Higher-Order Functions

COMP 105 Assignment

Due Tuesday, February 21, 2017 at 11:59PM

Contents

Overview **1**

Setup **2**

Dire Warnings **2**

Reading Comprehension (10 percent) **2**

Programming and Proof (90 percent) **5**
- Overview . . . 5
- Book problems . . . 6
- Relating imperative code to functional code . . . 7
- Graph problems . . . 8
- Calculational reasoning about functions . . . 9
- Ordered lists . . . 10

What and how to submit **11**

Avoid common mistakes **11**

How your work will be evaluated **12**
- Structure and organization . . . 12
- Functional correctness . . . 12
- Proofs and inference rules . . . 13

This assignment is all individual work. There is **no pair programming**.

Overview

The purpose of this assignment is to give you sufficient experience using first-class and higher-order functions that you can incorporate them into your programming practice. You will use existing higher-order functions, define higher-order functions that consume functions, and define higher-order functions that return functions. The assignment builds on what you've already done, and it adds new ideas and techniques that are described in sections 2.7, 2.8, and 2.9 of Ramsey's book.

Setup

The executable μScheme interpreter is in `/comp/105/bin/uscheme`; if you are set up with `use comp105`, you should be able to run `uscheme` as a command. The interpreter accepts a `-q` ("quiet") option, which turns off prompting. Your homework will be graded using `uscheme`. When using the interpreter interactively, you may find it helpful to use `ledit`, as in the command

```
ledit uscheme
```

Dire Warnings

The μScheme programs you submit must not use any imperative features. **Banish `set`, `while`, `print`, and `begin` from your vocabulary! If you break this rule for any exercise, you get No Credit for that exercise.** You may find it useful to use `begin` and `print` while debugging, but they must not appear in any code you submit. As a substitute for assignment, use `let` or `let*`.

Except as noted below, do not define helper functions at top level. Instead, use `let` or `letrec` to define helper functions. When you do use `let` to define inner helper functions, avoid passing as parameters values that are already available in the environment.

Your solutions must be valid μScheme; in particular, they must pass the following test:

```
/comp/105/bin/uscheme -q < myfilename > /dev/null
```

without any error messages or unit-test failures. If your file produces error messages, we won't test your solution and you will earn No Credit for functional correctness. (You can still earn credit for structure and organization). If your file includes failing unit tests, you might possibly get some credit for functional correctness, but we cannot guarantee it.

We will evaluate functional correctness by testing your code extensively. **Because this testing is automatic, each function must be named be exactly as described in each question. Misnamed functions earn No Credit.**

Reading Comprehension (10 percent)

As usual, you can download the questions[1].

1. The first step in this assignment is to learn the standard higher-order functions on lists—you will use them a lot. Review Sections 2.7.2, 2.8.1, and 2.8.2. Now consider each of the following functions:

   ```
   map  filter  exists?  all?  curry  uncurry  foldl  foldr
   ```

   Below, please put the name of each function below to the statement that *most accurately* describes it. Any given statement may describe more than one function, exactly one function, or no functions. Any given function name must appear in exactly one position.

[1] ./cqs.hofs.txt

- Each of the functions listed after the arrow *might* return a Boolean:

  →

- Each of the functions listed after the arrow *always* returns a Boolean:

  →

- None of the functions listed after the arrow *ever* returns a Boolean:

  →

2. Here are the same functions again:

   ```
   map  filter  exists?  all?  curry  uncurry  foldl  foldr
   ```

   And again, please put the name of each higher-order function next to the statement that most accurately describes it:

   - Each of the functions listed after the arrow takes a list and a function. Each one *always* returns a list of *exactly* the same size as the original list:

     →

   - Each of the functions listed after the arrow takes a list and a function. Each one *always* returns a list of *at least* the same size as the original list:

     →

   - Each of the functions listed after the arrow takes a list and a function. Each one *always* returns a list of *at most* the same size as the original list:

     →

   - Each of the functions listed after the arrow *might* return a list:

     →

   - *None* of the functions listed after the arrow *ever* returns a list:

     →

3. Here are the same functions again:

   ```
   map  filter  exists?  all?  curry  uncurry  foldl  foldr
   ```

   And again, please put the name of each higher-order function next to the statement that most accurately describes it:

   - Each of the functions listed after the arrow takes one argument, which is a function that itself takes two arguments:

     →

   - Each of the functions listed after the arrow takes one argument, which is a function:

     →

   - Each of the functions listed after the arrow takes more than one argument:

     →

You are now ready to tackle most parts of exercise 14.

4. Review section 2.7 from page 112 to page 115.

   (a) Define function `twice` using `val` and `lambda`, not `define`.

   (b) Using `lambda`, write an *expression* that evaluates to the absolute-value function. Don't use any definition forms. (*Hint*: you can negate a number n by subtracting it from zero.)

