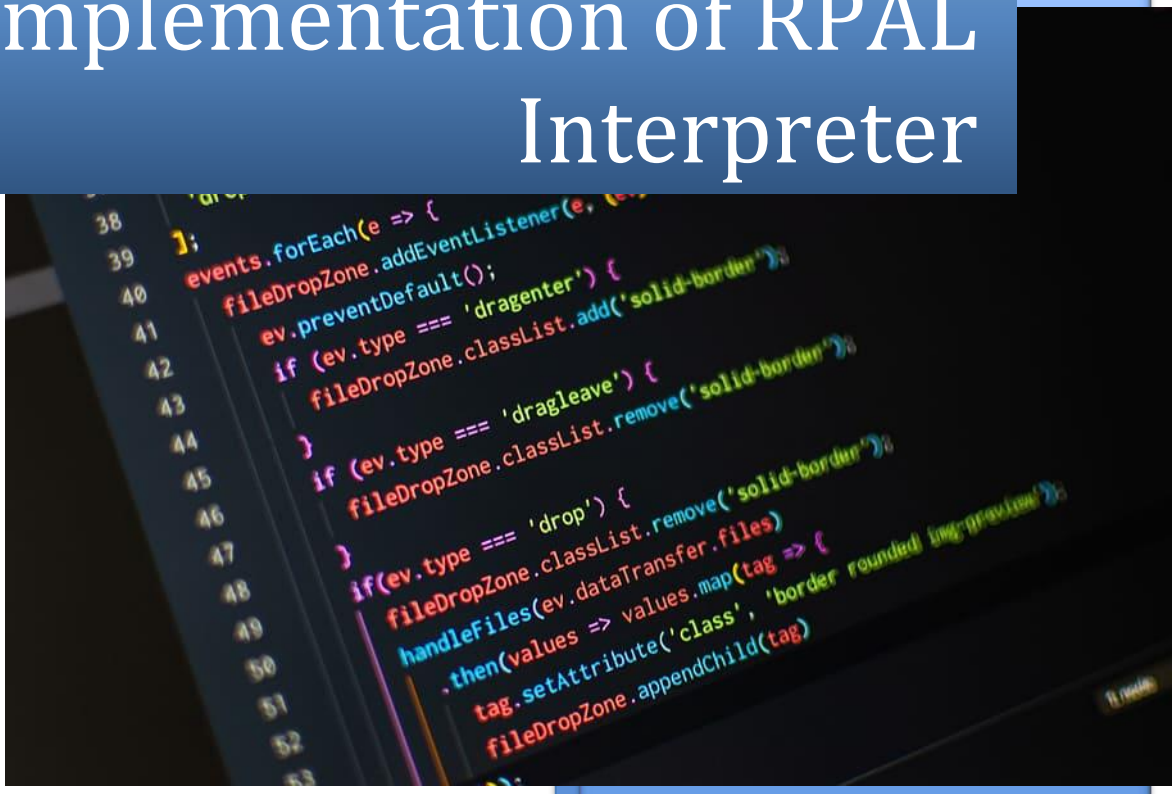
Programming Project - Implementation of RPAL Interpreter



Team Syntax Terrors

220311C – Kasige N. D.

220469P – Perera M. R. S.

Introduction

RPAL (Right-reference Pedagogical Algorithmic Language) is a functional programming language which makes it easy to understand key functional programming concepts. In this project our main objective is to design an interpreter for RPAL using python programming language.

Main Classes Used

Add headings (Format > Paragraph styles) and they will appear in your table of contents.
For a better usability and maintainability of the design, we have created separate classes for main components such as **ControlStructureBuilder**, **CSEMachine**, **Lexer** and **AST**.
To support the main components and the functionality of the interpreter, we have created classes for basic components such as **Node**, **Tokens**, **Closure** and **ClosureEta.**

Main Functions

The main functions used in designing the interpreter are listed below:

1) **_preorder_flatten()**
This function performs the preorder traversal of the tree-like node structure and returns the flatten list of values.

