211hw4_S11

Homework 4 (Due Monday 2/14/2011 at 10:00am)

Use the `Intermediate Student with lambda` language level. You can choose to use `lambda`, as appropriate, in any of the assigned problems. [For 2011: You may *not* use `local` to define functions; all program functions (including helpers) must be defined at the top level.]

Book Problems:

- 18.2.2 (10 pts)
  - In this problem, simply annotate each definition by attaching identifying subscripts (written in line such as x1 for "x sub 1" and y4 for "y sub 4") to _all_ variable occurrences so that **all** of the uses of **each** defining occurrence are identified. A good way to choose a subscript value for a variable is to use the lexical nesting level of the binding occurrence. For example,

```
(define (my-max lon)
  (cond [(empty? lon) (error 'my-max "applied to no arguments")]
        [(empty? (rest lon)) (first lon)]
        [else
         (local [(define head (first lon))
                 (define max-tail (my-max (rest lon)))]
           (if (>= head max-tail) head max-tail))]))
```

    becomes

```
(define (my-max1 lon1)
  (cond [(empty? lon1) (error 'my-max "applied to no arguments")]
        [(empty? (rest lon1)) (first lon1)]
        [else
         (local [(define head2 (first lon1))
                 (define max-tail2 (my-max1 (rest lon1)))]
           (if (>= head2 max-tail2) head2 max-tail2))]))
```

- 20.1.1 (10 pts)
- 21.1.2 (10 pts)
- 21.2.1 (10 pts)
- 21.2.3 (10 pts)
  - "Lists of names" just means "lists of symbols."
- 22.2.2 (10 pts) [For 2011: 12 pts]
  - Name your function `make-sort` .
  - `make-sort` produces functions. Test these results by applying them to a few different arguments.
- 23.3.9 (10 pts) [For 2011: 8 pts]
  - The series represented by `greg` is (4, -4/3, 4/5, -4/7, ...).
- 24.0.8 (5 pts)
  - Instead of drawing arrows from the underlined occurrences, simply annotate each expression by attaching subscripts (written in line such as x1 for "x sub 1" and y4 for "y sub 4") to **all** variable occurrences so that **all** of the uses of each defining occurrence are identified. For example,

```
(lambda (f g x)
  (f x (lambda (x) (g (g x)))))
```

    becomes

```
(lambda (f1 g1 x1)
  (f1 x1 (lambda (x2) (g1 (g1 x2)))))
```

- Problems not in the book:*
- (5 pts) Write the most general (parametric) contract for the following function:

```
; compose : ?
; Purpose: (compose f g) returns the result of composing functions f and g: x |-> f(g(x))
  (define (compose f g)
    (lambda (x) (f (g x))))
```

- (20 pts) Write the same `mergesort` function as described in exercise 26.1.2, except decompose the problem "top-down" rather than "bottom-up". You will need to define a function `split: (list-of number) -> 2-element-structure-of-lists-of-number)` that partitions its input into two lists of approximately (+/- 1) the same length in O(n) time where n is the length of the input list. `(split l)` returns a 2-element structure containing two lists of numbers. See below for hints in defining a 2-element structure. After splitting the list in half, `mergesort` recursively sorts each half and then merges them together. *Be careful about attempting to split a one-element list!*

Hints for writing the "split" function:

A 2-element structure can be defined (you MUST do this somewhere!) in one of two ways:

a) Define your own structure. The CS term for a pair of elements is "dyad", so for instance, one could define a structure as

```
(define-struct dyad (first second))
```

Review your notes on how "define-struct" automatically creates constructor and accessor functions. You will still need to, in words, define what types the "first" and "second" elements are supposed to be (note: there are several ways to do this).

b) Use Scheme's built-in functions. First, you must explicitly define that you are going to use a list of exactly two elements, i.e. `(list a b)`. Scheme already provides functions to access the first and second elements of such a list, called, oddly enough, "first" and "second":

```
(first (list a b)) ==> a

(second (list a b)) ==> b
```

Again, you will still need to define exactly what types the first and second elements are supposed to be. **DO NOT USE `(first (rest lon))`** as this an encapsulation violation of an indefinite-length list!

Think simple!
Follow these important rules whenever you write function on a list using *structural recursion* (note that the generative recursion that mergesort uses follows slightly different rules):

- Write down and follow the list function template!
- Think about what can be done with what your template says you have to work with, namely, `(first aList)` and the recursive result, `(myFunction (rest aList))`.
- Never think in terms of more than one element of the list at a time (this is equivalent to the previous statement). This means do NOT think in terms of finding the length of the list or finding the middle of the list or anything of that nature! Think "first" and "rest", period.
- Use a `local` definition to keep you from repeating the recursive call.