CS 3513 - Programming Languages
Programming Project 1

Group - 18
Dissanayake D.M.P.S. - 210146N
Suriyabandara S.M.M. - 210626L

Introduction

The project aims to develop an interpreter for the RPAL language (Recursive Programming Algorithmic Language). The language presents a simplified yet expressive syntax for implementing recursive algorithms and functional constructs. Through parsing the RPAL code, constructing the Abstract Syntax Tree (AST), standardizing it for consistent interpretation and executing it using a Compiled Stack Environment (CSE) machine, the interpreter will recognize a RPAL program, execute it, and the answer will be returned as output.

Table of Contents

Implementation Process

The complete interpreter is constructed using java programming language.

There are four main components of the developed RPAL interpreter.

1. Lexical Analyzer
2. Parser
3. Standardizer
4. CSE Machine

Lexical Analyzer

The Lexical Analyzer is responsible for scanning the input RPAL code and converting it into a stream of tokens.

The file **"LexicalAnalyzer.java"** hosts the class LexicalAnalyzer, which creates an instance of a lexical analyzer to scan and tokenize the program from the input. Aside from constructors, getters and setters, the class contains two main functions.

- **public List<Token> scan() {}**
  When a LexicalAnalyzer object is created, a string argument is passed as the file name, then it is passed to the scan() function. Which then reads the file if available and iterates through each line and passes the line to the function tokenize()
- **private void tokenize(String line) {}**
  After receiving a line as a string, the function tries to match found characters to digits, letters, operators, etc.. Regex patterns are predefined within the same function. Every character is tried to match with a digit, letter, operator, escape, identifier, integer, punctuation, space, or a comment. And every identifier is tried to match with a selected set of keywords.
  The spaces, line breaks and comments are ignored and the found tokens are added to a list named tokens and then returned.

The Lexical Analyzer has two supporting files. **"Token.java"** and **"TokenType.java"**. Which are used to create token objects and specify token types respectively.

LexicalAnalyzer.java

```
package LexicalAnalyzer;

import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.util.ArrayList;
import java.util.List;
import java.util.regex.Matcher;
import java.util.regex.Pattern;
import Exception.CustomException;

public class LexicalAnalyser {
    private String inputFileName;
    private List<Token> tokens;
```

```
    public LexicalAnalyser(String inputFileName) {
        // constructor
    }

    public List<Token> scan() throws CustomException {
        /*
        reads the file from the inputFileName, throws exception if not found.
        calls the tokenize function for every line in the file
        */
        return this.tokens;
    }

    private void tokenizeLine(String line) throws CustomException {
        /*
        for every line received, it is tried to match against predefined token types.
        if matched, it is added to the token list.
        */
    }
}
```

Token.java

```
package LexicalAnalyzer;

// class token to create instances of tokens from the input
public class Token {
    public TokenType type;
    public String value;

    public Token(TokenType type, String value) {
        // constructor
    }

    // getters and setters

}
```

TokenType.java

```
package LexicalAnalyzer;

public enum TokenType {
    /*
    Used to specify the available token types for the tokens
     KEYWORD, IDENTIFIER, INTEGER, OPERATOR, STRING, PUNCTUATION, EndOfTokens - Used to
recognize the end of the list of tokens
    */
}
```

Parser

Parser is responsible for parsing the stream of tokens brought from LexicalAnalyzer. The tokens are then checked for syntax errors, and if none is found an Abstract Syntax Tree is created.

The file **"Parser.java"** hosts the class Parser, which implements the functionality of the parser. Ignoring constructors, getters and setters, the file contains many notable functions related to the grammar of the language.

