CS 3513- Programming Languages

Project- REPORT
Designing an interpreter for RPAL language

Prepared by Group Netcore

-Vishmitha W.D.L 220669E

- Pankaji R.K.K.M.M.S 220442D

Introduction

The RPAL Interpreter is a lightweight, command-line Java application that brings the core ideas of a purely functional language to life. It reads RPAL (Right-Programming Algebraic Language) source

files and processes them through four clear stages:

1. **Lexical Analysis** – The Scanner breaks input text into meaningful tokens.

2. **Parsing** – The Parser assembles those tokens into an Abstract Syntax Tree (AST), capturing the structure of functions, applications, tuples, and control constructs.

3. **AST Standardization** – The AST class transforms every construct into a uniform "standard" form and generates delta closures for built-in operations (arithmetic, logical, tuple, and string primitives).

4. **Evaluation** – The CSE_Machine drives a Control-Structure Environment (CSE) machine over the standardized tree, supporting conditionals (via Beta_Condition_Eval), recursion (via EtaRecursiveFixedPoint), tuple operations, and fixed-point combinators.

Designed for CS 3513 (Programming Languages), this interpreter demonstrates how high level functional constructs map to low-level control-structure evaluations. Its simple Makefile and entry point (myrpal.java) let you compile, run, inspect the AST, and clean up with a handful of commands. Through this project you'll gain hands-on experience with lexical analysis, recursive descent parsing, AST transformations, closure creation, and stack-based evaluation foundational techniques in compiler and interpreter design.

Project Structure Overview

Below is a breakdown of the directories and key files in your RPAL interpreter project, along with a brief description of each component's role:

RPAL INTERPRETER/

```
├── Abstract_Syntax_Tree/
│   ├── AST.java
│   ├── ASTNode.java
│   └── ASTNodeType.java
├── Cse_Machine/
│   ├── CSE_Machine.java
│   ├── Beta_Condition_Eval.java
│   ├── DeltaControlStructure.java
│   ├── EtaRecursiveFixedPoint.java
│   ├── Environment.java
│   ├── Node_Duplication.java
│   └── Tuple.java
├── Lex_Analyzer/
│   ├── Scanner.java
│   └── Token.java
├── Parser/
│   └── Parser.java
├── Rpal_Test_Programs/
│   └── *.rpal      (sample RPAL source files for testing)
├── myrpal.java     (main entry point)
├── Makefile        (compile/run/clean commands)
└── README.md       (project overview & usage instructions)
```

1. Abstract_Syntax_Tree/

Purpose: Defines the in-memory representation of RPAL programs.

Contents:

- **AST.java** – Manages the entire tree, supports standardization and delta-closure generation.
- **ASTNode.java** – Node structure: type, value, children/siblings, and source-line info.
- **ASTNodeType.java** – Enumeration of all possible node kinds (literals, operators, control constructs).

2. Cse_Machine/

Purpose: Implements the Control-Structure Environment (CSE) evaluator for a standardized AST.

- **CSE_Machine.java** – Core evaluation engine: value stack, dispatch of operations, control-structure loop.
- **Beta_Condition_Eval.java** – Handles conditional ("if-then-else") control nodes.
- **DeltaControlStructure.java** – Encapsulates primitive closures (delta operations).
- **EtaRecursiveFixedPoint.java** – Implements fixed-point combinator support for recursion.
- **Environment.java** – Maintains nested variable-binding scopes for closures.
- **Node_Duplication.java** – Visitor for deep-copying AST nodes.
- **Tuple.java** – Models tuple values and operations.

3. Lex_Analyzer/

Purpose: Tokenizes raw RPAL source into a stream of Token objects.

Contents:

- **Scanner.java** – Reads characters, applies regex patterns, and assembles tokens.
- **Token.java** – Stores token type, literal value, and source-line number for parser use.

4. Parser/

Purpose: Builds an AST from tokens using a recursive-descent grammar.

Contents:

- **Parser.java** – Drives grammar rules, matches tokens, and constructs ASTNode hierarchies.

