Homework 1: Scheme

This homework is due Wednesday, Feb. 6 (in class).

Hand in a print-out of the Scheme code for the following problems, clearly documented, and in the order of the problems below. Your solutions will be graded on their correctness, style, and documentation. Please test your functions using the Scheme interpreter.

Please program in an applicative style, using lists (not vectors). That is, for this homework you may *not* use the imperative constructs of Scheme, essentially any function with an exclamation point (!) in its name (such as `set!`, `set-car!`, and `set-cdr!`). You may assume that argument lists are not circular.

1. The $3x + 1$ problem concerns iterates of the following function:

$$T(n) = \begin{cases} (3n+1)/2, & \text{if } n \equiv 1 \pmod 2, \\ n/2, & \text{if } n \equiv 0 \pmod 2. \end{cases} \tag{1}$$

which takes odd numbers n to $(3n + 1)/2$ and even numbers n to $n/2$.

   (a) (3 points) Implement the function `T` which computes T.

   (b) (4 points) The iterates of T are defined as follows. Let $T^{(0)}(n) = n$, and for all integers $k > 0$, let $T^{(k)}(n) = T(T^{(k-1)}(n))$.

   Define a function `T-iterate` so that `(T-iterate k n)` returns $T^{(\mathtt{k})}(\mathtt{n})$ for $\mathtt{k} \geq 0$.

   (c) (6 points) The sequence of iterates $(n, T(n), T^{(2)}(n), \ldots)$ is called the trajectory of n. For example, the trajectory of 7 is:

$$7, 11, 17, 26, 13, 20, 10, 5, 8, 4, 2, 1, 2, 1, 2, 1, \ldots$$

   Define a function `trajectory-head` so that `(trajectory-head n k)` returns a list of the first `k`+1 elements of the trajectory of `n`. For example, `(trajectory-head 7 3)` returns the list `(7 11 17 26)`. (This problem illustrates one technique for dealing with infinite data structures in a program.)

   (d) (6 points) The $3x + 1$ conjecture states that for each $n > 0$, there is some k such that $T^{(k)}(n) = 1$. Define a function `trajectory-to-1` such that `(trajectory-to-1 n)` returns the first subsequence of the trajectory of `n` that includes 1, provided that `n` is not a counter-example to the conjecture. For example `(trajectory-to-1 1)` returns `(1)`, and `(trajectory-to-1 7)` returns `(7 11 17 26 13 20 10 5 8 4 2 1)`.

   (e) (extra credit only) Define functions `max-value` and `total-stopping-time` such that `(max-value n)` is the maximum element of `(trajectory-to-1 n)` and `(total-stopping-time n)` is one less than the length of `(trajectory-to-1 n)`.

   Write Scheme procedures that draw graphs of the trajectories of various (small) numbers. (You may use side-effects for output, of course.)

   Let us call `max-value` and `total-stopping-time` "measures" of a number, since they are measures of its trajectory. An integer $n > 0$ is a *peak* in a measure f, if and only if for all $0 < m < n$, $f(m) < f(n)$. These numbers are called peaks because if one graphs the positive integers on the x-axis and the value of some derived measure on the y-axis, then each peak will be a point higher than has been reached before. For example,

3 is a peak in `max-value`. Write a function `peaks` such that `(peaks max-value n)` returns a list of all the peaks in `max-value` from 1 to `n` (inclusive). For example, `(peaks total-stopping-time 10)` should return the list `(1 2 3 6 7 9)`.

2. (5 points) Write a function `remove-2nd` such that `(remove-2nd what where)` returns a the list that is like `where`, except that the second occurrence of `what` in `where` is removed.

3. (10 points) Write a program `(subst new old ls)` that builds a list obtained by replacing each top-level occurrence of the item `old` in the list `ls` with the item `new`. You may assume that an "item" is either a number or a symbol. For example, `(subst 'z 'a '(a b a c a))` returns `(z b z c z)`, `(subst 'z 'a '(a b (a) (c a)))` returns `(z b (a) (c a))`, `(subst 0 1 '(0 1 0 1))` returns `(0 0 0 0)`, `(subst 'dog 'cat '(my dog is fun))` returns `(my dog is fun)`, and `(subst 'two 'one '())` returns `()`.

4. (extra credit only) Write a program `(subst-all new old ls)` that builds a list obtained by replacing each occurrence of the item `old` in the list `ls` with the item `new`. You may assume that an "item" is either a number or a symbol. For example, `(subst-all 'z 'a '(a b a c a))` returns `(z b z c z)`, and `(subst 'z 'a '(a b (a) (c a)))` returns `(z b (z) (c z))`.

5. (10 points) Define a procedure `map-first-two` that works like `map`, except that the procedure argument is always a procedure of two arguments (like +). Use the first and second elements of the list argument as the first pair of arguments to the procedure argument, then the second and third elements, then the third and forth, and so on. If there are fewer than two elements in the list, the empty list is the value. For example: `(map-first-two + '(2 3 4 5 7)` returns `(5 7 9 12)` and `(map-first-two max '(2 4 3 5 4 1)` returns `(4 4 5 5 4)`.

6. (extra credit only) Do some of the exercises 7.8–7.10 on pages 208–209 of *Scheme and the Art of Programming*.

7. (5 points) Define a function `trajectory` such that `((trajectory n) k)` returns the result of `(trajectory-head n k)`. Thus `trajectory` is a curried version of `trajectory-head`.

8. (10 points) Define a function `curry2` that takes a two-argument procedure and returns a version of that procedure. For example `(curry2 trajectory-head)` returns a procedure that is equivalent to what you programmed in the previous problem.

9. (extra credit only) Is it possible to write a general version of `curry` that would curry procedures that take arbitrary numbers of arguments in Scheme? If so, write it. If not state why it cannot be done.

10. (extra credit only) A bag (or multi-set) can be described by a "characteristic function" (whose range is the natural numbers) that determines how many times an element occurs in the bag. For example, the function ϕ such that $\phi($`coke`$) = \phi($`pepsi`$) = 6$ and for all other arguments x, $\phi(x) = 0$ is the characteristic function for a bag containing 6 `coke` and 6 `pepsi` (i.e., a 6-pack of each). In the above bag the "number of occurrences" of `code` is 6. Allowing the user to construct a bag from a characteristic function gives one the power to construct bags that may "contain" an infinite number of elements (such as the bag in which each prime number n occur n times).

    Implement bags by implementing the following operations.

- `(with-characteristic f)`, returns a bag that contains each x `(f` x`)` times.
- `(union b1 b2)`, returns a bag containing all the elements of `b1` and `b2`, where the number of occurrences of x in the result is the sum of the number of occurrences of x in `b1` and `b2`.
- `(intersect b1 b2)`, returns a bag containing all x such that x is both an element of `b1` and of `b2`, where the number of occurrences of x in the result is the minimum of the number of occurrences of x in `b1` and `b2`.
- `(occurrences b e)`, returns the number of occurrences of `e` in `b`.

To allow a bag to have an infinite number of elements, one can allow characteristic functions whose range is either a natural number or the symbol `infinity`. Add this feature to your implementation above.

Is there any sensible way to define the complement of a bag that is analogous to the complement of a set?