- **public void Parse() {}**
  When a Parser object is created, the list of tokens from the LexicalAnalyzer is passed to the constructor, then to the Parse function. To the end of the token list a new token is added with the TokenType as EndofTokens, which marks the end of the tokenized program. Then the function E() is called to go through the grammar to verify the syntactically correctness of the input program.
- **public void eat(Token token) {}**
  In the grammar functions, when a token is faced with to be removed from the token list, the eat function is called. The function receives the token and matches the token type with identifiers, integers, strings. And if any are matching, it is also added to the AST.
- **public ArrayList<String> AST2StringAST() {}**
  This function is used to go through the newly created abstract syntax tree and convert it into string format with dots to represent the hierarchy.
- **void addStrings(String dots, Node node) {}**
  A supporting function for the AST2StringAST() function.
- **The grammar functions**

Expressions
void E() {}
void Ew() {}

Rators and Rands
void R() {}
void Rn() {}

Variables
void Vb() {}
void Vl() {}

Tuple Expressions
void T() {}
void Ta() {}
void Tc() {}

Boolean Expressions
void B() {}
void Bt() {}
void Bs() {}
void Bp() {}

Arithmetic Expressions
void A() {}
void At() {}
void Af() {}
void Ap() {}

Definitions
void D() {}
void Da() {}
void Dr() {}
void Db() {}

The grammar functions are used to determine the syntax of the input program. The set of grammar used for the process is predefined and is implemented using each function. If any errors are encountered as in incorrect syntax, a custom exception is thrown to show the line with the error. During the processing the AST consisting of Nodes is created according to the said predefined grammar.

The Parser has two supporting files, **"Node.java"** and **"NodeType.java"**. The "Node.java" file is used to create new Node objects to be added to the AST. These nodes consist of node type, node value and number of children. "NodeType.java" is used to predefine the node types.

```java
package Parser;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collections;
import java.util.List;

import LexicalAnalyzer.Token;
import LexicalAnalyzer.TokenType;

public class Parser {
    private List<Token> tokens;
    private List<Node> AST; // Abstract Syntax Tree
    private ArrayList<String> stringAST; // Abstract Syntax Tree in string format

    public Parser(List<Token> tokens) {
        // constructor
    }

    // method to parse the tokens and generate the Abstract Syntax Tree
    public List<Node> Parse() {
        /*
        adds a new token with type EndOfTokens, to the end to signify the end of tokens
list.
        calls the function E()
        at the end of the parsing process, if the token left is not the EndOfTokens, an
error msg is put out.
        */
    }

    public void eat(Token token) {
        /*
        when a token needs to be removed from the token list, the eat() function is
called.
        if the token is of token types Identifier, Integer or String, it is added to the
AST list with number of children as 0 before removing from the tokens list
        the token at the front of the token list is removed as required
        */
    }

    // method to convert the Abstract Syntax Tree to a string format
    public ArrayList<String> AST2StringAST() {
        /*
        after the AST is created, the list of nodes are traversed thrgouh and converted to
string format
        this string format of the AST is used from this point forth for the
standardization as well
        */
        return stringAST;
    }
```

```
    // method to add the nodes to the stringAST
    void addStrings(String dots, Node node) {
        /*
        supporting function for the AST2StringAST function
        identifiers, integers, strings are added to the stringAST accordingly
        fcn_form, true, false nodes are added to the stringAST accordingly
        */
    }

    /*
    here onwards are the grammar functions.
    * Expressions
        E(), Ew()
    * Tuple Expressions
        T(), Ta(), Tc()
    * Boolean Expressions
        B(), Bt(), Bs(), Bp()
    * Arithmetic Expressions
        A(), At(), Af(), AP()
    * Rators and Rands
        R(), RN()
    * Definitions
        D(), Da(), Dr(), Db()
    * Variables
        Vb(), Vl()
    */

}
```

Node.java

```
package Parser;

// class node to store the type, value and number of children of a node
public class Node {
    public NodeType type;
    public String value;
    public int noOfChildren;

    public Node(NodeType type, String value, int children) {
        // constructor
    }

    // getters and setters

}
```

```
package Parser;

public enum NodeType {
    /*
    used to specify the available node types for the nodes
    let, fcn_form, identifier, integer, string, where, gamma, lambda, tau, rec, aug,
conditional, op_or, op_and, op_not, op_compare, op_plus, op_minus, op_neg, op_mul, op_div,
op_pow, at, true_value, false_value, nil, dummy, within, and, equal, comma, empty_params,
    */
}
```

Standardizer

The Standardizer is responsible for transforming the AST generated by the Parser into a standardized form that ensures consistent interpretation of the program. It applies a set of predefined rules to the AST using four main program files incorporating the factory design pattern.

