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

Lab 3 - Local State Variables

References

- Lecture slides: *SYSC_3101_W18_8_Assignment_and_Local_State*
- *Structure and Interpretation of Computer Programs*, Section 3.1, *Assignment and Local State*, up to the end of Section 3.1.1, *Local State Variables*
- The handout for Lab 2 contains links to *The Racket Guide*, *The Racket Reference*, and *DrRacket: The Racket Programming Environment*.

Racket Coding Conventions

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

Getting Started

1. Download file lab3.rkt from cuLearn. This file contains the procedures you'll use in Exercises 1, 3 and 5.
2. Launch DrRacket and open lab3.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. *SICP* Section 3.1.1 presents different ways to model transactions on bank accounts. One example is `make-withdraw`:

```
(define (make-withdraw balance)
  (lambda (amount)
    (if (>= balance amount)
        (begin (set! balance (- balance amount))
               balance)
        "Insufficient funds")))
```

`make-withdraw` is used to create independent objects ("withdrawal processors"). Each object has its own local state variable `balance`:

```
> (define W1 (make-withdraw 100))
> (define W2 (make-withdraw 100))

; Withdrawals from W1 don't affect W2, and vice-versa
> (W1 50)
50
```

```
> (W2 70)
30
> (W2 40)
"Insufficient funds"
> (W1 40)
10
```

Experiment with `make-withdraw`. What happens when these expressions are evaluated?

```
> (define W1 (make-withdraw 100))
> (W1 50)
> (W1 20)
```

What happens when these expressions are evaluated?

```
> ((make-withdraw 100) 50)
> ((make-withdraw 100) 20)
```

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

Exercise 2

(SICP Exercise 3.1)

An *accumulator* is a procedure that is called repeatedly with a single numeric argument and accumulates its arguments into a sum. Each time it is called, it returns the currently accumulated sum.

Applying what you learned from Exercise 1, write a procedure `make-accumulator` that generates accumulators, each maintaining an independent sum. The input to `make-accumulator` should specify the initial value of the sum; for example,

```
> (define A (make-accumulator 5))
> (A 10)
15
> (A 10)
25
```

Exercise 3

SICP Section 3.1.1 presents procedure `make-account`, which provides a way to model simple bank accounts that can process deposits and withdrawals. Each time this procedure is called, it returns a `dispatch` procedure that represents a bank account object:

```
(define (make-account balance)
  (define (withdraw amount)
    (if (>= balance amount)
        (begin (set! balance (- balance amount))
               balance)
        "Insufficient funds"))

  (define (deposit amount)
    (set! balance (+ balance amount))
    balance)

  (define (dispatch msg)
    (cond ((eq? msg 'withdraw) withdraw)
          ((eq? msg 'deposit) deposit)
          (else (error "Unknown request -- make-account" msg))))

  dispatch)
```

When the `dispatch` procedure is called with the "message" `'deposit` as an argument, it returns the `deposit` procedure. When it is given the `'withdraw` message, `dispatch` returns the `withdraw` procedure. The `deposit` or `withdraw` procedure is then applied to a specified amount.

This example shows how `make-account` can be used. Notice that `my-account` and `your-account` are bound to completely separate account objects, each with its own local state variable `balance`.

```
> (define my-account (make-account 100))
> (define your-account (make-account 100))
> ((my-account 'withdraw) 50)
50
> ((my-account 'withdraw) 60)
"Insufficient funds"
> ((my-account 'deposit) 40)
90
> ((my-account 'withdraw) 60)
30
> ((your-account 'withdraw) 10)
90
```

Experiment with `make-account`. Try the examples shown in the previous paragraph.

What happens when the following expressions are evaluated?

```
> (define my-account (make-account 100))
> (define my-withdraw-processor (my-account 'withdraw))
```

```
      > (define my-deposit-processor (my-account 'deposit))
      > (my-withdraw-processor 50)
      > (my-withdraw-processor 60)
      > (my-deposit-processor 40)
      > (my-withdraw-processor 60)
```

Is there any advantage to requesting account transactions this way, as opposed to the approach shown earlier in this exercise?

Exercise 4

(SICP Exercise 3.3)

Make a copy of `make-account` and rename it as `make-password-account`. Modify this procedure so that it that creates password-protected accounts. That is, `make-password-account` should take a symbol as an additional argument, as in

```
> (define acc (make-password-account 100 'secret-password))
```

The resulting account object should process a request only if it is accompanied by the password with which the account was created, and should otherwise return a complaint:

```
> ((acc 'secret-password 'withdraw) 40)
60

> ((acc 'some-other-password 'deposit) 50)
"Incorrect password"
```

Note: ensure that no run-time error occurs when `"Incorrect password"` is displayed. Hint: as demonstrated by the examples in the previous paragraph, `make-password-account` should return a `dispatch` procedure, using the same technique as `make-account`. When the `dispatch` procedure is called it returns a procedure. That procedure is always called with a single argument (an amount of money). In other words, the expression:

```
 > (acc 'some-other-password 'deposit)
```

calls the `dispatch` procedure bound to `acc` and returns a procedure, which means that the expression:

```
> ((acc 'some-other-password 'deposit) 50)
"Incorrect password"
```

calls the procedure returned by `dispatch`, passing it `50`.

Exercise 5

In the make-account procedure in *SICP* Section 3.1.1 (used in Exercise 3), the local state variable balance is a formal parameter of make-account. We could also create the local state variable explicitly, using let, as shown here:

```
(define (make-account-with-let initial-balance)
  (let ((balance initial-balance))

    (define (withdraw amount)
      (if (>= balance amount)
          (begin (set! balance (- balance amount))
                 balance)
           "Insufficient funds"))

    (define (deposit amount)
      (set! balance (+ balance amount))
       balance)

    (define (dispatch msg)
      (cond ((eq? msg 'withdraw) withdraw)
            ((eq? msg 'deposit) deposit)
            (else (error "Unknown request -- make-account" msg))))

    dispatch))
```

Notice the changes:

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

Experiment with make-account-with-let. Verify that the bank account objects returned by this procedure respond to the same messages and return the same values as the objects returned by make-account.

Exercise 6

(SICP Exercise 3.4)

Make a copy of your make-password-account procedure from Exercise 4 and rename it as make-password-account-monitored.

Applying what you learned from Exercises 4 and 5, modify this procedure. Use a let statement to add another local state variable so that, if an account is accessed more than seven consecutive times with an incorrect password, it invokes the procedure call-the-cops. Note: balance and the password can remain formal parameters - they don't have to be defined in the let.