Homework 9: LISP and Functional Programming

This homework is due Wednesday, May 9 (by 5:00 PM).

Scheme code, clearly documented, in one file, and in order of the problems below, should be mailed to `mobasher` on Zippy. Please also hand in a print-out. Your solutions will be graded on their correctness, style, and documentation. Please test your functions, but do not send us the test data or test results.

You must program in an applicative style. 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 following problems in Scheme programming are given in lieu of the entire Turing machine simulator. You may want to write auxiliary functions to make writing some of these easier.

   (a) (3 points) Define a function `instr-direction` that returns the fourth element of a list, counting the car of the list as the zeroth element (as in `list-ref`), and assuming that the list has at least 5 elements. You may not use `list-ref` itself for this problem.

   (b) (5 points) Write a function `list-update` that takes a list, an index, and an object as an argument and returns a new list that is the same as the original except that at the given index the given object is stored. The indexes are to be interpreted so that the first element of the list is at index 0 (as in `list-ref`). Assume that the index is not "out of range." For example, `(list-update '(x y z) 1 'u)` should return the list `(x u z)`.

   (c) (7 points) Write a function `tape-move-right` that takes as its argument a list containing a nonnegative integer i followed by a list of characters; this function should return a new list containing $i+1$ followed by the original list, unless the length of the original list is less than $i+1$, in which case $i+1$ should be followed by a list containing the original characters followed by the character `#\_`. You may assume that the length of the list of characters is at least as large as i.

   (d) (10 points) A stream can be thought of as a list, the tail of which is not evaluated until it is asked for. The first element of a stream is obtained by the function `head`, the rest of the stream is obtained by the function `tail`. There is a predicate `empty-stream?` that tests whether a stream has any elements. (Streams can be created using `cons-stream` and the global `the-empty-stream`.)

   Define a function `stream->list` that takes a (finite) stream argument and returns list consisting of all the elements of that stream in their original order (that is, the car of the result is the head of the argument, etc.) For example,

   ```
   (stream->list (cons-stream 'a (cons-stream 'b the-empty-stream)))
   ```

   should return the list `(a b)`.

2. (10 points) Write a function `filter` that takes a predicate p and a list and returns a list of all the elements e of the list argument such that p applied to e returns `#t`. For example, `(filter (lambda (x) (> 0 x)) '(1 -3 -5 3))` should return the list `(-3 -5)`. (This is MacLennan's exercise 10-14.)

3. (30 points) Compare the modularity of programs written in a functional style with those written using object-oriented techniques such as mutation, message passing, and inheritance. How do they differ in support for information hiding? Which style allows one to write programs that are easier to read and change? Are the two styles of programming opposed or is it possible to write functional programs that use some object-oriented techniques (such as message passing)?

Extra Credit: You may receive a maximum of 40 points extra (homework) credit if you implement the Turing Machine Simulator in Scheme, in functional style. Credit will only be given if work out all the above problems, hand in them and the simulator on time, and hand in your term paper on time (the Term paper is due May 7). Thus you should do the above work first. To avoid the complexities of I/O, you may define a function `tm-simulator` that takes two arguments: a list and a string; the string represents the tape, the while the list represents the program. This function should return the string representing the final tape. The list representing the program will look, when printed, just like a normal Turing machine program; for example, one might use the following call:

```
(tm-simulator '((0 r = w right 0)) "r_tape")
```

would return the string `"w_tape"`. Be sure to do proper error checking.