Ruse Exercises for Instructors

Version 6.0.1

Philip Schielke

April 4, 2016

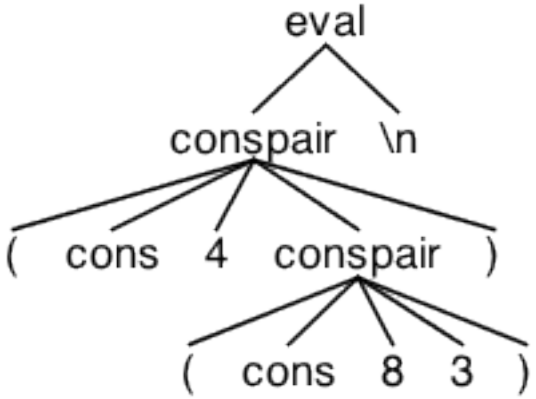
Philip Schielke <philip.schielke@concordia.edu>

This document describes a set of exercises that instructors may have their students implement within the supplied RUSE interpreter. A brief introduction to the language and codebase is described and a set of exercises is presented.

Solutions to exercises may be provided to verified instructors by contacting the author via email.

1 Introduction

You will be working on an interpreter for RUSE, a subset of Scheme/Racket, which is built on the ANTLR lexing and parsing system. The system will automatically produce a parse tree for each inputed RUSE expression Once the tree is built, additional Java code (called visitors) will look at each node of the tree and produce the correct result. This is a much more flexible way to build interpreters than including action code within the grammar itself For example if the grammar matches the following `(cons 4 (cons 8 3))`, the interpreter would generate a parse tree that looks like this:



Later, when that tree is evaluated, the interpreter will create a Pair object with 8 and 3, and later a Pair object of 4 and the pair 8, 3.

2 Current Language

Currently the interpreter supports the following:

- `+ - * /`: with two operands only:
  `(+ 3 4)` -> 7, `(-5 4)` -> 1
- `= < >`: relational operators, equals, less than, and greater than (2 operands only):
  `(= 1 1)` -> `#t`, `(> 4 5)` -> `#f`
- `and or not`: logical and, or (two operands), and not (negation, 1 operand):
  `(and (= 1 1) (= 1 2))` -> `#f`,
  `(or (= 1 1) (= 1 2))` -> `#t`,
  `(not (= 1 1))` -> `#f`
- `if`: a simple if conditional (if <relational expr> <then expr> <else expr>)
  `(if (= 1 1) "yes" "no")` -> `"yes"`,
  `(if (= 1 0) "yes" "no")` -> `"no"`,
  `(if (and (> 1 0) (< 0 1)) (+ 1 2) (+ 3 4))` -> 3
- `cond`: a more complex conditional, like a series of `if-else` statements, or a `switch` statement.
  `(cond (test1 (action1)) (test2 (action2)) ... (testn (actionn)))`
  Each test is evaluated in order. The action associated with the first test that evaluates to `#t` is evaluated and its result is the result of the `cond`. All other actions are ignored. For example

  ```
  (cond ((= 1 0) (+ 1 2))
        ((= 1 1) (+ 1 3))
        ((= 2 2) (+ 1 4)))
  ```

  would return 4, because `(= 1 1)` is the first test that evaluates to true. The corresponding action `(+ 1 3)` evalutes to 4.
- `cons car cdr`: work just like on the previous assignment
- `list`: this creates a list of items. A list is a special form of a cons pair, where the rightmost value of the list is the empty list. For example:
  `(list 4 "hello" (+ 3 4))` -> `(4 "hello" 7)` Empty list, `'()` is also matched as a literal.
- `list?`: Checks to see if its argument is a list.
  `(list? (list 1 4))` -> `#t`, `(list? (cons 'a 'b))` -> `#f`.
- `empty?`: Checks to see if a list is empty.
  `(empty? (list 1 2))` -> `#f`, `(empty? (cdr (list 'a)))` -> `#t`.