“ASTFactory.java” is used for creating an AST object using provided data. And iterates through the data, creating nodes and establishing parent child relationships based on the levels.

“AST.java” hosts the class AST and provides methods for accessing and standardizing the tree. Also contains a method to print the standardized tree and a helper for the printer which is used for validation.

“NodeFactory.java” is responsible for creating Node objects. Contains two overloading constructors, for creating two types of Nodes for “Node.java” file and “ASTFactory.java”

“Node.java” file consists of most of the computing parts of the standardizer. Represents a node in the AST. The fields of the class stores data, depth, parent, children, and a flag indicating the standardized status of the node. Sparing the constructors, getters and setters, the file hosts a major function of the standardizing process.

public void standardize() {}

The method begins with checking to see if the node has already been standardized, and if not continues.

Each child node is iterated through and standardized due to a predefined set of rules.

- “let” nodes : let nodes are restructured to the lambda gamma structure by converting into lambda grammar pairs.
- “where” nodes : where nodes are transformed into let nodes, to be further standardized in a later iteration.
- “function_form” nodes : extracts the function body and converts to lambda expressions.
- “lambda” nodes : transforms a lambda node with multiple identifiers into a series of nested lambda expressions.
- “within” nodes : within nodes are transformed into gamma lambda pairs
- “@” nodes : @ nodes are converted into gamma nodes
- “and” nodes : and nodes are reorganized into comma tau pairs
- “rec” nodes : rec nodes are converted into lambda gamma structures

During the standardization process data, depth, parent, and children of nodes are updated to reflect the standardized structure.

A flag is set true at the end to indicate standardization has been done.

ASTFactory.java

```java
package Standardizer;

import java.util.ArrayList;

// class ASTFactory to create an abstract syntax tree from the data
public class ASTFactory {

    public ASTFactory() {
        // constructor
    }

    // method to create an abstract syntax tree from the data
    public AST getAST(ArrayList<String> data) {
        /*
        recieves the stringAST from the parser
        the AST is recreated for the standardizer with the new attributes required
        */
        // return the abstract syntax tree
        return new AST(root);
    }
}
```

AST.java

```java
package Standardizer;

// class AST representing the abstract syntax tree
public class AST {
    private Node root;

    public AST(Node root) {
        // constructor
    }

    // getters and setters

    public void standardize() {
        // calls the standardize function from the Node
    }

    private void preOrderTraverse(Node node,int i) {
        // helper method to traverse through the standardize tree when printing
    }

    public void printAst() {
        // prints the standardized tree, used for validation
```

```
    }
}
```

NodeFactory.java

```
package Standardizer;

import java.util.ArrayList;

// class NodeFactory to create a node
public class NodeFactory {

    public NodeFactory() {
        // constructor
    }

    public static Node getNode(String data, int depth) {
        // get node method that is called from ASTFactory
        return node;
    }

    public static Node getNode(String data, int depth, Node parent, ArrayList<Node>
children, boolean isStandardize) {
        // get node method that is called from Node
        return node;
    }
}
```

