CSE 1729 - Introduction to Principles of Programming November 13, 2018

Laboratory Assignment 11

## Objectives

- Work with mutations
- Work with objects

## Activities

Editorial comment: the code you write in questions 1 will undoubtedly be less esthetically pleasing than the code you could write without destructive modification. Not that I am biased. -r.

1. Early in the course, we saw how to compute the factorial function using nondestructive functional programming. We've seen that a good way to do this is to define helper function which passes itself an accumulator at each recursive call. Using destructive assignment, we can use a helper function with no arguments that can update an accumulator in place rather than passing it as an argument to itself. This helper function doesn't even need to evaluate to anything useful, as the result ends up in the in-place accumulator.

   What does this look like? Here is an example that takes this approach to sum the first $n$ integers:

```
(define (sum-first-n n)
  (let ((sum 0)
        (count 0))
    (define (helper)
      (cond ((= count n) 'done)
            (else
             (set! count (+ count 1))
             (set! sum (+ sum count))
             (helper))))
    (helper)
    sum))
```

   Notably, when `helper` is finished, the result is in the variable `sum`.

   (a) Write a new version of the factorial procedure `(fact n)` that works this way (that is, using a `helper` with no parameters that modifies an accumulated value). Make sure it works as intended by testing it on the numbers 1 through 5.

   (b) We've also looked at procedures to compute "hailstone sequences." Recall that a hailstone sequence is a sequence of numbers $a_1, a_2, \cdots$ such that

$$a_i = \begin{cases} \frac{a_{i-1}}{2} & \text{if } a_{i-1} \text{ is even} \\ 3a_{i-1} + 1 & \text{otherwise} \end{cases}$$

Write a procedure (hailstone n) that computes a list all the numbers in the hailstone sequence starting from n using the approach above (that is, by using a `helper` function with no arguments) and two variables – the value `hailstone` was called with that `helper` changes, and the variable that will contain the answer list of numbers. Your `helper` should add the current number to the list each time it is called. Note that we want the list in order from n to 1, but it might be easier to build that list in reverse order – just use the `reverse` function before you return it.

2. Extend the bank account example in the slides with the following upgrade and new methods:

   - `withdraw` – modify this method so it prints a message showing what the balance is and how much less than the request it is if the request is larger than the balance.
   - `accrue` – add 1 year of simple interest to the balance. At first assume an interest rate of 1%
   - `setrate` – change the interest rate to the given argument, where 1% is represented as 0.01, not 1.

   You may start with the following code which is (essentially) copied from the lecture slides:

```
(define (new-account initial-balance)
  (let ((balance initial-balance))
    (define (deposit f)
       (set! balance (+ balance f))
        balance)
    (define (withdraw f)
      (cond ((> f balance)  "Insufficient funds")
            (else
               (set! balance (- balance f))
               balance)))
    (define (bal-inq) balance)
    (lambda (method)
      (cond ((eq? method 'deposit) deposit)
            ((eq? method 'withdraw) withdraw)
            ((eq? method 'balance-inquire) bal-inq)))))
```

3. Create two bank account objects and demonstrate that the balance and rate can be set and modified independently.

4. Show how to implement Stacks as objects. You can base your implementation on a list, and you should provide the following methods:

   - `is-empty?` – true if the stack is empty.
   - push – adds an element to the top of the stack
   - `top` – returns the top element of the stack without changing the stack
   - pop – returns the top element of the stack, which is removed from the stack

Use the "standard" form for defining an object: when you make an object, it returns a function that can be used to access or modify parts of the object. See, for example, the bank account code given above – `(define my-acct (new-account 100)` sets `my-acct` to be a bank account with balance of 100, and `((my-acct 'deposit) 10)` would add 10 to its balance. Use the code below as a starting point for Stack.

```
(define (make-stack)
  (let ((state-variable initial-value))
      (define (is-empty?)
         ...)
      (define (push thing-to-push)
         ...)
      (define (top)
         ...)
      (define (pop)
         ...)
      (lambda (meth-name)
         (cond ((eq? meth-name 'is-empty)
                 is-empty?)
                ((eq? meth-name 'push)
                 push)
                ((eq? meth-name 'top)
                 top)
                ((eq? meth-name 'pop)
                 pop)))))
```

5. You have seen `set!`: there are other destructive operators that work on pairs:

   - `(set-car! lst val)` changes `lst` so that `(car lst)` refers to `val`
   - `(set-cdr! lst val)` changes `lst` so that `(cdr lst)` refers to `val`

   The interesting thing about these is that if you use them to change a list, the list changes, and any variable referring to that list changes. Consider the following:

```
>(define a '(1 2 3))
>a
(1 2 3)
>(set-car! a 'fish)
>a
(fish 2 3)
>(set-cdr! (cdr a) '(eat))
>a
(fish 2 eat)
```

   The `set-cdr!` in the example illustrates that changing a part of a list destructively changes the whole list. Let's try it and see what happens.

(a) Write a Scheme function called `nconc!` that does append destructively.

Remember how append works:

```
(define (append x y)
   (if (null? x)
       y
       (cons (car x)(append (cdr x) y))))
```

On each recursive call for which `x` is not empty, we build a new list using `cons`.

Here is an alternative:

1. Recursively cdr down list x until you reach the last element
2. Set the cdr of that last element to be y

Write the scheme function `(nconc!  x y)` that implements the above alternative. It should return a list that looks just like what you would get from append, which is perhaps the trickiest thing about the function.

(b) Demonstrate that append and `nconc!` behave differently. Try the following:

```
(define a '(1 2 3))
(define b '(4 5 6))
(append a b)
a
b
(nconc! a b)
a
b
```

How do you explain the differences? What do you think would happen if you evaluated `(nconc!  a a)`? (Hint: try drawing some box-and-pointer diagrams.)