- `zero?`: Checks to see if its argument is 0.
  `(zero? (- 5 5))` -> `#t`, `(zero? (- 5 4))` -> `#f`.

- `'()`: This is the empty list. A list with no items in it.

- `lambda`: Lambda is used to create anonymous functions. The format is:
  `(lambda (<paramaters>) <function body>)`
  Simply entering a lambda expression into the interpreter will create a function that then disappears. In this example we are creating a function to square a value with lambda, and applying that function to an argument, 7:
  `((lambda (x) (* x x)) 7)`
  The result is 49.

- `define`: is used to set the values of global identifiers or assign names to functions. For example, `(define a 3)` sets the value of symbol 'a' to be 3. Thus if you enter this into the interpreter `a` you would get the value `3`. `(define b "Hello World")` sets the value of the symbol 'b' to be the string `"Hello World"`. `(define myPair (cons 1 2))` would set the symbol `myPair` to be `(1 . 2)`.
  The format for using `define` to create a function is this:
  `(define (<function name> <list of parameters>) <function body>)`
  So, for example, to write a function called `square` that squares the value of input `x` we would write this:
  `(define (square x) (* x x))`
  To call this function: `(square 5)` -> `25`.

- `let`: creates local variables and binds them to values. For example:
  `(let ((x 4)) (+ x 3))` would return 7, because `x` is bound to 4 and then added to 3.

- `set!`: `(set! id expr)`, evaluates expr and installs the result into the location for id, which must be bound as a local variable or defined as a top-level variable

- `eval, apply, displayln, load`

3 Code Organization

You have been given a NetBeans project containing code for an interpreter that handles all the constructs above. That code includes:

1. `"ruse.g4"` - This is the ANTLR grammar file for RUSE
2. `"Pair.java"` - Code to implement the Pair type (also used for lists)
3. `"Symbol.java"` – A class to hold RUSE symbols
4. `"SymbolTable.java"` - Code to implement the symbol table used for local variables in functions and globals set by `define`
5. `"Function.java"` - Creates a class Function to store details about RUSE functions.
6. `"EmptyList.java"` – A class for the singleton RUSE empty list object.
7. `"Void.java"` – A special class for RUSE void.
8. `"EvalVisitor.java"` - This file contains all the code to visit nodes of the parse tree. You will do most of your work in here.

4 Exercises

1. Logical and/or do not "short circuit", but they should. Currently, both expressions are evaluated first and then the logical operation is performed on the two results. This is bad. For example, `(and (= 0 1) (= (/5 0) 0))` should simply return `#f`, but currently it crashes the parser. If the first expression of an `and` expression is false, the second expression should not be evaluated. Likewise, if the first expression of an `or` expression is true, the second one should not be evaluated.

2. Implement the Boolean literals `#t` (true) and `#f` (false). For example, you could use `#t` like this: `(if #t 1 0)`. These are already matched in the lexer, but you will have to match them in the parser as well (additional rules in `expr`), as well as adding visitor functions to `"EvalVisitor.java"`. These functions should return `Boolean` values `true` or `false`.

3. Add modulo (%) to the language. The % symbol is already matched by the lexer, but nothing is done with it. You will add a rule to `expr` in `"ruse.g4"`, and will need to a visitor function for modulo in `"EvalVisitor.java"`. I suggest you look at the code for divide '/', and use that as your basis for doing modulo.

4. You will extend the language to make the addition of strings perform string concatenation. For example `(+ "Hello " "world!")` would return the string `"Hello world!"`. Currently, trying to add two strings just results in a Java exception being thrown. You will need to modify the `VisitSimplePlus` visitor and possibly use the `instanceOf` operator, to check the types of the children under the '+' operator. Also, to receive full credit, you will need to get rid of the extra "" (double quote) characters within your new string.