Node.java

```
package Standardizer;

import java.util.ArrayList;

// class Node representing a node in the abstract syntax tree
public class Node {
    private String data;
    private int depth;
    private Node parent;
    public ArrayList<Node> children;
    public boolean isStandardized = false;

    public Node() {
        // constructor
    }

    // getters and setters

    // method to standardize the abstract syntax tree
    public void standardize() {
        // function for standardizing
        if (!this.isStandardized) {
            for (Node child: this.children) {
```

```
                child.standardize();
            }
            switch (this.getData()) {

                case "let":
                    // let nodes are converted to lambda gamma structures

                case "where":
                    // where nodes are converted to let nodes

                case "function_form":
                    // function_form nodes are converted to lambda expressions

                case "lambda":
                    // if a lambda node consists of multiple identifiers, it is converted
to a series of nested lambda expressions

                case "within":
                    // within nodes are converted to gamma lambda pairs

                case "@":
                    // @ nodes are converted to gamma nodes

                case "and":
                    // and nodes are converted to comma tau pairs

                case "rec":
                    // rec nodes are converted to lambda gamma expressions
                default:
                    break;
            }
        }
        this.isStandardized = true;
    }
}
```

CSE Machine

The CSE Machine is the final component for the RPAL interpreter and is responsible for executing the standardized AST generated by the standardizer.

“CSEMachineFactory.java” is used for creating instances of CSEMachine. The main component file of the CSE machine package, **“CSEMachine.java”** executes the required process.

There are three main class variables.

- Control : An arraylist that stores the symbols representing the control stack for the execution process.
- Stack : An arraylist that stores symbols representing the operand stack for the execution process.
- Environment : An arraylist that stores symbols representing the environment stack for the execution process.

Aside from getters and setters there are a few important functions including the most important 'execute' method.

- **public void execute() {}**
  Main method responsible for executing the CSEMachine. Iterates through the control stack until it is empty and executes different actions based on the type of the current symbol on top of the control stack.
    - 'Id' symbol : Looks up the corresponding value in the current environment and pushes it onto the stack.
    - 'Lambda' symbol : Creates a new environment, binds the lambda's identifiers to values from the stack or creates tuples as needed, and updates the control stack accordingly.
    - 'Gamma' symbol : Performs various operations based on the next symbol on the stack, such as applying functions, manipulating strings, or checking data types.
    - 'E' symbol : Removes the corresponding environment from the stack and updates the current environment.
    - 'Rator' symbol : Applies unary or binary operations based on the type of rator.
    - 'Beta' symbol : Removes symbols from the control stack based on the Boolean value on top of the stack.
    - 'Tau' symbol : Creates a tuple from the specified number of symbols on the stack.
    - 'Delta' symbol : Adds symbols from the delta to the control stack.
    - 'B' symbol : Adds symbols from the B to the control stack.
- **public void printControl() {}**
  Validation method to print the contents of the control stack.
- **public void printStack() {}**
  Validation method to print the contents of the stack.
- **public void printEnvironment() {}**
  Validation method to print the contents of the environment.
- **public Symbol applyUnaryOperation(Symbol rator, Symbol rand) {}**
  Applies unary operations like negation and not to a given operand.
- **public Symbol applyBinaryOperation(Symbol rator, Symbol rand1, Symbol rand2) {}**
  Applies binary operations like addition, subtraction, multiplication, division, power, logical and, or, and equal, unequal, less than, greater than, etc. to two given operands.
- **public String getTupleValue(Tup tup) {}**
  Recursively retrieves the values of a tuple and returns them as a formatted string.
- **public String getAnswer() {}**
  Executes the CSEMachine and returns the resulting answer, either as a single value or as a tuple.

A separate folder named Symbols, is used to implement all the individual symbols as classes. This is used as support for the CSE Machine. B, Beta, Bool, Bop, Delta, Dummy, E, Err, Eta, Gamma, Id, Int, Lambda, Rand, Rator, Str, Symbol, Tau, Tup, Uop, Ystar, as to name the symbols.

