Partial RPAL Interpreter - Report

Index No. - 180192L
Name - N.M. Gamlath

The Partial RPAL interpreter takes an Abstract Syntax Tree generated from an RPAL program, standardizes it and evaluates it on a CSE machine.

This interpreter was written in Java. The class diagram can be found on Page 6.

To run the interpreter:

In the root of the directory, execute
`java myrpal <filename.txt>`
Where filename.txt is the path to the text file containing the Abstract Syntax Tree.

The Java files can be compiled into class files using the Make utility. The appropriate makefile is included under the root directory. Although the necessary class files are already included in the directory, they can be recompiled for verification.

How the execution occurs:

The command `java myrpal <filename.txt>` calls the main method of myrpal, which

- Calls the ASTreader class to read the given text file and return the root Node of AST
- Calls the standardize() method of the root Node, generating a standard tree
- Creates a CSE Machine and adds the control structures of the standard tree
- Calls evaluate() method of the machine, which displays the relevant Print output if any.

Package structure

- main.nodes - contains all the details of implementation of nodes in AST and ST.
- main.csemachine.elements - Implements the elements which appear on the Control and Stack of a CSE machine
- main.csemachine - Implements remaining CSE machine functionality

Function prototypes of significant functions in each class
Note that only structurally significant functions are included. (i.e.- no getters, setters, debug functions)

ASTreader

- public static void read (String filename)
  Reads the file and returns the root node of the AST, with all descendants connected.

- private static void getDepth(String line)
  Returns the depth of each node, corresponding to number of dots in the line

Node (abstract class)

- public ArrayList<Node> getChildren()
  Returns the ArrayList containing child nodes

- public void updateDepth()
  Recalibrates depths of all descendant nodes. This is called on the new root node after standardizing the AST

- public Node standardizedVersion()
  Returns the Node which should take its place in the standard tree

- protected abstract Node getReplacement()
  Returns the correct Node which should be substituted in its place, in the standard tree.

- protected abstract void attachStandardizedChildren(Node replacement)
  Attaches the standardized child nodes to the replacement Node.

AcceptedNode (extends Node)

This class represents all forms of Nodes which can be directly accepted into the tree without self standardization. (Eg - Tau and Conditional Nodes).
A String is stored to denote the specific type of node (e.g.- “tau”, “->”)

- @Override
  public Node getReplacement() -
  Returns a childless copy of itself

- @Override
  public void attachStandardizedChildren() -
  Attaches standardized versions of child nodes to a childless copy of itself.

Similarly, all Node subclasses override the attachStandardizedChildren() and getReplacement() as appropriate. These methods implement the standardization rules laid out in RPAL semantics, but do not standardize conditional, tau and unary / binary operators.

The list of all Node subclasses is as follows -

- AcceptedNode
- AndNode
- FcnFormNode
- GammaNode
- InfixNode
- LetNode
- LambdaNode
- RecNode
- WhereNode
- WithinNode

NodeFactory class

- public static Node createNode(String type, int depth)
  Instantiates the correct subclass of Node from the line in the AST.

ControlElement class

This class provides a template for any element that may be found on the control or the stack of an executing CSE Machine. The base class is used to store identifiers and values supported by RPAL (integer, string, truthvalue etc.)

- public void doWhenPopped(Machine m)
  Specifies the behaviour of the Element when popped off of the Control by the Machine. It may pop additional items off the Control or Stack, and make changes to the Machine's Environment structure. Therefore the Machine which calls this function passes itself as a parameter.
  The base class implements this method by pushing itself onto the stack. It will be evaluated / bound and removed from the stack by another type of element, during execution.

Subclasses of ControlElement and their specific implementation details are as follows

- *DeltaElement*
  This is never directly popped off the control, so does not implement the doWhenPopped method.

- *BetaElement*
  Its doWhenPopped implementation is in accordance with Rule 8, which replaces the delta-delta-beta configuration on the control with the corresponding expanded delta structure.

- *BinaryOpElement*
  Its doWhenPopped implementation pops two operands off the stack and pushes the result, in accordance with Rule 6.

- *EtaElement*
  This is only found on the Stack and never on the control. It encapsulates a lambda element and is used in recursion. It is added and removed by the gamma element, and so is dealt with inside the gamma's doWhenPopped() implementation.

