

Carleton University
Department of Systems and Computer Engineering
SYSC 3101 - Programming Languages - Winter 2023

Lab 4 - Local State Variables

(20 Marks)

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

1. Download file `lab4.rkt` from Brightspace.
2. Launch DrRacket and open `lab4.rkt`.

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.

Exercise 1

In lectures, we explored different ways to model counters.

File `lab4.rkt` contains procedure `make-upcounter`:

```
(define (make-upcounter counter)
  (lambda ()
    (set! counter (+ counter 1))
    counter))
```

`make-upcounter` is called to create independent counter objects. Each object has its own local state variable named `counter`:

Step 1: Click DrRacket's Run button.

Type these expressions in the interactions area. What happens when the expressions are evaluated?

```
> (define counter1 (make-upcounter 0))
> (counter1)
> (counter1)
```

What happens when these expressions are evaluated?

```
> ((make-upcounter 0))
> ((make-upcounter 0))
```

Do they do the same thing as the previous example? Make sure you can explain any differences.

Step 2: We should verify that counters are independent objects. Create another counter object:

```
> (define counter2 (make-upcounter 10))
```

Now type these expressions in the interactions area:

```
> (counter1)
> (counter2)
> (counter1)
> (counter2)
```

Why can you conclude that count-up operations on `counter1` don't affect `counter2`, and vice-versa?

Exercise 2

File `lab4.rkt` contains procedure `make-counter`. This procedure provides a way to model counters that can perform more than one operation; for example, counting up and down.

```
(define (make-counter counter)

  (define (count-up)
    (set! counter (+ counter 1))
    counter)

  (define (count-down)
    (if (> counter 0)
        (begin (set! counter (- counter 1))
                counter)
        "Counter is 0"))

  (define (dispatch cmd)
    (cond ((eq? cmd 'inc) count-up)
          ((eq? cmd 'dec) count-down)
          (else (error "Unknown command:" cmd))))

  dispatch)
```

Each time `make-counter` is called, it returns a `dispatch` procedure that represents a counter object. Each object has its own local state variable named `counter`.

Type these expressions in the interactions area:

```
> (define counter3 (make-counter 0))
> counter3
```

What is bound to counter3?

When the `dispatch` procedure is called with the "command" `'inc` as an argument, it returns the count-up procedure. When it is given the `'dec` message, `dispatch` returns the count-down procedure. Verify this by typing these expressions in the interactions area:

```
> (counter3 'inc)
> (counter3 'dec)
> (counter3 'reset)
```

Type the following expressions. What do they do?

```
> ((counter3 'inc))
> ((counter3 'inc))
> ((counter3 'dec))
> ((counter3 'dec))
> ((counter3 'dec))
> ((counter3 'reset))
```

Exercise 3

In the `make-counter` procedure used in Exercise 2, the local state variable `counter` is a formal parameter of `make-counter`. We could also create the local state variable explicitly, using `let`, as shown here:

```
(define (make-counter-with-let initial-count)

  (let ((counter initial-count))

    (define (count-up)
      (set! counter (+ counter 1))
      counter)

    (define (count-down)
      (if (> counter 0)
          (begin (set! counter (- counter 1))
                  counter)
          "Counter is 0"))

    (define (dispatch cmd)
      (cond ((eq? cmd 'inc) count-up)
            ((eq? cmd 'dec) count-down)
            (else (error "Unknown command:" cmd))))

    dispatch))
```

Notice the changes:

- the body of `make-counter-with-let` is a `let` expression;
- `counter` is now a local variable, and is defined in the `let` statement;
- the procedure's parameter name has been changed to `initial-count`, and this parameter is used to initialize `counter`.

Experiment with `make-counter-with-let`. (This procedure is defined in `lab4.rkt`.) Verify that the counter objects returned by this procedure respond to the same commands and return the same values as the objects returned by `make-counter`.

Exercise 4 (5 Marks)

In `lab4.rkt`, make a copy of procedure `make-counter-with-let` and rename the copy `make-counter-ex4`. Replace the `dispatch` procedure with a `lambda` expression that does the same thing as `dispatch`.

`make-counter-ex4` will return the procedure created when the `lambda` expression is evaluated.

Test `make-counter-ex4` by typing expressions similar to those in Exercise 2.

Exercise 5(5 Marks)

In `lab4.rkt`, make a copy of procedure `make-counter-ex4` and rename the copy `make-counter-ex5`. Modify this procedure so that counter objects recognize two additional messages:

- `'get` returns the current value of the counter, but doesn't change its state;
- `'reset` resets the counter to 0.

Exercise 6(5 Marks)

In `lab4.rkt`, make a copy of procedure `make-counter-ex5` and rename the copy `make-counter-ex6`. Modify this procedure so that it is passed two arguments: the initial counter value and the amount by which the counter is incremented each time it is given the `'inc` command.

For example, here's how we create a counter object with value 0, which increments with a step-size of 5.

```
> (define counter6 (make-counter-ex6 0 5))

> ((counter6 'get))
0
> ((counter6 'inc))
5
> ((counter6 'inc))
10
> ((counter6 'dec))
9
```

```

> ((counter6 'inc))
14
> ((counter6 'reset))
0

```

Exercise 7(5 Marks)

In lab4.rkt, make a copy of procedure make-counter-ex6 and rename the copy make-counter-ex7. Modify this procedure so that each counter maintains a "high water mark", which is the maximum value the counter reaches as it is incremented. The counter will return this value when it is given the 'max command. The high water mark is set to 0 when the counter is given the 'reset command.

For example,

```

> (define counter7 (make-counter-ex7 0 2)) ;; count is 0
                                           ;; increment amount is 2
                                           ;; high water mark is 0

> ((counter7 'inc))
2
> ((counter7 'inc))
4
> ((counter7 'max))
4
> ((counter7 'inc))
6
> ((counter7 'max))
6
> ((counter7 'dec))
5
> ((counter7 'dec))
4
> ((counter7 'max)) ;; after decrementing the counter twice,
6                  ;; the high water mark is still 6
> ((counter7 'reset))
0
> ((counter7 'max))
0

```

Remember to demonstrate your work to TA to get the lab grade then submit your lab4.rkt. file on Brightspace.

If you don't present your work to any TA, a grade of 0 is assigned. The submissions on the bright space will not be graded. It is your responsibility to ensure that your material has been submitted at the end of the laboratory session. If no material is received, a grade of 0 is assigned.