2) **_apply_beta()**
This function handles conditional execution. If the control stack pops the symbol 'β' (beta), this function is called. It checks the top of the stack:

- If the value is true, it runs the true control structure.
- If the value is false, it runs the false control structure.

3) **lookup()**
In this function, we use to search for previously created environments from the env list. If the variable is found, it returns its value; otherwise, it raises an error.

4)**_apply_aug()**
In this function, it checks the next element in the stack.
If the top of the stack is 'nil', it creates a new 1-element tuple from the next value.
If it's a tuple, it appends the next value to that tuple and pushes the result back on the stack.

5)Type Identification Functions
RPAL supports six data types: Integer, Truthvalue, String, Tuple, Function, and Dummy.
The following functions return true or false when applied to a value, based on its type:

- **_apply_istuple()**
- **_apply_isinteger()**
- **_apply_istruthvalue()**
- **_apply_isstring()**

- **_apply_isfunction()**
- **_apply_isdummy()**

6) There are some other functions in the Rpal programming language and they are implemented as below functions.

- **_apply_stem() -** This function returns the first letter of the given string
- **_apply_stern() -** This function returns the rest of the letters in string except the first letter.
- **_apply_conc() -** This function concatenates two strings and returns the new string.

Structure of the program

In our project, we implement all the components; lexical analyzer, parser, standardize tree and CSE machine using python language.

1) Lexical Analyzer

Lexical analysis is the first step of the interpreter. We use the lexical analyzer to read the source code and convert it into tokens so that the parser can understand the code.
We generate the lexical analyser using RPAL's Lex.

The **lexical analyser** removes all whitespaces and comments from the code. Then, it identifies different token types such as identifiers, keywords, strings, operators, etc.
After generating the sequence of tokens, it passes them to the parser for further processing.

2) Parser

Parsing is the second phase of language translation. The tokens generated by the lexical analyser, are passed to the **parser** as input.

Each grammar rule is coded as a recursive function to implement the **recursive descent parsing**. And each function returns a Node object having value and the list of child nodes as attributes, which forms **abstract syntax tree (ast)** as the final output of parsing.

3) Standardize Tree

The abstract syntax tree (ast) generated in the parsing stage will be further generalized by standardizing it. And the **standardized tree** generated here will be flattened into control structures to be given to the CSE machine for evaluation.

To make the evaluation more efficient and faster, we limited standardizing steps only to the `'and'`, `'lambda'`, `'rec'`, `'@'`, `'function_form'`, `'within'`, `'where'` and `'let'`

nodes. As per CSE machine evaluation rules, unary operators, binary operators, `'tau'` nodes, conditional operators and `','` nodes were not standardized.

4) CSE Machine

The CSE machine is a method for evaluating expressions using three components: Control, Stack, and Environment.

To run the CSE machine, we first standardize the expression tree and use it as input to the machine. The machine then evaluates the tree to produce the final result of the expression.

First, we generate the control structures using the **ControlStructureBuilder.py** file. This step displays the control structures that are generated before evaluation begins.

```
PS F:\My Projects\rpal_interpreter> python myrpal.py ./testing_scripts/sample.txt

 Control Structures for the CSE Machine :

 δ0 => ['gamma', ('lambda', 1, '<ID:Sum>'), ('lambda', 2, '<ID:A>')]
 δ1 => ['gamma', '<ID:Print>', 'gamma', '<ID:Sum>', ('τ', 5), '<INT:1>', '<INT:2>', '<INT:3>', '<INT:4>', '<INT:5>']
 δ2 => ['gamma', ('lambda', 3, '<ID:Psum>'), 'gamma', 'Y*', ('lambda', 4, '<ID:Psum>')]
 δ3 => ['gamma', '<ID:Psum>', ('τ', 2), '<ID:A>', 'gamma', '<ID:Order>', '<ID:A>']
 δ4 => [('lambda', 5, ['<ID:T>', '<ID:N>'])]
 δ5 => ['δ6', 'δ7', 'β', 'eq', '<ID:N>', '<INT:0>']
 δ6 => ['<INT:0>']
 δ7 => ['+', 'gamma', '<ID:Psum>', ('τ', 2), '<ID:T>', '-', '<ID:N>', '<INT:1>', 'gamma', '<ID:T>', '<ID:N>']
 δ8 => [('τ', 2), '<ID:T>', '-', '<ID:N>', '<INT:1>']
```