5. Rpal_Test_Programs/

Purpose: Collection of sample .rpal files to validate interpreter functionality.

6. myrpal.java

Purpose: Application entry point.

Responsibilities: Parses command-line arguments (-ast flag),

invokes Scanner & Parser , optionally prints the AST, and then delegates to CSE_Machine for evaluation.

Execute The Programme

can compile and run it directly from the Windows Command Prompt (CMD) without needing any

IDE.

Compile & Run in Windows CMD

1. Open Command Prompt

   Press Win + R, type cmd , and hit Enter.

2. Navigate to your project folder

   C:\> cd \path\to\RPAL INTERPRETER

3. Compile all Java sources

   C:\path\to\RPAL INTERPRETER> javac *.java

   This generates .class files for every .java in the folder and its subdirectories.

4. Run an RPAL program

   C:\path\to\RPAL INTERPRETER> java myrpal Rpal_Test_Programs\example.rpal

   Replace example.rpal with the name of your test file. The interpreter will print the evaluated

   result to the console.

5. Print only the AST (no evaluation)

   C:\path\to\RPAL INTERPRETER> java myrpal -ast Rpal_Test_Programs\example.rpal

   This shows the tree structure and exits.

6. Clean up compiled files

   To remove all .class files in the current directory:

   C:\path\to\RPAL INTERPRETER> del /S *.class

Classes and Significant Functions

Class: myrpal.java

Introduction:

The myrpal class serves as the entry point for the RPAL interpreter. It handles command-line arguments, orchestrates the scanning and parsing phases to build the AST, optionally prints the AST, and then delegates evaluation to the CSE machine.

Methods:

- **private static AST createAST(String filePath)**
  Constructs the AST for the given source file.
  - Opens the file using Scanner.
  - Invokes Parser on the scanner to build the AST.
  - Returns the newly created AST object, or null on I/O errors.

- **private static String interpretAST(AST ast)**
  Evaluates a standardized AST using the CSE machine.
  - Instantiates CSE_Machine with the provided AST.
  - Calls evaluateRPALProgram() to perform the evaluation.
  - Returns the evaluation result as a String.

Classes in the Parser package

Class: Parser.java

Introduction:

The Parser class implements a recursive-descent parser that consumes tokens from the Scanner and builds an Abstract Syntax Tree (AST) representing an RPAL program. It drives the grammar rules for programs, expressions, and sub-expressions, organizing tokens into a tree structure for later evaluation.

Methods:

- **public Parser(Scanner scanner)**
  Constructor that initializes the parser with a Scanner instance. It sets up the AST object and reads the first token into currentToken.
- **public AST Build_AST() throws IOException**
  Entry point for parsing. Invokes parseProgram() to process the entire input according to the RPAL grammar and returns the completed AST.
- **private void parseProgram() throws IOException**
  Parses the top-level "Program → expression EOL" rule.
  - Creates the root PROGRAM AST node.
  - Parses a single expression via parseExpression().
  - Ensures the program ends with an end-of-line token.

- **private ASTNode parseExpression() throws IOException**
  Recursively parses RPAL expressions (lambdas, applications, operators, literals, etc.) and returns an ASTNode representing that expression subtree.

- **private void match(Token.Type expected) throws IOException**
  Checks that currentToken matches the expected token type.
    - If it does, advances to the next token; otherwise, throws a parse error.

- **(Additional helper methods)**
  Although not all are shown here, the parser typically includes private methods such as:
    - **parseTerm()** – handles sub-expressions like variables, tuples, and parenthesized groups.
    - **parseApplication()** – deals with function application syntax.
    - **parseLambda()** – recognizes and constructs lambda-abstraction nodes.
    - **parseAtomic()** – parses atomic literals (numbers, identifiers).

Each of these helper methods follows the RPAL grammar to build the correct ASTNode hierarchy for its language construct.

