What To Turn In

Please hand in work in two pieces, one for the Problems and one for the Programming (Partner Optional):

Problems: Turn in handwritten or typed answers by the due date. Be sure your work is stapled and that your answers are clearly marked and in the correct order. Include only your 334 ID, and not your name.

Programming: Turn in a printout of your code separately from your problem answers. Again, use only your 334 ID at the top. If you worked with a partner, only one person needs to hand in a printout, but be sure both IDs are at the top of the file. Turn in an electronic copy as well using the instructions at the end of the programming questions.

Reading

1. (Required) Read Mitchell, Chapters 3, 4.1–4.2 (just skim the recursion and fixed point section)
   A thorough overview of collection techniques. (You can most likely skip the details of 3.3–3.6.)

Self Check

S1. ........................................................................... Parse Tree
   Mitchell, Problem 4.1

S2. ........................................................................... Lambda Calculus Reduction
   Mitchell, Problem 4.3

Problems

Q1. (15 points) ............................................... Reference Counting
   Mitchell, Problem 3.6

Q2. (10 points) ............................................... Parsing and Precedence
   Mitchell, Problem 4.2
Q3. (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:

\[
\begin{pmatrix}
(\lambda f.\lambda g. f (g 1)) & (\lambda x.x + 4) \\
(\lambda y.3 - y) & g
\end{pmatrix}
\]

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 \(\lambda\) as far to the left as possible.

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

(c) (5 points) In pure \(\lambda\)-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:

```java
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?

Q4. (10 points) Lambda Reduction with Sugar

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

```
let compose = \(f.\lambda g. \lambda x. f(g x)\) in
let h = \(\lambda x. x + x\) in
((compose h) h) 3
```

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

\[
(\lambda compose. \\
(\lambda h. ((compose h) h) 3) (\lambda x.x + x))
(\lambda f.\lambda g. f(g x))
\]

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.
**Q5. (20 points) Garbage Collection Techniques**

Read the Wilson Garbage Collection paper. This paper discusses many foundational ideas behind modern garbage collection. Please answer the following questions with one or two sentences each. **The most credit will be given for clear, concise answers — you should not need to write much.**

(a) What are the limitations of mark-and-sweep and reference-counting collectors?

(b) What problem does copying-collection solve?

(c) What is the main insight behind incremental collection?

(d) 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. If you’re curious, have a look at the additional GC papers on the Readings web page, including papers on the HotSpot Java Virtual Machine implements garbage collection, the Immix collector, and others.

---

**Programming (Partner Optional)**

**P1. (15 points) Filter**

You may work with a partner on this problem if you’d like. However, it is not required. If you’d like to be matched with a partner, please send me email and I’ll pair you up as emails arrive.

Your GitLab account will have a project for your to use for this question. You can follow the same instructions as last week for cloning it and (optionally) adding a partner. You should answer the following in the hw2.lisp file in your repository.

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 two examples filter all negative numbers out of a list and filter all odd 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-intersect` 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 \texttt{(member x l)} described in the 334 Lisp FAQ handy.

(c) Now, use \texttt{filter} to implement the function \texttt{exists}. Given a predicate function \texttt{p} and a list \texttt{l}, this function returns true if there is at least one \texttt{a} in \texttt{l} such that \texttt{(p a)} returns true:

\begin{verbatim}
* (exists #'(lambda (x) (eq x 0)) '(-1 0 1))
  t
* (exists #'(lambda (x) (eq x 2)) '(-1 0 1))
  nil
\end{verbatim}

You may assume that \texttt{p} will terminate without crashing for all \texttt{a}.

Lastly, the function \texttt{all} returns true if \texttt{(p a)} is true for all \texttt{a} in \texttt{l}:

\begin{verbatim}
* (all #'(lambda (x) (> x -2)) '(-1 0 1))
  t
* (all #'(lambda (x) (> x 0)) '(-1 0 1))
  nil
\end{verbatim}

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

\textbf{What To Turn In.}

- Your code should be documented (comment lines start with “;”) and include the your 334 ID number at the top. (Include both IDs if working with a partner.)
- Hand in a paper copy of your code separate from the written questions. (Only once per pair.)
- Commit and push all changes to your GitLab repository.
- Verify your commits by navigating to your lab repository in a web browser and examining the version that is stored there.
- If you have a partner, the shared repository you are using is either your own or your partner's. The other one will be unused. There is no need to do anything to that repository. Our submission scripts will ignore unused repositories and look only at the ones with completed solutions.