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

Assignment 1

Procedures, Recursion, and Patterns of Computational Processes

Posted: Wednesday, February 7, 2018

Due: Friday, February 16, 2018. (Solutions submitted after Friday, but no later than 11:55 pm on Sunday, February 18, will be considered to have been submitted on time.)

Instructions

- Exercise 1 - 7 can be completed using material covered up to the end of Week 3, Lecture 1. See *Procedures and Abstractions, Parts 1, 2 and 3*, in the Lecture Materials section on cuLearn. Exercise 8 uses some of the procedures developed in *Procedures and Abstractions, Part 4, Higher-Order Procedures.*

- 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.

- Follow Racket/Scheme coding conventions for formatting code (that is, indentation and placement of `()`'s) and selecting names. For more information, see the handout for Lab 1.

  In addition to being readable, your procedures should be concise. Feel free to define "helper" procedures that are called by your procedures.

- Don't use `set!` or any other procedure that rebinds variables.
- Don't use cons-pairs or lists; they're not needed.
- Do not use `begin` expressions to group expressions that are to be evaluated in sequence.

- You can use `lambda` expressions to create procedures and `let` expressions to create local variables, but none of the exercises require you to use these special forms.

- Submit one file for each exercise, containing all of your procedures for the exercise along with any other information that is required for a complete solution. For example, some exercises require you to show the substitution model you used to determine the process generated by the procedure.

  Make sure you use the file names and procedure names specified in each exercise. We will be using unit tests to automate the testing of your code, but these won't run if you pick different names for your files and procedures.

Exercise 1. In file ex1.rkt, define a procedure named `sum-largest-squares` that takes three numbers as arguments and returns the sum of the squares of the largest two numbers. Your solution must use of the `square` and `sum-of-squares` procedures that were presented in class.

Exercise 2. The operator in a Racket/Scheme combination can be a compound expression, as shown by procedure `a-b`:

```
(define (a-b a b)
  ((cond [(> b 0) +]
     [(= b 0) -]
     [else *]) a b))
```

What does procedure `a-b` do for all integers `a` and `b`? Justify your answer by using the substitution model of execution to illustrate the process generated by the procedure for different values of `a` and `b`. Save your solution in file ex2.rkt.

Exercise 3. In file ex3.rkt, define a recursive procedure named `sum-odd-digits` that computes the sum of the odd digits of a positive integer. For example, `(sum-odd-digits 1285)` computes 6.

Exercise 4. Each container in a warehouse is marked with a numeric code to indicate the quantity of each type of item stored in the container. The code consists of an unlimited number of triples of integer digits. In each triple, the last two digits specify the type of an item and the first digit specifies the quantity of that item. For example, the code 812493 indicates that there are 8 type-12 items and 4 type-93 items. The code 915815715 indicates that there are 24 type-15 items (9 plus 8 plus 7).

In file ex4.rkt, define a recursive procedure `(count-of-items code type)` that returns the quantity of items of the specified `type` in the container, as recorded in `code`. Return 0 if there are no items of that type.

Exercise 5. The greatest common divisor (GCD) of two integers *a* and *b* is defined to be the largest integer that divides both *a* and *b* with no remainder. For example, the GCD of 16 and 28 is 4.

Euclid's Algorithm is a famous technique for efficiently calculates the GCD of two positive integers. The basis of this algorithm is that observation that, if *r* is the remainder when *a* is divided by *b*, then the common divisors of *a* and *b* are precisely the same as the common divisors of *b* and *r*. Thus, we can use the equation

$$GCD(a,\ b)\ =\ GCD(b,\ r)$$

to successively reduce the problem of computing a GCD to the problem of computing the GCD of smaller and smaller pairs of integers. For example,

$GCD(206, 40) = GCD(40, 6) = GCD(6, 4) = GCD(4, 2) = GCD(2, 0) = 2$

reduces $GCD(206,40)$ to $GCD(2,0)$, which is 2.

Starting with any pair of positive integers, performing repeated reductions will eventually produce a pair where the second number is 0. The GCD is the other number in the pair.

Euclid's Algorithm can easily be defined as a procedure:

```
(define (gcd a b)
  (if (= b 0)
      a
      (gcd b (remainder a b))))
```

Is the process generated by `(gcd 279 15)` is iterative or recursive? Justify your answer by using the substitution model to illustrate the process generated by the procedure. Save your solution in file ex5.rkt.

Exercise 6. Consider these procedures:

```
(define (increment a) (+ a 1))

(define (decrement a) (- a 1))

(define (add-ab a b)
  (if (= a 0)
      b
      (increment (add-ab (decrement a) b))))

(define (sum-ab a b)
  (if (= a 0)
      b
      (sum-ab (decrement a) (increment b))))
```

Both procedures use recursive procedure calls to perform the addition (+ a b).

For each of parts (a) and (b), justify your answer by using the substitution model to illustrate the process generated by the procedure. Save your solution in file ex6.rkt.

(a) Is the process generated by `(add-ab 3 4)` iterative or recursive?
(b) Is the process generated by `(sum-ab 3 4)` iterative or recursive?

Exercise 7. A function *f* is defined as:

$$f(n) = n \text{ if } n < 4$$

and

$$f(n) = 2f(n-1) + 3f(n-3) + 4f(n-5) \text{ if } n \geq 4.$$

(a) Define a recursive procedure `f-recursive` that computes *f* by means of a recursive process.

(b) Define a recursive procedure `f-iterative` that computes *f* by means of an iterative process.

Save your procedures in file ex7.rkt.

Exercise 8. This higher-order summing procedure was developed in class:

```
(define (sum fn a next b)
  (if (> a b)
      0
      (+ (fn a)
         (sum fn (next a) next b))))
```

Given helper procedures `identity` and `increment`, we can define a procedure that calls `sum` to add all the integers between `a` and `b`, inclusive:

```
(define (identity x) x)

(define (increment n) (+ n 1))

(define (sum-integers a b)
  (sum identity a increment b))
```

Procedure `sum` generates a recursive process. In file ex8.rkt, reimplement `sum` so that it generates a iterative process. Verify that the behaviour of `sum-integers` doesn't change when it calls the new implementation of `sum`.