Classes in the Lex_Analyzer package

Class: Scanner.java

Introduction:

The Scanner class reads RPAL source text from a file and breaks it into a stream of Token objects. It uses regular expressions to recognize identifiers, numbers, operators, strings, punctuation, whitespace, and comments, and tracks line numbers for error reporting.

Methods:

- **public Scanner(String inputFile) throws IOException**
  Initializes the scanner to read from the given file path, setting up a buffered reader and starting the line counter at 1.
- **public Token readNextToken()**
  Retrieves the next lexical token by pulling characters (or reuse one pushed back) and delegating to **matchAndGetNextToken**; returns null at end-of-file.
- **private String getNextCharacterFromSource()**
  Reads and returns the next character as a String; increments line count on '\n' and closes the reader when the file ends.
- **private Token matchAndGetNextToken(String currentCharacter)**
  Examines the given character against precompiled patterns and calls the appropriate token-building method (**getIdentifierToken, getIntegerToken**, etc.).
- **private Token getIdentifierToken(String currentCharacter)**
  Builds an identifier or keyword token by accumulating letters, digits, and underscores, then marks reserved words with the reserved token type.
- **private Token getIntegerToken(String currentCharacter)**
  Reads consecutive digit characters to form an integer literal token.
- **private Token getOperatorToken(String currentCharacter)**
  Accumulates operator symbols into a single token; also detects comment start ("//") and hands off to **getCommentToken.**
- **private Token getStringToken()**
  Reads characters inside single quotes to form a string literal, stopping at the closing quote.
- **private Token getSpaceToken(String currentCharacter)**
  Consumes whitespace characters (spaces, tabs, newlines) into a delete-type token, which parsers will ignore.
- **private Token getCommentToken(String currentCharacter)**
  Reads characters following "//" up to end-of-line into a delete-type token to skip comments.

- **private Token getPunctuationToken(String currentCharacter)**
  Recognizes punctuation symbols ((,), ;, ,) and returns the corresponding token type.
- **private static String escapeMetaChars()**
  Produces an escaped string of operator meta-characters for use in building the operator-matching regular expression.

Class: Token.java

Introduction:

The Token class encapsulates a single lexical token produced by the Scanner. It holds the token's type (from the Token.Type enum), its literal value (e.g., the identifier or number string), and the source line number where it was found—essential for error reporting and downstream parsing.

Methods:

- **public int getTokenType()**
  Returns the numeric code representing this token's type (e.g., identifier, integer, operator).
- **public void setTokenType(int tokenType)**
  Sets the token's type code, typically assigned by the scanner based on pattern matching.
- **public String getTokenValue()**
  Retrieves the literal string value of the token (e.g., "sum", "42", "+").
- **public void setTokenValue(String tokenValue)**
  Assigns the literal value for this token.
- **public int getLineNumberOfSourceWhereTokenIs()**
  Returns the line number in the source file where this token was read.
- **public void setLineNumberOfSourceWhereTokenIs(int lineNumberOfSourceWhereTokenIs)**
  Records the source line number for this token, used for error messages and debugging.

Class: CSE_Machine.java

Introduction:

The CSE_Machine class implements the Control–Structure Environment (CSE) machine that drives the evaluation of a standardized RPAL AST. It maintains a value stack, applies primitive operations (deltas), handles control structures (including conditionals and recursion), and produces the final result as a string.

Methods:

- **public CSE_Machine(AST ast)**
  Constructor:
  - Initializes the internal value stack.
  - Loads the root delta control structure for primitive operations.
  - Standardizes the input AST and primes the machine for evaluation.

- **public String evaluateRPALProgram()**
  Main driver:
  - Processes the control-stack representation of the AST.
  - Executes each node in turn and collects the final result.
  - Returns the program's output as a formatted String.

- **private void processControlStructures(Stack<ASTNode> control)**
  Iterates the control-structure stack, popping each node and dispatching it to processCurrentNodeOfControlStructure(...).
- **private void processCurrentNodeOfControlStructure(ASTNode node)**
  Examines the node's type (literal, operator, identifier, tuple, β-node, etc.) and invokes the appropriate handler or delta operation.

