CSE1729 - Introduction to Programming — March 7, 2017

Problem Set 5 Solutions

1. Let f and g be two functions taking numbers to numbers. We define $\mathbf{m}_{fg}$ to be the function so that

$$\mathbf{m}_{fg}(x) = \text{the larger of } f(x) \text{ and } g(x).$$

For example, if $f(x) = x$ and $g(x) = -x$ then $\mathbf{m}_{fg}$ is the absolute value function. Define a function `max` which takes a *pair* of functions, `f` and `g`, as an argument and returns the function $\mathbf{m}_{fg}$ as its value. (So the return value is a *function*: the function which, at every point x, returns the larger value of $f(x)$ and $g(x)$.)

```
(define (max f g)
  (define (int-max x y)
    (if (> x y) x y))
  (lambda (x) (int-max (f x) (g x))))
```

2. Define a SCHEME function, `zip`, which takes as arguments two lists $(a_1 \ldots a_n)$ and $(b_1 \ldots b_n)$ of the same length, and produces the list of pairs $((a_1.b_1)\ldots(a_n.b_n))$.

```
(define (zip lista listb)
  (if (or (null? lista)
          (null? listb))
      '()
      (cons (cons (car lista)
                  (car listb))
            (zip (cdr lista)
                 (cdr listb)))))
```

3. Define a SCHEME function, `unzip`, which takes a list of pairs $((a_1.b_1)\ldots(a_n.b_n))$ and returns a *pair* consisting of the two lists $(a_1 \ldots a_n)$ and $(b_1 \ldots b_n)$.

(Note that these functions are not exactly inverses of each other, since `zip` takes its lists as a two arguments, while `unzip` produces a pair of lists.)

```
(define (unzip plist)
  (if (null? plist)
      '(() ())
      (let* ((rest (unzip (cdr plist)))
             (first (car rest))
             (second (cdr rest)))
        (cons (cons (caar plist) first)
              (cons (cdar plist) second)))))
```

4. Consider the problem of *making change*. Given a list of “denominations” $(d_1 d_2 \ldots d_k)$ and a positive integer n, the problem is to determine the number of ways that n can be written as a sum of the d_i. For example, if we consider the denominations used for US coins, (1 5 10 25), the number 11 can be written in 4 ways:

$$10+1,\ 5+5+1,\ 5+\underbrace{1+\cdots+1}_{6}, \quad \text{and} \quad \underbrace{1+\cdots+1}_{11}.$$

Note that we do not consider 10+1 and 1+10 as “different” ways to write 11: all that matters is the number of occurrences of each denomination, not their order.

Write a SCHEME function `(change k ℓ)` that returns the number of ways to express `k` as a sum of the denominations appearing in the list ℓ.

(Hint: There is a nice recursive decomposition of this problem: Let $C(n,\ell)$ denote the number of ways to express n as a sum of elements from the list $\ell = (\ell_1\ell_2\ldots\ell_k)$. We can mentally divide the different ways to do this into two types: those that do not use the denomination ℓ_1 and those that do. There are $C(n,(\ell_2\ldots,\ell_k))$ ways that do not use the denomination ℓ_1. There are $C(n-\ell_1,\ell)$ ways that involve at least one ℓ_1. It follows that

$$C(n,\ell) = C(n,(\ell_2\ldots,\ell_k)) + C(n-\ell_1,\ell),$$

which you can use in your definition. (To do this, make sure you understand what happens, e.g., when n is smaller than all of the elements in ℓ.)

```
(define (change n denominations)
  (cond ((< n 0) 0)
        ((= n 0) 1)
        ((null? denominations) 0)
        (else (+ (change n (cdr denominations))
                 (change (- n (car denominations))
                         denominations)))))
```

5. (**The Cantor pairing function.**) A *pairing function* p is a function that places the natural numbers $\mathbb{N} = \{0,1,2,\ldots\}$ into one-to-one correspondence with the set of all *pairs* of natural numbers (usually denoted $\mathbb{N}\times\mathbb{N}$). It is somewhat surprising that such a function should exist at all: it shows that the set of natural numbers has the same "size" as the set of all *pairs* of natural numbers. To be more precise, a pairing function p takes two natural numbers x and y as arguments and returns a single number z with the property that the original pair can always be determined exactly from the value z. (In particular, the function maps no more than one pair to any particular natural number.)

