Homework 2

Due 21 February

Handout 9
CSCI 334: Spring 2018

Notes

This homework has three types of problems:

Self Check: You are strongly encouraged to think about and work through these questions, and may do so with a partner. Do not submit answers to them.

Problems: You will turn in answers to these questions.

Pair Programming: This part involves writing Lisp code. You are required to work with a partner for this section. If you have a preferred partner, be sure to email me (dbarowy@cs.williams.edu) with your partner's name **by the evening of Tuesday, Feb 13**. Your partner may be the same person you worked with for Homework #1. If I have not received an email from you by then, I will assign you a partner.

Turn-In Instructions

You must turn in your work for this assignment in two parts, one for the Problems part and one for the Pair Programming part. You will be assigned two GitHub repositories.

Note that this assignment should be anonymized: neither your name nor your partner's name should appear in any of the files submitted with this assignment except the file "`collaborators.txt`" (see below).

Problems: Problem sets should be typed up using LaTeX and submitted using your `hw2-problems` repository. For example, if your GitHub username is `dbarowy`, then your repository will be called `hw2-problems-dbarowy`. Be sure that your work is committed and pushed to your repository by Wednesday, February 21 at 11:59pm.

If you discuss the problem set with your partner or with a study group, please be sure to include their names in a `collaborators.txt` file in your repository.

Pair Programming: Programming solutions should be typed up and submitted using your partner repository. For example, if your GitHub username is `dbarowy` and you are working with a partner whose username is `wjannen`, look for a repository called `hw2-dbarowy-wjannen` (usernames will be in alphabetical order). Be sure that your work is committed and pushed to your repository by Wednesday, February 21 at 11:59pm.

Lisp files should have a "`.lisp`" suffix. For example, the pair programming problem **P1a** should appear as the file `p1a.lisp`. Your code should be documented using comments. Comment lines start with a "`;`".

Reading

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

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

Self Check

S1. (5 points) ..... Parse Tree

Draw the parse tree for the derivation of the expression "`25`" described in section 4.1.2 (bottom of page, Mitchell p. 53). Is there another derivation for "`25`"? Is there another parse tree?

S2. (5 points) ..... Lambda Calculus Reduction

Use lambda calculus reduction to find a shorter expression for $(\lambda x.\lambda y.xy)(\lambda x.xy)$. Begin by renaming bound variables. You should do all possible reductions to get the shortest possible expression. What goes wrong if you do not rename bound variables?

Problems

Q1. (15 points) ..... Reference Counting

This question is about a possible implementation of garbage collection for Lisp. Both impure and Pure Lisp have lambda abstraction, function application, and elementary functions `atom`, `eq`, `car`, `cdr`, and `cons`. Impure Lisp also has `rplaca`, `rplacd` and other functions that have side-effects on memory cells.

Reference counting is a simple garbage collection scheme that associates a reference count with each datum in memory. When memory is allocated, the associated reference count is set to 0. When a pointer is set to point to a location in memory, the count for that location is incremented. If a pointer to a location is reset or destroyed, the count for the location is decremented. Consequently, the reference count always tells how many pointers there are to a given datum. When a count reaches 0, the datum is considered garbage and returned to the free-storage list.

For example, after evaluation of `(cdr (cons 1 (cons 2 3)))`, the first `cons` is garbage, but the `(cons 2 3)` is not.

(a) (8 points) Consider the Lisp expression `(car (cdr (cons (cons` a b`) (cons` c d`))))` where a, b, c, and d are (previously-defined) names for other objects.

i. Suppose that a reference counting collector is run once after the entire expression is evaluated. How many of the three `cons` cells generated by the evaluation of this expression can be returned to the free-storage list? Describe the steps taken by the collector.

ii. Does your answer change if garbage collection is first run after `cdr` and then again after `car`? Describe the steps taken by the collector.

(b) (7 points) The "impure" Lisp function `rplaca` takes as arguments a cons cell c and a value v and modifies c's address field to point to v. Note that this operation does not produce a new cons cell; it modifies the one it receives as an argument. The function `rplacd` performs the same function with respect the decrement portion of its argument cons cell.