- **private ASTNode applyBinaryOperation(ASTNodeType opType, ASTNode left, ASTNode right)**
  Dispatches to the correct binary operation based on opType (arithmetic, comparison, logical).
- **private void binaryArithmeticOperation(ASTNodeType type)**
  Performs integer arithmetic (+, -, *, /) by popping two operands, computing the result, and pushing the result node.
- **private void binaryLogicalEqualNotEqualOperation(ASTNodeType type)**
  Handles EQ / NEQ comparisons for integers, strings, and truth values.
- **private void compareIntegers(ASTNode a, ASTNode b, ASTNodeType type)**
  Compares two integer literal nodes and pushes a boolean result (true or false).
- **private void compareStrings(ASTNode a, ASTNode b, ASTNodeType type)**
  Compares two string literal nodes for equality/inequality.
- **private void compareTruthValues(ASTNode a, ASTNode b, ASTNodeType type)**
  Compares two boolean nodes.
- **private void binaryLogicalOrAndOperations(ASTNodeType type)**
  Performs boolean AND / OR by popping two truth-value nodes and pushing the combined result.
- **private void orAndTruthValues(ASTNode a, ASTNode b, ASTNodeType type)**
  Helper that applies the actual AND/OR logic to two boolean nodes.
- **private void applyUnaryOperation(ASTNodeType type)**
  Dispatches to neg(...) or not(...) based on type.
- **private void neg(ASTNode node)**
  Arithmetic negation: negates an integer literal and pushes the result.
- **private void not(ASTNode node)**
  Logical NOT: negates a boolean literal and pushes the result.

- **private void augTuples(ASTNodeType type)**
  Handles tuple concatenation and selection operations (conc, stem, stern).
- **private void conc(ASTNode tupleNode)**
  Concatenates two tuples into one.
- **private void stem(ASTNode tupleNode)**
  Returns the first (head) element of a tuple.
- **private void stern(ASTNode tupleNode)**
  Returns the tail (remaining elements) of a tuple.
- **private void itos(ASTNode node)**
  Converts an integer literal node to its string representation.
- **private void order(ASTNode node)**
  Maps a string or tuple to its lexicographic "order" value.
- **private void isNullTuple(ASTNode node)**
  Checks whether a tuple node is empty, pushing a boolean.
- **private void tupleSelection(ASTNodeType type, ASTNode tupleNode, ASTNode indexNode)**
  Generalized Nth-element selector (for stem, stern, etc.).
- **private void getNthTupleChild(ASTNode tupleNode, int n)**
  Retrieves the nth child element from a tuple node.
- **private void handleIdentifiers(ASTNode node)**
  Looks up identifiers in the current environment or delta definitions and pushes their value.
- **private void handleBeta(ASTNode node)**
  Processes Beta_Condition_Eval nodes, evaluating the condition and selecting the correct branch.
- **private int getNumChildren(ASTNodeType type)**
  Returns the arity (number of arguments) expected by a given delta operator.
- **private void pushTrueNode() / pushFalseNode() / pushDummyNode()**
  Utility methods to push literal true, false, or placeholder nodes onto the stack.

- **private void printEvaluationErrorToStdOut(int line, String msg1, String msg2)**
  Prints runtime error messages with source-line context.
- **private void printNodeValue(ASTNode node)**
  Converts the final AST node value into a printable format.
- **private boolean isReservedIdentifier(String value)**
  Checks if a given name corresponds to a built-in delta or keyword.

Class: Beta_Condition_Eval.java

Introduction:

Handles RPAL conditional expressions of the form cond → then | else. By extending ASTNode, it ensures the condition (cond) is evaluated first and only the appropriate branch (thenBody or elseBody) is executed, preventing infinite recursion in recursive definitions.

