Homework 2

Due Tuesday, 21 February

Handout 5

CSCI 334: Spring, 2012

Reading

1. **(Required)** Read Mitchell, Chapters 3, 4.1–4.2 (just skim the recursion and fixed point section)
2. **(Optional)** "Uniprocessor garbage collection techniques", Paul Wilson.

   A thorough overview of collection techniques. Feel free to skip 3.3–3.6.

Problems

1. (*12 points*) .... Filter

   We've already seen how using `mapcar` provides a generic way to easily manipulate collections of data. There are others that are equally useful. We examine one of them in this question.

   (a) Write a function `filter` that takes a predicate function `p` and a list `l`. This function returns a list of those elements in `l` that satisfy the criteria specified by `p`. For example, the following use filters all negative numbers out of a list:

   ```
   * (filter #'(lambda (x) (>= x 0)) '(-1 1 2 -3 4 -5))
   (1 2 4)
   * (defun even (x) (eq (mod x 2) 0))
   * (filter #'even '(6 4 3 5 2))
   (6 4 2)
   ```

   You will need to use the built-in operation `funcall` to call the function passed to `filter` as a parameter. That is, the function

   ```
   (defun example (f)
     (funcall f a1 ... an)
   )
   ```

   applies `f` to arguments `a1` – `an`. You may not use the built-in functions `remove-if` and `remove-if-not` in your solution.

   (b) Suppose that we are using lists to represent sets (in which there are no repeated elements). Use your `filter` function to define functions `set-union` and `set-interset` that take the union and intersection of two sets, respectively:

   ```
   * (set-union '(1 2 3) '(2 3 4))
   (1 2 3 4)
   * (set-intersect '(1 2 3) '(2 3 4))
   (2 3)
   ```

   You may find the built-in function `(member x l)` described in the 334 Lisp FAQ handy.

   (c) Now, use `filter` to implement the function `exists`, which returns true if there is at least one `a` in `l` such that `(p a)` returns true:

   ```
   * (exists #'(lambda (x) (eq x 0)) '(-1 0 1))
   t
   * (exists #'(lambda (x) (eq x 2)) '(-1 0 1))
   nil
   ```

You may assume that `p` will terminate without crashing for all `a`.

Lastly, the function `all` returns true if `(p a)` is true for all `a` in `l`:

```
* (all #'(lambda (x) (> x -2)) '(-1 0 1))
t
* (all #'(lambda (x) (> x 0)) '(-1 0 1))
nil
```

Implement this function using `exists`. (That is, you should not need to recursively traverse `l` or use `filter` directly.)

Turn in a file containing your solution with the unix command "`turnin -c 334 filename.lisp`", as you did last week. Please include a printout in your problem set solutions as well.

2. *(15 points)* .... Reference Counting

Mitchell, Problem 3.6

3. *(5 points)* .... Parse Tree

Mitchell, Problem 4.1

4. *(10 points)* .... Parsing and Precedence

Mitchell, Problem 4.2

5. *(5 points)* .... Lambda Calculus Reduction

Mitchell, Problem 4.3

6. *(15 points)* .... Symbolic Evaluation

The Lisp program fragment

```
(defun f (x) (+ x 4))
(defun g (y) (- 3 y))
(f (g 1))
```

can be written as the following lambda expression:

$$\left(\underbrace{(\lambda f.\lambda g.f\ (g\ 1))}_{\text{main}}\ \underbrace{(\lambda x.x+4)}_{f} \right)\ \underbrace{(\lambda y.3-y)}_{g}$$

Reduce the expression to a normal form in two different ways, as described below.

(a) *(5 points)* Reduce the expression by choosing, at each step, the reduction that eliminates a λ as far to the *left* as possible.

(b) *(5 points)* Reduce the expression by choosing, at each step, the reduction that eliminates a λ as far to the *right* as possible.

(c) (*5 points*) In pure λ-calculus, the order of evaluation of subexpressions does not effect the value of an expression. The same is true for Pure Lisp: if a Pure Lisp expression has a value under the ordinary Lisp interpreter, then changing the order of evaluation of subterms cannot produce a different value. However, that is not the case for a language with side effects. To give a concrete example, consider the following "Java"-like code fragment:

```
int f(int a, int b) {
  ...
}

{
  int x = 0;

  System.out.println(f(e1,e2));
}
```

Write a function `f` and expressions `e1` and `e2` for which evaluating arguments left-to-right and right-to-left produces a different result. Your expressions may refer to `x`. Try it out in your favorite imperative language — C, C++, Java, etc. Which evaluation order is used?

7. (*10 points*) .... Lambda Reduction with Sugar

Here is a "sugared" lambda-expression using `let` declarations:

$$\begin{array}{l} \texttt{let}\ compose = \lambda f.\, \lambda g.\, \lambda x.\, f(g\,x)\ \texttt{in} \\ \quad \texttt{let}\ h = \lambda x.\, x + x\ \texttt{in} \\ \qquad ((compose\,h)\,h)\,3 \end{array}$$

The "de-sugared" lambda-expression, obtained by replacing each `let` $z = U$ `in` V by $(\lambda z.\, V)\, U$ is

$$\begin{array}{l} (\lambda compose. \\ \quad (\lambda h.\ ((compose\ h)\ h)\ 3)\ \ (\lambda x.\, x + x)) \\ \quad (\lambda f.\, \lambda g.\, \lambda x.\, f(g\ x)) \end{array}$$

This is written using the same variable names as the `let`-form in order to make it easier to read the expression.

Simplify the desugared lambda expression using reduction. Write one or two sentences explaining why the simplified expression is the answer you expected.

8. (*10 points*) .... Advanced GC (Bonus Question)

Read the Wilson Garbage Collection paper. This paper discusses many foundation ideas behind modern garbage collection. Answer some or all of the following questions with one or two sentences each:

- What are their limitations of mark-and-sweep and reference-counting collectors?
- What problem does copying-collection solve?
- What is the main insight behind incremental collection?
- What about generational collectors? When will they work well? When will they work poorly?

Most modern collectors use a combination of several techniques to best handle current systems with built-in concurrency and much larger heaps. Here are a few additional resources for further reading that are available on cs334 the web page:

- The Sun HotSpot Java Virtual Machine white paper, which contains a brief overview of the collector in the standard Sun JVM.

- “Beltway: Getting Around Garbage Collection Gridlock”, Steve M. Blackburn, Richard Jones, K. S. McKinley, and J. Eliot B. Moss, PLDI, 2002.
- “Composing High-Performance Memory Allocators”, E. D. Berger, B. G. Zorn, and K. S. McKinley, PLDI 2001.

The first should be quite accessible after the Wilson paper. The second and third are more recent, and fairly dense, garbage collection research papers.