Next, we pass the generated control structures to the **CSE_Machine.py** file, which evaluates them using the control, stack, and environment mechanisms.

All the rules for the CSE machine are defined in this file. Binary and unary operations, as well as the environment lookup function, are implemented in a separate file called **Utils.py**. When a lambda expression is encountered, a lambda closure is created using the **Closure.py** file.

```
Control: ['e0', 'gamma', ('lambda', 1, '<ID:Sum>'), ('lambda', 2, '<ID:A>')]
Stack: ['e0']
Enviornment: [{}]

Control: ['e0', 'gamma', ('lambda', 1, '<ID:Sum>')]
Stack: ['e0', <Closure δ2 λ <ID:A> env={}>]
Enviornment: [{}]

Control: ['e0', 'gamma']
Stack: ['e0', <Closure δ2 λ <ID:A> env={}>, <Closure δ1 λ <ID:Sum> env={}>]
Enviornment: [{}]

Control: ['e0', 'e1', 'gamma', '<ID:Print>', 'gamma', '<ID:Sum>', ('τ', 5), '<INT:1>', '<INT:2>', '<INT:3>', '<INT:4>', '<INT:5>']
Stack: ['e0', 'e1']
Enviornment: [{}, {'<ID:Sum>': <Closure δ2 λ <ID:A> env={}>}]

Control: ['e0', 'e1', 'gamma', '<ID:Print>', 'gamma', '<ID:Sum>', ('τ', 5), '<INT:1>', '<INT:2>', '<INT:3>', '<INT:4>']
Stack: ['e0', 'e1', 5]
Enviornment: [{}, {'<ID:Sum>': <Closure δ2 λ <ID:A> env={}>}]

Control: ['e0', 'e1', 'gamma', '<ID:Print>', 'gamma', '<ID:Sum>', ('τ', 5), '<INT:1>', '<INT:2>', '<INT:3>']
Stack: ['e0', 'e1', 5, 4]
Enviornment: [{}, {'<ID:Sum>': <Closure δ2 λ <ID:A> env={}>}]

Control: ['e0', 'e1', 'gamma', '<ID:Print>', 'gamma', '<ID:Sum>', ('τ', 5), '<INT:1>', '<INT:2>']
Stack: ['e0', 'e1', 5, 4, 3]
Enviornment: [{}, {'<ID:Sum>': <Closure δ2 λ <ID:A> env={}>}]

Control: ['e0', 'e1', 'gamma', '<ID:Print>', 'gamma', '<ID:Sum>', ('τ', 5), '<INT:1>']
Stack: ['e0', 'e1', 5, 4, 3, 2]
Enviornment: [{}, {'<ID:Sum>': <Closure δ2 λ <ID:A> env={}>}]

Control: ['e0', 'e1', 'gamma', '<ID:Print>', 'gamma', '<ID:Sum>', ('τ', 5)]
Stack: ['e0', 'e1', 5, 4, 3, 2, 1]
Enviornment: [{}, {'<ID:Sum>': <Closure δ2 λ <ID:A> env={}>}]
```

Makefile

In makefile we have included commands to run an rpal file, to get the abstract syntax tree(ast) and to get standardized tree(st).

```
# Run a specific file
run:
    $(PYTHON) $(INTERPRETER) $(file)

# Print AST for a specific file
ast:
    $(PYTHON) $(INTERPRETER) -ast $(file)

# Print standardized AST for a specific file
st:
    $(PYTHON) $(INTERPRETER) -st $(file)
```

In addition to that we included commands to automate tests for evaluating programs (`make run-all`), getting ast along with result of the program (`make run-all-ast`) and getting st along with the result of the program (`make run-all-st`).