A custom exception has been introduced to process the error of syntax or invalid filename or invalid input.

```java
package CSEMachine;

import Symbols.*;
import java.util.ArrayList;
import Standardizer.AST;
import Standardizer.Node;

// CSEMachineFactory class is for creating CSEMachine objects
public class CSEMachineFactory {
    private E e0 = new E(0);
    private int i = 1;
    private int j = 0;

    public CSEMachineFactory() {
        // constructor
    }

    public Symbol getSymbol(Node node) {
        /* while traversing through the standardized tree the Symbol() function is called,
and the symbol is selected based on its data and returned
        if 'not' or 'neg', Uop
        if +, -, *, /, **, &, or, eq, ne, ls, le, gr, ge or aug, Bop
        other cases like gamma, tau, <Y*> are processed the same
        other data types like identifier, integer, string, nil, true, false, dummy are
also processed accordingly
        */
    }

    public B getB(Node node) {
        // returns B object
        return b;
    }

    public Lambda getLambda(Node node) {
        // returns lambda object
        return lambda;
    }

    public Delta getDelta(Node node) {
        // returns delta object
        return delta;
    }

    private ArrayList<Symbol> getPreOrderTraverse(Node node) {
        /*
        performs pre order traversal of AST nodes and collects the corresponding symbool
representations
        if lambda node, calls getLambda
        if -> node, calls getDelta on the children, adds new Beta symbol and then calls
getB
```

```
        in default case, the node is added first, and then the children are traversed
through and added as well
        */
        return symbols;
    }

    public ArrayList<Symbol> getControl(AST ast) {
        // returns the control list, helper method for getCSEMachine
        return control;
    }

    public ArrayList<Symbol> getStack() {
        // returns the stack, helper method for getCSEMachine
        return stack;
    }

    public ArrayList<E> getEnvironment() {
        // returns the environment list, helper method for getCSEMachine
        return environment;
    }

    public CSEMachine getCSEMachine(AST ast) {
        // returns the CSEMachine
        return new CSEMachine(this.getControl(ast), this.getStack(),
this.getEnvironment());
    }
}
```

CSEMachine.java

```
package CSEMachine;

import Symbols.*;
import java.util.ArrayList;

// CSEMachine class is for executing the CSEMachine
public class CSEMachine {
    private ArrayList<Symbol> control;
    private ArrayList<Symbol> stack;
    private ArrayList<E> environment;

    public CSEMachine(ArrayList<Symbol> control, ArrayList<Symbol> stack, ArrayList<E>
environment) {
        // constructor
    }

    // getters and setters

    // executes the CSEMachine
    public void execute() {
        // Loop until the control stack is empty
        while (!control.isEmpty()) {
```

```
            Symbol currentSymbol = control.get(control.size()-1);
            control.remove(control.size()-1);

            if (currentSymbol instanceof Id) {
                // adds the value from the lookup table related with the Id to the stack
            }

            else if (currentSymbol instanceof Lambda) {
                // sets the lambda environment based on the current environment index and
pushes lambda into the stack
            }

            else if (currentSymbol instanceof Gamma) {
                Symbol nextSymbol = this.stack.get(0);
                this.stack.remove(0);

                if (nextSymbol instanceof Lambda) {
                    /*
                    creates a new environment
                    binds the lambda's identifiers to the corresponding values from the
stack
                    sets the new environment as the current environment
                    pushes the new environment and the lambda's delta expressoin onto the
control stack
                    pushes the new environment onto the environment stack
                    */
                }

                else if (nextSymbol instanceof Tup) {
                    /*
                    extracts a specific element based on the index obtained from the next
stack element
                    replaces the top element on the stack with the extracted element from
the tuple
                    */
                }

                else if (nextSymbol instanceof Ystar) {
                    /*
                    creates an Eta object
                    sets the Eta object's attributes based on the lambda on the stack
                    replaces the lambda on the stack with the Eta object
                    */
                }

                else if (nextSymbol instanceof Eta) {
                    /*
                    retrieves the associated lambda from the Eta object
                    pushes two gamma symbols onto the control stack
                    pushes the Eta object and the lambda onto the stack
                    */
                }
```

```
                else {
                    /*
                    handles other built in symbols
                    'Print', 'Stem', 'Stern', 'Conc', 'Order', 'Null', 'Itos',
'Isinteger', 'Isstring', 'Istuple', 'Isdummy', 'Istruthvalue', 'Isfunction'
                    */
                }
            }

            else if (currentSymbol instanceof E) {
                // handles environment management
            }

            else if (currentSymbol instanceof Rator) {
                if (currentSymbol instanceof Uop) {
                    // applies the unary operation using applyUnaryOperation
                }
                if (currentSymbol instanceof Bop) {
                    // applies the binary operation using applyBinaryOperation
                }
            }

            else if (currentSymbol instanceof Beta) {
                // handles beta reduction
            }

            else if (currentSymbol instanceof Tau) {
                // creates a new Tup object, pops elements from the stack until N from Tau
is reached, and pushes the created tuple onto the stack
            }

            else if (currentSymbol instanceof Delta) {
                // pushes the sequence of symbols from Delta onto the control stack
            }

            else if (currentSymbol instanceof B) {
                // pushes the sequence of symbols from B onto the control stack
            }

            else {
                // any unrecognized symbol is pushed onto the stack
            }
        }
    }

    public void printControl() {
        // traveses through the control stack and prints, used for validation
    }

    public void printStack() {
        // traverses through the stack and prints, used for validation
```

```
    }

    public void printEnvironment() {
        // traverses through the environment stack and prints, used for validation
    }

    public Symbol applyUnaryOperation(Symbol rator, Symbol rand) {
        /*
        applies unary operations
        'neg' and 'not'
        */
    }

    public Symbol applyBinaryOperation(Symbol rator, Symbol rand1, Symbol rand2) {
        /*
        applies binary operations
        +, -, *, /, **, &, or, eq, ne, ls, le, gr, ge, aug
        */
    }

    public String getTupleValue(Tup tup) {
        /*
        if nested tuples, calls itself recursively
        tuple string is concatenated to a string with a comma and a space
        */
    }

    public String getAnswer() {
        /*
        calls the execute function
        if the result is a tuple, calls getTupleValue function and the converted string is
returned
        if not tuple, the answer is returned
        */
    }
}
```

