Rn   -> '<IDENTIFIER>'
     -> '<INTEGER>'
     -> '<STRING>'
     -> 'true'                                  => 'true'
     -> 'false'                                 => 'false'
     -> 'nil'                                   => 'nil'
     -> '(' E ')'
     -> 'dummy'                                 => 'dummy' ;
```

```
# Definitions ###############################################

D     -> Da 'within' D                          => 'within'
      -> Da ;
Da    -> Dr ( 'and' Dr )+                       => 'and'
      -> Dr ;
Dr    -> 'rec' Db                               => 'rec'
      -> Db ;
Db    -> Vl '=' E                               => '='
      -> '<IDENTIFIER>' Vb+ '=' E               => 'fcn_form'
      -> '(' D ')' ;

# Variables #################################################

Vb    -> '<IDENTIFIER>'
      -> '(' Vl ')'
      -> '(' ')'                                => '()';
Vl    -> '<IDENTIFIER>' list ','                => ','?;
```

12.3 RPAL's LEXICON

```
Identifier -> Letter (Letter | Digit | '_')*            => '<IDENTIFIER>';

Integer    -> Digit+                                    => '<INTEGER>';

Operator   -> Operator_symbol+                          => '<OPERATOR>';

String     -> ''''
              ( '\' 't' | '\' 'n' | '\' '\' | '\' ''''
              | '('     | ')'     | ';'     | ','
              | ' '
              | Letter | Digit | Operator_symbol
              )* ''''                                   => '<STRING>';

Spaces     -> ( ' ' | ht | Eol )+                       => '<DELETE>';

Comment    -> '//'
              ( '''' | '(' | ')' | ';' | ',' | '\' | ' '
                | ht | Letter | Digit | Operator_symbol
              )* Eol                                    => '<DELETE>';

Punction   -> '('                                      => '('
           -> ')'                                       => ')'
           -> ';'                                       => ';'
           -> ','                                       => ',';

Letter     -> 'A'..'Z' | 'a'..'z';

Digit      -> '0'..'9';

Operator_symbol
           -> '+' | '-' | '*' | '<' | '>' | '&' | '.'
            | '@' | '/' | ':' | '=' | '~' | '|' | '$'
            | '!' | '#' | '%' | '^' | '_' | '[' | ']'
            | '{' | '}' | '"' | '`' | '?';
```