Carleton University
Department of Systems and Computer Engineering
SYSC 3101 – Programming Languages - Winter 2020

Lab 3 - `lambda` Expressions and Higher-Order Procedures

References

Two documents at the Racket website provide plenty of information about the Racket dialect of Scheme:

The Racket Guide, https://docs.racket-lang.org/guide/index.html

The Racket Reference, https://docs.racket-lang.org/reference/index.html

A guide to the DrRacket IDE can be found here:

http://docs.racket-lang.org/drracket/index.html

Racket Coding Conventions

Please adhere to the conventions described in the Lab 1 handout.

Getting Started

Launch the DrRacket IDE.

If necessary, configure DrRacket so that the programming language is Racket. To do this, select `Language > Choose Language` from the menu bar, then select `The Racket Language` in the `Choose Language` dialog box.

`#lang racket` should appear at the top of the definitions area. Don't delete this line.

"The Rules"

Do not use special forms that have not been presented in lectures. Specifically,

- Do not use `set!` to perform assignment; i.e., rebind a name to a new value.
- Do not use any of the Racket procedures that support *mutable* pairs and lists (`mpair, mcons, mcar, mcdr, set-mcar!, set-mcdr!`), as described in Section 4.10 of *The Racket Reference*.
- Do not use `begin` expressions to group expressions that are to be evaluated in sequence.

Exercise 0

As you read this exercise, type the expressions in DrRacket's interactions area.

Here's a `lambda` expression that has two formal parameters, `a` and `b`. When we type this expression in the interactions area, we see that it evaluates to a procedure object:

```
>(lambda (a b)(+ a b))
#<procedure>
```

We predict that the procedure will return the sum of its two arguments. We can design some simple experiments to confirm this: we type combinations that call the procedure created by the `lambda` expression:

```
> ((lambda (a  b)  (+  a  b))  1  2)
> 3

> ((lambda (a b) (+ a b)) -5 5)
0
```

We can use `define` to give the procedure a name:

```
>(define add (lambda (a b) (+ a b)))

> add
#<procedure:add>
```

After defining `add`, we can call it:

```
>(add 1 2)
3
```

Exercise 1

For this exercise, we recommend that you type your predictions, experiments, observations and conclusions in a file, so that you have a record of your lab work when you study for the exams.

Without using DrRacket, determine which of the following expressions are valid `lambda` expressions; that is, predict which of these expressions evaluate to procedure objects. Next, check whether your answers are correct by typing the expressions in DrRacket's interactions area.

```
(lambda (x y z) (x y z))
(lambda () 10)
(lambda (x) x)
(lambda (x y) x)
```

Without using DrRacket, for each of the valid expressions, predict what the procedure created by `lambda` expression does. Check whether your predictions are correct by designing some combinations that call the procedures (see the examples in Exercise 0). Type the combinations in DrRacket's interactions area.

Exercise 2

For this exercise, we recommend that you type your predictions, experiments, observations and conclusions in a file, so that you have a record of your lab work when you study for the exams.

Without using DrRacket, evaluate these expressions.

Part (a)

```
((lambda (x y) (+ x (* x y))) 1 2)
```

Part (b)

```
((lambda (x y)
(+ x ((lambda (z)
         (+ (* 3 z) (/ 1 z)))
         (* y y))))
1 2)
```

Next, use DrRacket to check your answers.

Exercise 3

For this exercise, we recommend that you type your predictions, experiments, observations and conclusions in a file, so that you have a record of your lab work when you study for the exams.

Review Exercise 0.

Without using DrRacket, predict what DrRacket would display when it evaluates these expressions:

Part (a)

```
>(define (square x) (* x x))
>square
>(square 5)

>(define sq (lambda (x) (* x x)))
>sq
>(sq 5)
```

Part (b)

```
;; make-adder is a procedure that returns a procedure
>(define (make-adder num) (lambda (x) (+ x num))
>make-adder
>(make-adder 3)
>((make-adder 3) 7)
```

Part (c)

```
>(define plus3 (make-adder 3))
>plus3
>(plus3 7)
```

Use DrRacket to check your predictions.

Exercise 4

Racket provides a procedure, `(build-list n f)`. Parameter n is a natural number, and parameter `f` is a procedure that takes one argument, which is a natural number. `build-list` constructs a list by applying `f` to the numbers between `0` and `n-1`, inclusive.

In other words, `(build-list n f)` produces the same result as:

```
(list (f 0) (f 1) .. (f (- n 1)))
```

For example, given:

```
(define (increment x) (+ x 1))
```

the expression

```
(build-list 5 increment)
```

produces this list:

```
(1 2 3 4 5)
```

Of course, the procedure passed to `build-list` can be a `lambda` expression:

```
(build-list 5 (lambda (x) (+ x 1)))
```

In a file named `lab3.rkt`, define these three procedures. Each procedure must call `build-list`. The procedure passed to `build-list` must be a `lambda` expression, not a named procedure:

- `build-naturals` returns the `list (list 0 .. (- n 1))` for any natural number `n`. Example: `(build-naturals 5)` returns `(0 1 2 3 4).`
- `build-rationals` returns the list (list $1 \frac{1}{2} \ldots \frac{1}{n}$) for any natural number `n`. Example, (build-rationals 5) returns ($1 \ \frac{1}{2} \ \frac{1}{3} \frac{1}{4} \frac{1}{5}$)
- `build-evens` returns the list of the first `n` even natural numbers (note: 0 is an even number). Example: `(build-evens 5)` returns `(0 2 4 6 8).`

Exercise 5

In file `lab3.rkt`, define a procedure named `cubic` that takes three numeric arguments, `a, b` and `c`:

```
(cubic a b c)
```

`and` **return another procedure.** This procedure takes a numeric argument, `x`, and evaluates the cubic x^3+ax^2+bx+c at `x`. Use `lambda` expression to define a procedure returned by `cubic.`

For example, `((cubic 1 2 3) 4)` calculates $4^3 + 1 \times 4^2 + 2 \times 4 + 3$, which is 91.

Exercise 6

In file `lab3.rkt,` define a procedure named `twice` that takes a procedure of one argument and and returns a procedure that applies the original procedure twice. For example, if `square` is a procedure that squares its argument, then `(twice square)` returns a procedure that raises its argument to the power 4. If `inc` is a procedure that adds 1 to its argument, then `(twice inc)` returns a procedure that adds 2 to its argument.

Use a `lambda` expression to define the procedure returned by `twice.`

Check your `twice` procedure using these tests:

```
(define (square x) (* x x))
((twice square) 5)
625  ; (5²)²

(define (inc x) (+ x 1))
((twice inc) 5)
7    ; (5 + 1) + 1
```