- *ExpElement*
  Its doWhenPopped implementation causes the current environment to be exited and the machine set to the correct environment.

- *GammaElement*
  Its doWhenPopped implementation contains cases to handle each type of element that could possibly be popped off the stack.

- *IdentifierElement*
  Identifiers may be user defined (e.g - fun x = x + 3) or reserved by RPAL (eg- Print). The doWhenPopped() implementation pushes reserved identifiers onto the stack, where they will later be popped by a Gamma element, and, for other identifiers, looks up their binding in the current environment and pushes it onto the stack.

- *LambdaElement*
  Stores env, index and bindings. Index and bindings are set at the time of its creation inside the ControlStructure. Its doWhenPopped() implementation sets the environment to the current one, and pushes it to the stack.

- *TauElement*
  Stores the number of children the Tau node had. Its doWhenPopped implementation pops that number of children off the stack and pushes a tuple containing them, in accordance with Rule 9.

- *UnaryElement*
  Its doWhenPopped implementation pops an operand off the stack and pushes the result of applying the unary operator, in accordance with Rule 7.

ControlElementFactory

- public static ControlElement createElement(Node n)
  Returns the appropriate ControlElement based on the type and other features of the Node.

- private static isUnaryOp(String s)
  Checks whether a string is a unary operator - used to determine whether to create a UnaryOpElement out of a certain Node.

- private static isBinaryOp(String s)
  Checks whether a string is a unary operator - used to determine whether to create a BinaryOpElement out of a certain Node.

ControlStructure

This class represents a stack of ControlElements. Its methods are called by the ControlStructureGroup class.

- public void addElement (ControlElement ce)
- public ControlElement removeLastElement()

ControlStructureGroup

- public int createControlStructure(Node n)
  This generates a control structure relevant to the subtree whose root is at the Node n, and calls the addToControlStructure method on it in order to fill it with ControlElements.

- public void addToControlStructure(ControlStructure cs, Node n)
  This performs a preorder traversal of the subtree rooted at N and adds all the nodes to the control structure in the manner defined for each node's type. The ControlElementFactory's create method is called here.

Environment

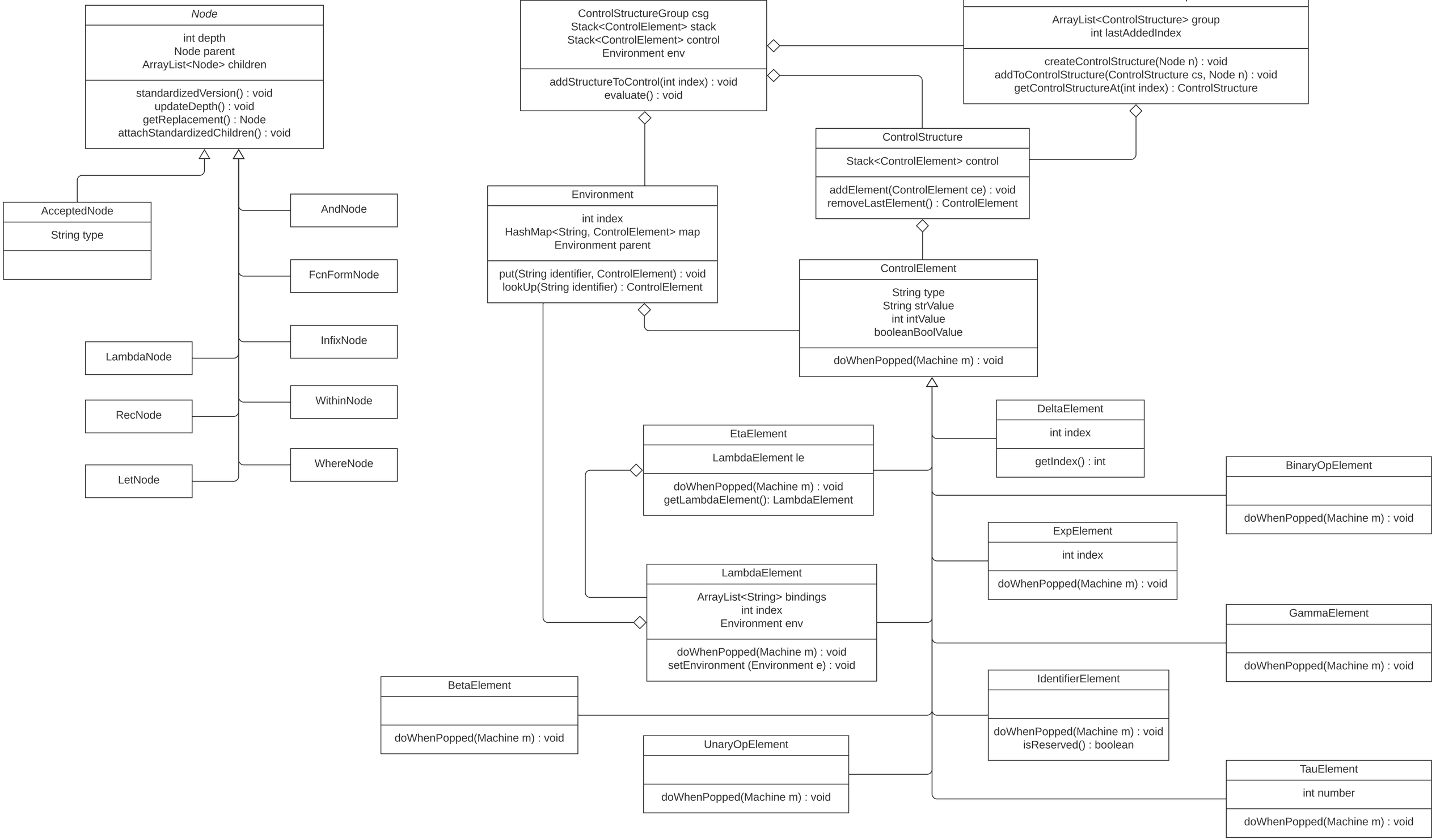
- public ControlElement lookUp(String identifier)
  Searches this environment and its ancestors for the value (ControlElement) corresponding to the identifier

