

---

## Reading

---

1. Read Mitchell, Chapters 3, 4.1–4.2 (just skim the recursion and fixed point section)
2. (optional) “Comparing Mark-and-Sweep and Stop-and-copy Garbage Collection” by Benjamin Zorn, which is available from the cs334 web.

---

## Problems

---

1. (15 points) ..... Reference Counting

Mitchell, Problem 3.6

2. (5 points) ..... Parse Tree

Mitchell, Problem 4.1

3. (10 points) ..... Parsing and Precedence

Mitchell, Problem 4.2

4. (5 points) ..... Lambda Calculus Reduction

Mitchell, Problem 4.3

5. (10 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  $\lambda$  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.

6. (10 points) ..... Lambda Reduction with Sugar

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

```
let compose = λf. λg. λx. f(g x) in
  let h = λx. 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

```
(λcompose.
  (λh. ((compose h) h) 3) (λx. x + x))
(λf. λg. λx. 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.

7. (10 points) ..... Order of Evaluation

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.

To give a concrete example, consider the following section of Lisp code

```
(defvar a '(1 2 3))
(defvar b '(4 5 6))
...
(defun f (x y z) (cons (car x) (cons (car y) (cdr z))))
...
(f e1 e2 e3)
```

The ordinary evaluation order for the function call

```
(f e1 e2 e3)
```

is to evaluate the arguments  $e1 e2 e3$  from left to right and then pass this list of values to the function  $f$ .

Give an example of Lisp expressions  $e1 e2 e3$ , possibly using functions `rplaca` or `rplacd` with side effects, so that evaluating expressions from left to right gives a different result from evaluating them from right to left. You may refer to the lists  $a$  and  $b$  in your expressions if you like, and you may define them to be other values as well. Explain briefly, in one or two sentences, why one order of evaluation is different from the other.

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

Read the paper “Comparing Mark-and-Sweep and Stop-and-copy Garbage Collection” by Benjamin Zorn from the readings page on the web site.

This paper introduces several additional aspects of garbage collection. Answer the following questions with one or two sentences:

- (a) What is Stop-and-Copy collection, and why is it useful?
- (b) What is generational collection and why does it enhance both Mark-and-Sweep and Stop-and-Copy?

- (c) Would generational collection improve a reference-counting collector?
- (d) How does Zorn measure and evaluate different collection techniques? What can you conclude from the paper?

Most modern collectors use a combination of several techniques, and they must deal with other issues not addressed in Zorn's study, such as 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 second and third are more recent, and fairly dense, garbage collection research papers.