Methods:

- **public Beta_Condition_Eval()**
  Initializes the node's type to BETA and creates empty stacks for the then- and else-branch AST nodes.
- **public Beta_Condition_Eval acceptASTNode(Node_Duplication nodeCopier)**
  Supports copying this node via a Node_Duplication visitor, returning a duplicated instance.
- **public Stack<ASTNode> getThenBody()**
  Returns the stack of AST nodes representing the "then" branch.
- **public void setThenBody(Stack<ASTNode> thenBody)**
  Sets the AST node stack to be used for the "then" branch.
- **public Stack<ASTNode> getElseBody()**
  Returns the stack of AST nodes representing the "else" branch.
- **public void setElseBody(Stack<ASTNode> elseBody)**
  Sets the AST node stack to be used for the "else" branch.

Class: DeltaControlStructure.java

Introduction:

Represents a DELTA-style closure node in the CSE machine. It bundles the list of bound variable names, the environment where it was created, the AST nodes forming its body, and a unique numeric identifier.

Methods:

- **DeltaControlStructure()**
  – Constructs a new DELTA closure node and initializes its internal variable list.
- **DeltaControlStructure acceptASTNode(Node_Duplication copier)**
  – Creates and returns a deep-copy of this closure node via the provided node-duplication visitor.
- **String getValueOfASTNode()**
  – Returns the string form of this AST node (used when printing or debugging).
- **List<String> getBoundVars()**
  – Retrieves the list of variable names bound in this closure.
- **void setBoundVars(List<String> vars)**
  – Replaces the current bound-variable list with the given one.
- **void addBoundVars(String var)**
  – Adds a single variable name to the closure's bound-variable list.
- **Stack<ASTNode> getBody()**
  – Returns the stack of AST nodes that make up this closure's body.
- **void setBody(Stack<ASTNode> codeBody)**
  – Sets or replaces the closure's body AST-node stack.
- **int getIndex()**
  – Returns this closure's unique numeric identifier.
- **void setIndex(int idx)**
  – Assigns a new identifier to this closure.
- **Environment getLinkedEnv()**
  – Retrieves the environment captured when this closure was created.

- **void setLinkedEnv(Environment env)**
  – Updates the linked (captured) environment for this closure.

Class: Environment.java

Introduction:

Models a chain of environments (variable-binding maps) to support closures and nested scopes. Each Environment holds its own name-to-ASTNode mappings and a reference to an optional parent environment for lookups.

Methods:

- **Environment()**
  – Initializes a fresh environment with an empty mapping.
- **Environment getParent()**
  – Returns the parent environment in the chain (or null if none).
- **void setParent(Environment parent)**
  – Links this environment to the given parent for inheritance of bindings.
- **ASTNode lookup(String key)**
  – Searches for the given name in this environment's map; if not found, recurses into the parent chain. Returns the bound ASTNode or null.
- **void addMapping(String key, ASTNode value)**
  – Associates the given name with an ASTNode in this environment's own map

Class: EtaRecursiveFixedPoint.java

Introduction:

Wraps a DeltaControlStructure to implement the “eta-expanded” (fixed-point) form of recursive functions. It marks a node for fixed-point handling without itself performing the recursion, deferring to the CSE machine’s evaluation logic to break loops.

Methods:

- **EtaRecursiveFixedPoint()**
  – Constructs a new ETA (fixed-point) node and tags its type accordingly.
- **String getValueOfASTNode()**
  – Returns the literal or symbolic representation of this ETA node.
- **EtaRecursiveFixedPoint acceptASTNode(Node_Duplication copier)**
  – Produces a deep-copy of this ETA node via the duplication visitor.
- **DeltaControlStructure getDelta()**
  – Retrieves the underlying DeltaControlStructure representing the body of the recursive function.
- **void setDelta(DeltaControlStructure delta)**
  – Sets or updates the embedded delta-closure for this fixed-point node.

Class: Node_Duplication.java

Introduction:

Provides deep-copy functionality for AST nodes. It ensures that each clone has independent child and sibling subtrees, and properly duplicates any embedded structures such as closures or ETA nodes.