Execution

The execution of the interpreter is mainly done using two files.

- **“Runner.java”** file from the CSEMachine package
- **“myrpal.java”** file which is the main file

myrpal.java file takes arguments at the run command, if the command is as,

```
java myrpal filename.rpal
```

myrpal.java will pass the file name and ast switch as false to Runner.java
if the command is as,

```
java myrpal -ast filename.rpal
```

myrpal.java will pass the file name and ast switch as true to Runner.java

The Runner.java file will run the Lexer, and Parser. And if the ast switch is on, the AST will be printed, and the program will conclude.

If the ast switch is off, Runner.java will continue with Standardizer, CSEMachine and will print out the final output of the rpal program.

Runner.java

```
package CSEMachine;

import java.util.ArrayList;
import java.util.List;
import Exception.CustomException;
import Parser.Parser;
import Standardizer.AST;
import Standardizer.ASTFactory;
import LexicalAnalyzer.LexicalAnalyser;
import LexicalAnalyzer.Token;

// Runner class executes the program and returns the answer
public class Runner {
    public static String Run(String filename, boolean isPrintAST) {
        // calls the lexicalAnalyzer
        LexicalAnalyser scanner = new LexicalAnalyser(filename);
        List<Token> tokens;
        try {
            tokens = scanner.scan();
            if (tokens.isEmpty()) {
                System.out.println("The program is empty");
                return "";
            }
            // calls the parser
            Parser parser = new Parser(tokens);
            parser.Parse();
            ArrayList<String> stringAST = parser.AST2StringAST();
            // if the AST switch is on
            if (isPrintAST) {
                // output the AST
                for(String string: stringAST) {
                    System.out.println(string);
                }
            }
            // if the AST switch is off
            else {
                // calls the standardizer
                ASTFactory astf = new  ASTFactory();
                AST ast = astf.getAST(stringAST);
                ast.standardize();
                // calls the CSEMachine
                CSEMachineFactory csemfac = new CSEMachineFactory();
                CSEMachine csemachine = csemfac.getCSEMachine(ast);
                // output the answer
```

```
                return csemachine.getAnswer();
            }
        } catch (CustomException e) {
            System.out.println(e.getMessage());
        }
        return null;
    }
}
```