5. Currently, the `list` operation can take any number of arguments. However, +, -, *, and / only take two arguments. Modifiy the grammar in `"ruse.g4"` and visitor functions in `"EvalVisitor.java"` to allow those arithmetic operations to accept any number of arguments. Use the code for `list` as an example of how to do this.

6. By default, this language binds variables statically. You will modify the interpreter to make variables within functions bind dynamically. Given this code:

   ```
   (define a 4)
   (define (foo x) a)
   (define (bar a) (foo a))
   (bar 7)
   ```

   gives, 4, since the binding of `a` in `foo` must be the `a` in global scope since there is no local variable `a`. If this were bound dynamically, the above code would return 7, since the `a` in `foo` would then be bound to the local variable `a` in the function that called it, namely `bar`, where `a` is bound to the value 7.

7. Write a recursive factorial function in RUSE using the `if` construct. You will discover that this correctly written recursive function doesn't work properly. (You will basically get the equivalent of an infinite loop causing a Java stack overflow.) Fix this. (You might consider how the `if` statement should work. Think about how you fixed the `and` and `or` operations earlier.)

8. Lazy evaluation. Consider this RUSE code, `(define a (* b 3))`. If the variable `b` is not defined then this code will result in an error. Lazy evaluation says don't evaluate the expression `(* b 3)` until the variable `a` is actually needed. You'll add a new version of define called `define-lazy`. This version of define will not evaluate the expression being assigned until it is actually needed. For example, consider this code:

   ```
   (define-lazy a (* b 3))
   (define b 4)
    a
   ```

   This would return 12. If we then did:

   ```
   (define b 5)
   a
   ```

   This would return 15. Here are the steps you need to take:

   - add a matching rule for the keyword define-lazy to the lexer portion of `"ruse.g4"`
   - add a rule to `define` in `"ruse.g4"`. This will look a lot like the rule labeled `#DefSimple`. Label it `#DefLazy`.
   - add a vistitor to `"EvalVisitor.java"`, `visitDefLazy`. This rule will look a lot like `visitDefSimple`, but will not visit `ctx.expr()`, but instead will put the expression directly into the symbol table.
   - Modify the function `visitJustID`. If the Object stored in the symbol table is an expression, you need to visit it before returning the value from `visitJustID`.

9. Implement a floating-point type. This will work in a way very similar to the integer type. Floating point types should work in arithmetic expressions (not %) and relational expressions as well. Floating point values are already matched by the lexer. See the `FLOAT` matching rule in `"ruse.g4"`. You will need to add a matching rule to `expr` in `"ruse.g4"`, and will need to a visitor function for floats in `"EvalVisitor.java"`, as well as modifying the other visitor functions for arithmetic and relational operators in that file.

10. Type checking. This interpreter does very little type-checking. For example, if you execute `(car 5)` or `(+ (cons 2 3) 7)`, the interpreter simply crashes with a Java exception. Your goal is to make type-checking work at least a little bit better. You can simply use try and catch to avoid crashing the parser when there is a type error. Or, you

can check for the correct types before performing an operation. For example, consider the code in `visitCar`. This function evaluates the argument to `car`, and assumes it is a Pair by casting it to pair. If the object returned by `visit` is an `Integer`, for example, this will cause the interpreter to crash because it cannot conver `Integer` to `Pair`. If you assign the result of `visit` to a variable of type `Object` and then check to make sure it is `Pair` you could exit gracefully if it is not a `Pair`. You should check for errors in the following: car, cdr, arithmetic operations, relational operations, if, cond, and, or, not. I will check several expressions for each of these.

11. Implementing the `map` operator. The first argument to `map` is a function and the remaining arguments are lists of parameters to that function. For example `(map + '(1 2 3) '(4 5 6))` takes each element from the first list and adds it to the corresponding element in the second list, returning another list. In this case you would get `'(5 7 9)`. Another example, `(map (lambda (x) (* x x)) '(2 4 5))`, will take each item in the list and apply the square function to them, and return a list: `'(4 16 25)`. Your task is to fully implement `map`.