Homework 3: Introductory SML Programming

Due: Wednesday, February 16.

Since the purpose of this homework is to ensure basic skills in functional programming, this is an individual homework. That is, for this homework, you are to do the work on your own, not in groups.

For all SML programs, you must run your code with SML. You must also provide evidence that your program is correct (for example, test cases). Hand in a printout of your code and your testing.

You should program in a purely functional style (for example, don't use `ref`).

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 (in SML) the function `T: int -> int` 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: int*int -> int
```

such 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: int*int -> int list` 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: int -> int list
```

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) (10 points, extra credit only; programming, applications—number theory) Define functions

```
max_value: int -> int
total_stopping_time: int -> int
```

such that `(max_value n)` is the maximum element of `(trajectory_to_1 n)`, and such that `(total_stopping_time n)` is one less than the length of `(trajectory_to_1 n)`.

(f) (10 points, extra credit only; programming, applications—number theory) Write SML functions that draw graphs of the trajectories of various (small) numbers. (You may use side-effects for output, of course.)

(g) (10 points, extra credit only; programming, applications—number theory) Let us call `max_value` and `total_stopping_time` "statistics" of a number, since they summarize its trajectory. An integer $n > 0$ is a *peak* in a statistic 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 statistic 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 (curried) function

```
peaks: (int -> int) -> int -> int list
```

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: ''a * ''a list -> ''a list
```

such that `remove_2nd(what,where)` returns a list that is like `where`, except that the second occurrence of `what` in `where` is left out. You may assume that the type `''a` is an equality type (see page 86 of Paulson's book).

3. (10 points) Write a function

```
subst: ''a * ''a * ''a list -> ''a list
```

such that `subst(new, old, ls)` 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 the type `''a` is an equality type (see page 86 of Paulson's book). 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. (10 points) Define a curried function

```
map_first_two: ('a*'a -> 'b) -> 'a list -> 'b list
```

that works like SML's built-in `map` (see page 157) except that the function argument is always a function of two arguments (like the operator +). Use the first and second elements of the list argument as the first pair of arguments to the function 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 (op +) [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]`.

5. (5 points) Define a function `trajectory: int -> int -> int list` such that `((trajectory n) k)` returns the result of `trajectory_head(n,k)`. Thus `trajectory` is a curried version of `trajectory_head`.

6. (5 points) Define a function `curry3: ('a*'b*'c -> 'd) -> 'a -> 'b -> 'c -> 'd` that takes a three-argument function and returns a curried version of that function. Thus `((curry3 f) x y z` returns the result of `f(x,y,z)`.

7. (5 points, extra credit only; semantics, programming) Is it possible to write a general version of `curry` that would curry functions that take arbitrary numbers of arguments in SML? If so, write it. If not state why it cannot be done.

8. (10 points, extra credit only; programming, semantics) 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).

   The elements of a bag are of some equality type, `''elem` (see page 86 of Paulson's book). Let the polymorphic type constructor `bag` be some polymorphic type that you decide on (you can declare this with something like the following).

```
type ''elem bag = ...;
(* or perhaps something like *)
datatype ''elem bag = ...;
```

   The operations on bags are described informally as follows.

   - The function

     ```
     with_characteristic: (''elem -> int) -> (''elem bag)
     ```

     is such that `(with_characteristic f)`, returns a bag that contains each x of type `''elem` `f(`x`)` times.

   - The function

     ```
     union: ''elem bag * ''elem bag -> ''elem bag
     ```

     is such that `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`.

   - The function

     ```
     intersect: ''elem bag * ''elem bag -> ''elem bag
     ```

     is such that `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`.

   - The function

     ```
     occurrences: ''elem bag * ''elem -> int
     ```

     is such that `occurrences(b,e)`, returns the number of occurrences of `e` in `b`.