Lisp programs using `rplaca` or `rplacd` may create memory structures that cannot be garbage collected properly by reference counting.

i. Describe a configuration of `cons` cells that cannot be effectively garbage collected by a reference counting collector. Explain why the algorithm does not work properly on this structure.

ii. Write a Lisp program that creates such a structure using `rplaca` or `rplacd`.

Q2. (10 points) ..... Parsing and Precedence

Draw parse trees for the following expressions, assuming the grammar and precedence described in Example 4.2 (Mitchell, p. 56):

(a) `1 + 1 * 1`

(b) `1 + 1 - 1`.

(c) `1 - 1 + 1 - 1 * 1`, if + is given higher precedence than −.

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:

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

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 affect the value of an expression. The same is true for Pure Lisp: if a Pure Lisp expression has a value in 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 like Java.

i. Write a Java instance method `f` and expressions `e1` and `e2` for which evaluating arguments left-to-right and right-to-left produces different results. (Hint: Recall that in Java, an instance method may refer to variables declared outside of the scope of the function definition.)

ii. What evaluation order is used by Java?

Q4. (10 points) ..... Lambda Reduction with Sugar

Lambda expressions can be made easier to understand by the use of "syntactic sugar." Syntactic sugar is additional syntax that simplifies readability while leaving the meaning (semantics) of a language expression unchanged.

For example, here is a "sugared" lambda expression using some extra syntax known as a `let` declaration:

$$\begin{array}{l} \texttt{let}\ \mathit{foo} = \lambda x.\lambda y.x+y\ \texttt{in} \\ \quad \mathit{foo}\ 2\ 3 \end{array}$$

The above expression may be "desugared" by replacing each `let` $z = U$ `in` V with $(\lambda z.\, V)\, U$. First, we identify z, U, and V:

$$\begin{array}{rcl} z & = & \mathit{foo} \\ U & = & \lambda x.\lambda y.x+y \\ V & = & \mathit{foo}\ 2\ 3 \end{array}$$

which yields:

$$(\lambda foo.(foo\ 2\ 3))(\lambda x.\lambda y.x + y))$$

and after reducing this expression, the value 5.

(a) Desugar the following expression:

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

(b) Simplify the desugared lambda expression using reduction. Briefly explain 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 garbarge collection, the Immix collector, and others.

Q6. (9 points) ..... Challenge Problem

There are a wide variety of algorithms to choose from when implementing garbage collection for a specific language. In this problem, we examine one algorithm for finding garbage in Pure Lisp (Lisp without side effects) based on the concept of *regions*.

Generally speaking, a region is a section of the program text. For the purposes of this assignment, consider each function as a separate region. Region-based collection reclaims garbage each time program execution leaves a region. Since we are treating functions as regions in this problem, our version of region-based collection will try to find garbage each time a program returns from a function call.

Here is a simple idea for region-based garbage collection:

> When a function exits, free all the memory that was allocated during execution of the function.

However, this simple idea has a flaw: some memory locations that are freed may still be accessible to the program. Since reclaimed memory may be reused by the language's memory allocator, accessing that memory with the expectation that the original values are unmodified will lead to unexpected behavior.

(a) Explain the flaw by writing a Lisp program whose operation will be broken by the above garbage collection scheme. Briefly explain why your program is problematic.

(b) How can we fix the above simple scheme so that your function does not break? It is not necessary for your method to find all garbage, but the locations that are freed should really be garbage. Your answer should be in the form:

When a function exits, free all memory allocated by the function except ...

Briefly explain why the new algorithm does not break your program.

(c) Now assume that you have a correctly functioning region-based garbage collector. Does your region-based collector have any advantages or disadvantages over a simple mark-and-sweep collector?

(d) Could a region-based collector like the one described in this problem work for a programming language with side effect like Java? If you think the problem is more complicated for Java, briefly explain using a Java program. The point of this question is to explore the relationship between side effects and a simple form of region-based collection.

Pair Programming

P1. (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 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-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`. Given a predicate function `p` and a list `l`, this function 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.)