   One famous pairing function is the following:

$$p(x,y) = \frac{1}{2}(x+y)(x+y+1)+y\,.$$

   (a) Write a SCHEME function `encode` that computes the pairing function above. (It should take a pair of numbers as arguments and return a single number.)

   (b) As mentioned above, this function has the property that if $(x,y) \neq (x',y')$ then $p(x,y) \neq p(x',y')$: it follows that, in principle, the pair (x,y) can be reconstructed from the single value $z = p(x,y)$. In fact, the values x and y can be reconstructed from $z = p(x,y)$ by first computing the quantities

$$w = \left\lfloor \frac{\sqrt{8z+1}-1}{2} \right\rfloor, \quad \text{and}$$
$$t = \frac{w^2+w}{2}\,.$$

   Then $y = z - t$ and $x = w - y$.

   Write a SCHEME function `decode` that takes as an argument an integer z and produces the pair (x,y) for which $p(x,y) = z$. You'll need the `floor` function: `(floor x)` returns the largest integer less than or equal to `x` (that is, it rounds `x` down to the nearest integer).

   Hint: You may wish to use the `let*` form for this problem. `let*` has the form

```
(let* ((x1 <expr1>)
       (x2 <expr2>)
       ...
       (xk <exprk>))
  <let-expr>)
```

The form is a simple method for writing "nested lets." It is evaluated, informally, as follows. Starting with the external environment, expr1 is evaluated and the variable x1 is immediately bound to the resulting value. Following this, <expr2> is evaluated in the resulting environment and the variable x2 is bound to the value. This continues for the remaining variable/expression pairs. Finally, <let-expr> is evaluated in the resulting environment and its value is returned. Note, for example, that x1 may appear in <expr2>. (Recall that in a regular let expression, the <expri> are all evaluated in the external environment.)

```
(define (encode p)
  (let ((sum (+ (car p) (cdr p))))
    (+ (* sum
          (+ sum 1)
          (/ 1 2))
       (cdr p))))

(define (decode z)
  (let* ((w (floor (/ (- (sqrt (* 8 z)) 1) 2)))
         (t (/ (+ (* w w) w) 2))
         (y (- z t))
         (x (- w y)))
    (cons x y)))
```

6. Write a Scheme function positives which takes a list—call it ℓ—as an argument and returns a list consisting of all elements of ℓ that are positive. In particular, once you have positives defined correctly, you should be able to reproduce the following behavior.

```
> (positives (list -2 -1 0 1 2))
'(1 2)
> (positives (list 2 1 0 -2 -1))
'(2 1)
> (positives '(3 1 -1 1 -1))
'(3 1 1)
```

To keep things simple, it's fine if your function just removes from the list all numbers that are zero or less (leaving duplicates in the remaining list, as shown above).

```
(define (positives numbers)
  (cond ((null? numbers) '())
        ((< (car numbers) 0) (positives (cdr numbers)))
        ((else (cons (car positives) (positives (cdr numbers)))))))
```

7. Write a Scheme function that removes all duplicates from a list. (Hint: you might start by defining a function which removes all duplicates of a particular given value v from a list; then what?)

```
(define (remove-duplicates numbers)
  (define (remove v elements)
    (cond ((null? elements) '())
          ((= v (car elements)) (remove v (cdr elements)))
          (else (cons (car elements) (remove v (cdr elements))))))
  (if (null? numbers) '()
      (cons (car numbers)
            (remove (car numbers)
                    (remove-duplicates (cdr numbers))))))
```