CSE 1729 - Introduction to Principles of Programming November 17, 2020

Laboratory Assignment 9

Objectives

- Work with mutations
- Work with objects

Activities

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`.

   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.

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 (i.e. "Withdrawal not allowed since balance is x" where x is the current balance of the account). 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:

```scheme
(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.