Carleton University
Department of Systems and Computer Engineering
SYSC 3101 - Programming Languages - Winter 2018

Lab 4 - Modifying the Calculator Interpreter

References:

- Lecture slides: *SYSC_3101_W18_10_Calc_Interpreter*
- The handout for Lab 2 contains links to *The Racket Guide*, *The Racket Reference*, and *DrRacket: The Racket Programming Environment*.

Racket Coding Conventions

As always, your code should adhere to widely-used coding conventions for Scheme/Racket, as described in the Lab 1 handout. Remember to use DrRacket's Racket > Reindent All command to reformat code in the definitions area.

Getting Started

Download file lab4calc.rkt from cuLearn. This file contains the interpreter for a 4-function calculator language. (This interpreter is identical to the one in calc.rkt, which was presented in a recent lecture.)

Launch DrRacket and open lab4calc.rkt.

Exercise 1

To run the calculator interpreter, click Run, then type `(calc)` in the Interactions area. This will call the procedure that executes the interpreter's read-eval-print loop (REPL). The REPL will display a prompt (`calc:`) and wait for you to type an expression in the input box.

The calculator supports four operations: `+`, `-`, `*` and `/`. Arithmetic expressions are typed using the same syntax as Racket; for example, to calculate 1 + 2 + 3, enter this expression:

```
(+ 1 2 3)
```

The interpreter evaluates the expression, displays the result (6), then prompts you to enter another expression. To exit the interpreter, type a bad expression; e.g., `exit`.

Experiment with the calculator. Which operators require no arguments? What do they do? Try these expressions: `(+)`, `(-)`, `(*)`, `(/)`.

What operations are performed when the operators have exactly one argument? Try these expressions: `(+ 3)`, `(- 3)`, `(* 3)`, `(/ 3)`.

What operations are performed when the operators have multiple arguments? Try some expressions with two arguments. Try some expressions with three or four arguments.

Read procedures `calc-eval` and `calc-apply`. Make sure you understand how `calc-eval` evaluates expressions that have nested expressions; for example `(+ 2 (* 3 4) 5 6)`. Make sure you understand how `calc-apply` handles expressions with 0 arguments, 1 argument and multiple arguments.

Exercise 2

Modify `calc-apply` to provide an `abs` operator. This operator expects exactly one argument, and the expression `(abs x)` calculates the absolute value of `x`. Racket provides a procedure that calculates absolute values (check Section 4.2.2, *Generic Numerics*, in the *Racket Reference*.) The calculator interpreter should call this procedure.

When `abs` expressions with an incorrect number of arguments are entered, the calculator should call `error` to display this message: `"Calc: abs requires exactly 1 arg"`.

Test your modifications. Verify that the calculator correctly evaluates `abs` expressions in which the argument is a simple number; for example `(abs 4)`, as well as expressions in which the argument is a nested arithmetic expression; for example, `(abs (* 3 (- 5 8)))`.

Exercise 3

Modify `calc-apply` to provide an `**` operator. This operator expects exactly two arguments, and the expression `(** a b)` calculates `a` raised to the power of `b`. Racket provides a procedure that calculates powers (check Section 4.2.2, *Generic Numerics*, in the *Racket Reference*.) The calculator interpreter should call this procedure.

When `**` expressions with an incorrect number of arguments are entered, the calculator should call `error` to display this message: `"Calc: ** requires exactly 2 args"`.

Test your modifications. Verify that the calculator correctly evaluates `**` expressions in which the arguments are simple numbers as well as expressions in which one or both arguments are nested arithmetic expressions.

Interlude - Racket's `foldr` procedure

When you did Exercise 1, you saw that `calc-apply` calls `foldr` to perform addition and multiplication with 0 or more arguments, and subtraction and division with two or more arguments. Before starting the next exercise, we need to have a closer look at `foldr`.

`foldr` accepts a procedure *`proc`*, an initial value *`init`*, and a list:

```
(foldr proc init lst)
```

It applies *`proc`* to each element in the list, and combines the return values in a way that is determined by *`proc`*. Procedure *`proc`* must take 2 arguments. The first time *`proc`* is called, its first argument is the last item in the list, and the second argument is *`init`*. In subsequent invocations of *`proc`*, the second argument is the return value from the previous invocation of *`proc`*. The list is traversed from right to left, and the result of the entire `foldr` call is the result of the last invocation of *`proc`*.

Try these expressions in the Interactions area.

```
> (foldr + 0 '(2 3 4))

> (foldr + 1 '(2 3 4))

> (foldr * 0 '(2 3 4))

> (foldr * 1 '(2 3 4))
```

What does each expression calculate? For each expression, what are the arguments each time `foldr` calls `+` or *?

Exercise 4

Racket provides a `max` procedure that accepts one or more integers or real numbers and returns the largest of the numbers. For example,

`(max 1.0 3.0 2.0)` returns `3.0`

and

`(max 1 3 2)` returns `3`.

Modify `calc-apply` to provide a `max` operator. This operator should accept one or more numbers and should call Racket's `max` procedure to calculate the largest one.

When `max` expressions with an incorrect number of arguments are entered, the calculator should call `error` to display this message: `"Calc: max requires 1 or more args"`.

> *Hint:* Suppose you enter this calculator language expression: `(max 1 3 2)`. When `calc-apply` is called, parameter `fn` is bound to `'max` and parameter `args` is bound to the list `'(1 3 2)`. Racket's `max` procedure expects one or more numeric arguments, not a list, so the calculator interpreter can't call Racket's `max` this way:
>
> ```
> ...(max args)...
> ```
>
> How could you use `foldr` to apply Racket's `max` to all the numbers in the list?

Test your modifications. Verify that the calculator correctly evaluates `max` expressions in which the arguments are simple numbers as well as expressions in which the arguments are nested arithmetic expressions; for example, `(max (+ 2 1) (* 3 2) (- 4 2))`

Also, verify that the calculator can evaluate expressions such as this one:

```
(+ 3 (abs -4) (** 2 3) (max 4 6 5) (abs (* -2 3)))
```