- public void put(String identifier, ControlElement value)
  Stores an identifier and its relevant value as a key-value mapping in a HashMap.

Machine

- public void addStructureToControl(int index)
  Adds the structure located in the ControlStructureGroup at a certain index, to the control.

- public ControlElement evaluate()
  Until the control is empty, pops items off the top and calls their doWhenPopped method, passing the machine itself as a parameter. The elements can then execute the necessary actions on the control, stack and environment of the machine.


Node
int depth
Node parent
ArrayList<Node> children
standardizedVersion() : void
updateDepth() : void
getReplacement() : Node
attachStandardizedChildren() : void
AcceptedNode
String type
AndNode
FcnFormNode
InfixNode
LambdaNode
WithinNode
RecNode
WhereNode
LetNode
Machine
ControlStructureGroup csg
Stack<ControlElement> stack
Stack<ControlElement> control
Environment env
addStructureToControl(int index) : void
evaluate() : void
ControlStructureGroup
ArrayList<ControlStructure> group
int lastAddedIndex
createControlStructure(Node n) : void
addToControlStructure(ControlStructure cs, Node n) : void
getControlStructureAt(int index) : ControlStructure
ControlStructure
Stack<ControlElement> control
addElement(ControlElement ce) : void
removeLastElement() : ControlElement
Environment
int index
HashMap<String, ControlElement> map
Environment parent
put(String identifier, ControlElement) : void
lookUp(String identifier) : ControlElement
ControlElement
String type
String strValue
int intValue
booleanBoolValue
doWhenPopped(Machine m) : void
DeltaElement
int index
getIndex() : int
EtaElement
LambdaElement le
doWhenPopped(Machine m) : void
getLambdaElement(): LambdaElement
BinaryOpElement
doWhenPopped(Machine m) : void
ExpElement
int index
doWhenPopped(Machine m) : void
LambdaElement
ArrayList<String> bindings
int index
Environment env
doWhenPopped(Machine m) : void
setEnvironment (Environment e) : void
GammaElement
doWhenPopped(Machine m) : void
BetaElement
doWhenPopped(Machine m) : void
IdentifierElement
doWhenPopped(Machine m) : void
isReserved() : boolean
UnaryOpElement
doWhenPopped(Machine m) : void
TauElement
int number
doWhenPopped(Machine m) : void