Methods:

- **ASTNode takecopy(ASTNode original)**
  – Clones a generic ASTNode, including its entire child and sibling chains.
- **Beta_Condition_Eval takecopy(Beta_Condition_Eval source)**
  – Clones a Beta_Condition_Eval node, duplicating its then/else stacks and preserving semantics.

Class: Tuple.java

Introduction:

Models a Lisp-style tuple as a linked sequence of AST nodes. When printed, it renders itself in parenthesized, comma-separated form (e.g., (a, b, c)).

Methods:

- **Tuple()**
  – Constructs a new tuple node and sets its type to TUPLE.
- **String getValueOfASTNode()**
  – Builds and returns the string representation of this tuple's contents in tuple notation.
- **Tuple acceptASTNode(Node_Duplication copier)**
  – Produces a deep-copy of this tuple node via the duplication visitor, including its children.

Classes in the Abstract_Syntax_Tree package

Class: AST

Introduction:

Manages the full Abstract Syntax Tree (AST) for an RPAL program. It holds the root node, tracks pending delta (primitive operation) bodies, and provides functionality to standardize the tree and generate the corresponding delta control structures for evaluation.

Methods:

- **public void printAST()**
  – Traverses the AST from the root and prints each node (with indentation) in a human-readable form.
- **public void Standardize()**
  – Drives the standardization process, transforming the AST so that all operations and control structures follow a uniform, "standard" form for the CSE machine.
- **public DeltaControlStructure createDeltas()**
  – Walks the standardized AST and builds the root DeltaControlStructure, collecting all primitive-operation closures into a linked structure for the evaluator.
- **public boolean isASTStandardized()**
  – Returns true if Standardize() has already been run successfully, false otherwise.

Class: ASTNode

Introduction:

Represents a single node in the AST, storing its type, literal/value (if any), line number, and links to its first child and next sibling. Also supports deep copying of node subtrees via a duplication visitor.

Methods:

- **public ASTNode getChildOfASTNode()**
  – Returns the first (leftmost) child of this node, or null if none.
- **public void setChildOfASTNode(ASTNode child)**
  – Attaches the given node as this node's first child.
- **public ASTNode getSiblingOfASTNode()**
  – Returns this node's next sibling in the tree, or null if none.
- **public void setSiblingOfASTNode(ASTNode sibling)**
  – Links the given node as this node's next sibling.
- **public ASTNodeType getTypeOfASTNode()**
  – Retrieves the node's enum type (e.g., PLUS, IDENTIFIER).
- **public void setTypeOfASTNode(ASTNodeType type)**
  – Updates this node's type.
- **public String getValueOfASTNode()**
  – Returns the literal or identifier string stored in this node.
- **public void setValueOfASTNode(String value)**
  – Sets or changes the node's stored value.
- **public ASTNode acceptASTNode(Node_Duplication copier)**
  – Invokes the duplication visitor to produce a deep copy of this node (and its subtree).
- **public int getLineNumberOfSourceFile()**
  – Returns the original source-file line number where this node was parsed.
- **public void setLineNumberOfSourceFile(int line)**
  – Records or updates the source line number for error reporting.

Class: ASTNodeType

Introduction:

An enumeration of all possible AST node kinds (literals, operators, control structures, etc.). Each enum constant carries a human-readable "print name" used when displaying the AST.

Methods:

- **public String getPrintNameOfASTNode()**
  – Returns the string label associated with the enum constant, used by **printAST()** to show node types.

Conclusion

In this project, we have successfully designed and implemented an interpreter for the RPAL (Right-reference Pedagogic Algol) language, covering all core phases—lexical analysis, parsing, and evaluation of combinator expressions. Our interpreter:

- **Accurately tokenizes** RPAL source code into meaningful symbols
- **Builds a correct abstract syntax tree** via recursive-descent parsing
- **Evaluates expressions** using eager application of combinators and supports advanced features like higher-order functions

Throughout development, we encountered and overcame challenges in managing scope, handling function application without mutable state, and ensuring termination for well-formed programs. Rigorous testing on standard RPAL benchmarks demonstrated both correctness and reasonable performance