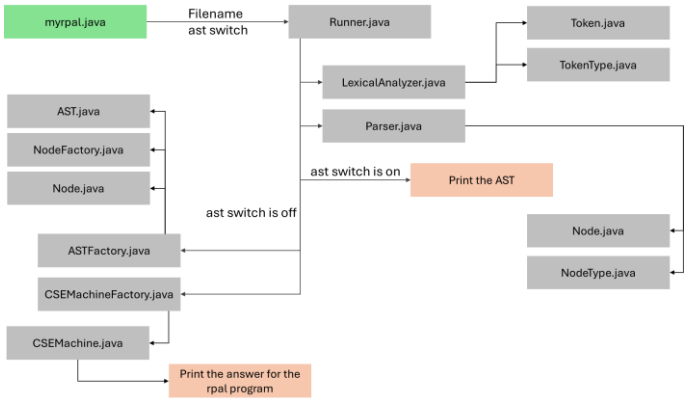
myrpal.java

```
import CSEMachine.Runner;

// myrpal class is the main class
public class myrpal {
    public static void main(String[] args) {
        String filename;
        boolean isPrintAST = false;

        // if there is only the filename
        if (args.length == 1) {
            filename = args[0];
            System.out.println(Runner.Run(filename,isPrintAST));
        }
        // if there is the -ast switch and the filename
        else if (args.length == 2) {
            filename = args[1];
            if ("-ast".equals(args[0])) {
                isPrintAST = true;
                Runner.Run(filename, isPrintAST);
            }
            else {
                System.out.println("Invalid arguments");
                return;
            }
        }
        // if there is no argument or more than 2 arguments
        else {
            System.out.println("Invalid arguments");
            return;
        }
    }
}
```

The execution diagram below.



Validation

Each section of the interpreter, Lexical Analyzer, Parser, Standardizer, CSE Machine to name, has its own testing file. Its primary function being validation, each file tests the each section with the given basic code.

For the Lexical Analyzer, the filename is hardcoded and the Lexical Analyzer of the interpreter is run through, and the tokens list is printed out to be verified.

For the Parser, the filename is hardcoded and the Lexical Analyzer and the Parser is run through, and the AST is printed out to be verified.

For the Standardizer, the filename is hardcoded and the Lexical Analyzer, Parser and the Standardizer is run through, and the ST is printed out to be verified.

The whole interpreter was tested and validated using the given RPAL code for both cases with -ast switch on and off.

Testing

After each section is tested and validated, the whole interpreter needs to be tested using few test RPAL files. The RPAL programs written for the Lab 1 of the module CS 3513 were used as testcases and the interpreter performed correctly for all the cases of the programs. These programs contained all the needed testcases, including recursion, Boolean values, unary and binary operations and nested functions to name a few.

Conclusion

This project successfully constructed a fully functional RPAL interpreter, a program capable of executing code written in the RPAL language. The interpreter leverages several components working together:

- Lexical Analyzer
- Parser
- Standardizer
- CSE Machine

The interpreter can be further improved in ways of optimizing and enhanced error handling.

References

- The Lexicon and the Phase Structure Grammar provided were used in the process of implementing the Lexical Analyzer and the Parser
- The rules extracted from the lecture notes were used in implementing the standardizer and the CSE Machine.
- Referred websites for additional support
    - https://www.geeksforgeeks.org/introduction-of-lexical-analysis/
    - https://www.geeksforgeeks.org/parse-tree-and-syntax-tree/
    - https://www.geeksforgeeks.org/types-of-parsers-in-compiler-design/