Sample Program

Usage of the interpreter can be understood using the following RPAL program.

```
let Sum(A) = Psum (A,Order A )
where rec Psum (T,N) = N eq 0 -> 0
| Psum(T,N-1)+T N
in Print ( Sum (1,2,3,4,5) )
```

To run and evaluate the program we use the following command.
`python myrpal.py ./testing_scripts/sample.txt`

```
PS F:\My Projects\rpal_interpreter> python myrpal.py ./testing_scripts/sample.txt
15
PS F:\My Projects\rpal_interpreter>
```

To get the AST (Abstract Syntax Tree) we use -ast switch in the above command.
`python myrpal.py -ast ./testing_scripts/sample.txt`

```
let
.function_form
..<ID:Sum>
..<ID:A>
..where
...gamma
....<ID:Psum>
....tau
.....<ID:A>
.....gamma
......<ID:Order>
......<ID:A>
...rec
....function_form
.....<ID:Psum>
.....,
......<ID:T>
......<ID:N>
.....->
......eq
.......<ID:N>
.......<INT:0>
......<INT:0>
......+
.......gamma
........<ID:Psum>
........tau
.........<ID:T>
.........-
..........<ID:N>
..........<INT:1>
.......gamma
........<ID:T>
........<ID:N>
.gamma
..gamma
...<ID:Sum>
...tau
....<INT:1>
....<INT:2>
....<INT:3>
....<INT:4>
....<INT:5>
15
PS F:\My Projects\rpal_interpreter>
```

To get the standardized tree we need to add -st switch to the above command
`python myrpal.py -st ./testing_scripts/sample.txt`

```
gamma
.lambda
..<ID:Sum>
..gamma
...<ID:Print>
...gamma
....<ID:Sum>
....tau
.....<INT:1>
.....<INT:2>
.....<INT:3>
.....<INT:4>
.....<INT:5>
.lambda
..<ID:A>
..gamma
...lambda
....<ID:Psum>
....gamma
.....<ID:Psum>
.....tau
......<ID:A>
......gamma
.......<ID:Order>
.......<ID:A>
...gamma
....<Y*>
....lambda
.....<ID:Psum>
.....lambda
......,
.......<ID:T>
.......<ID:N>
......->
.......eq
........<ID:N>
........<INT:0>
.......<INT:0>
.......+
........gamma
.........<ID:Psum>
.........tau
..........<ID:T>
..........-
...........<ID:N>
...........<INT:1>
........gamma
.........<ID:T>
.........<ID:N>
15
PS F:\My Projects\rpal_interpreter>
```

Discussion

In the process of developing the interpreter for RPAL, using separate classes for main components and basic components had a huge positive impact on both designing and bug-fixing stages, due to its high usability and maintainability. In addition to that, using **recursive descent parsing** over **table-driven parsing** as the parsing technique, was helpful to understand and implement the parser easily, rather than hard coding the correct grammar rule(in a table), to be used for each possible input variable and non-terminal symbol combination. Furthermore, usage of **ControlStructureBuilder** class, decouples the control structure generation from both program evaluation and standardized tree generation. And the **evaluate()** function provides an interface to evaluate the rpal program, by completing all the phases in language translation with the use of the system being designed.

Conclusion

In this project, we successfully designed and implemented the interpreter for Rpal Language using high level programming language python. It consists of a few components: lexical analyser, parser, abstract syntax tree generator, Standardizer and CSE machine. Through the use of well-structured classes and functions including support for different data types, closures, and control structures. We effectively demonstrated how functional programming concepts can be interpreted and executed. The project helped us apply theoretical concepts in practice and gave us a clear understanding of the internal workings of an interpreter.