5. Review the difference between `foldr` and `foldl` in section 2.8.1. You may also find it helpful to look at the implementation in code chunk 125b on page 125.

   (a) Do you expect `(foldl + 0 '(1 2 3))` and `(foldr + 0 '(1 2 3))` to be the same or different?

   (b) Do you expect `(foldl cons '() '(1 2 3))` and `(foldr cons '() '(1 2 3))` to be the same or different?

   (c) Look at the initial basis on page 149. Give one example of a function, other than `+` or `cons`, that can be passed as the first argument to `foldl` or `foldr`, such that `foldl` *always returns exactly the same result* as `foldr`.

   (d) Give one example of a function, other than `+` or `cons`, that can be passed as the first argument to `foldl` or `foldr`, such that `foldl` *may return a different result* from `foldr`.

   You are now ready to tackle all parts of exercises 14 and 15.

6. µScheme provides syntactic sugar for records, which are made from cons cells. Review the record syntax in section 2.16.6, which starts on page 183.
   You may also find it helpful to scan the tree data structure in section 2.6.

   Given the record definition

   ```
   (record course (room instructor enrollment))
   ```

   answer these questions:

   (a) How many arguments does function `make-course` expect?

   (b) How many arguments does function `course?` expect?

   (c) Is the following equation a valid algebraic law? That is, does it hold for all values of `n`?

   ```
   (course? (make-course '(Barnum 008) 'Ramsey n)) == #t
   ```

   (d) Is the following equation a valid algebraic law? That is, does it hold for all values of `r`, `i`, and `n`?

   ```
   (course-room (make-course r i n)) == i
   ```

   You are now ready to tackle the record operations in part (d) of problem 19.

7. This question builds on the record syntax described in section 2.16.6. Review function composition and currying, which are described in section 2.7.2. Assume you have the following definitions:

   ```
   (record prof (building room courses))

   (val rockstar  (make-prof 'Halligan 241 '(170)))
   ```

```
(val greybeard (make-prof 'Halligan 222 '(105 150TW)))
(val electric  (make-prof 'Halligan 205 '(105)))

(val nearby? (o ((curry =) 'Halligan) prof-building))
```

Answer these questions:

(a) How many arguments does `nearby?` expect, and what values are acceptable?

(b) What values may `nearby?` return?

(c) What does function `nearby?` do, and how does it work?

(d) If I evaluate the expression `(nearby? rockstar)`, what do you expect to happen and why?

(e) If I evaluate the expression `(nearby? greybeard electric)`, what do you expect to happen and why?

(f) If I evaluate the expression `(nearby? '(Halligan 107B (7)))`, what do you expect to happen and why?

*You are now ready to tackle the first three parts of exercise 19, as well as problem **M** below.*

8. Section 2.9.1 on page 129 describes the "third approach" to polymorphism. Here is a (weak) form of equality test:

```
(val equal-on-zero?
  (lambda (f g) (equal? (f 0) (g 0))))
```

Suppose function `specialized-set-ops` is passed value `equal-on-zero?`. Answer these questions:

(a) Give examples of two different values that might be stored in a set that uses `equal-on-zero?`.

(b) Explain in general what sorts of values may be stored in such a set.

(c) Give examples of two values that are not actually equal, but that would be considered equal by `equal-on-zero?`. (*Hint*: Look for ideas in previous homeworks.)

(d) Does μScheme have a primitive or predefined function that could be used in place of `equal-on-zero?`? Would it give more accurate results? Justify your answer.

You are now ready to tackle the final part of exercise 19.

Programming and Proof (90 percent)

Overview

For this assignment, you will do Exercises **14 (b-f,h,j)**, **15**, and **19**, from pages 201 to 204 of Ramsey, plus the exercises **A**, **G1**, **G2**, **G3**, **M**, and **O** below.

A summary of the initial basis can be found on page 149. A copy was handed out in class—while you're working on this homework, *keep it handy*.

As always, each top-level function you define must be accompanied by a contract and by unit tests written with `check-expect` or `check-error`. Each internal function written with `lambda` should be accompanied by a contract, but internal functions cannot be unit-tested.

Book problems

14. *Higher-order functions*. Do Exercise 14 on page 201 of Ramsey, parts (b) to (f), part (h), and part (j). **You must *not* use recursion—solutions using recursion will receive No Credit.** This restriction applies only to code you write. For example, `gcd`, which is in the initial basis, or `insert`, which is given, may use recursion.

For this problem only, you may define *one* helper function at top level.

Related reading: For material on higher order functions, see sections 2.8.1 and 2.8.2 starting on page 121. For material on `curry`, see section 2.7.2.

15. *Higher-order functions*. Do Exercise 15 on page 202. **You must *not* use recursion—solutions using recursion will receive No Credit.** As above, this restriction applies only to code you write.

For this problem, you get full credit if your implementations return correct results. You get **EXTRA CREDIT**[2] if you can duplicate the behavior of `exists?` and `all?` exactly. To earn the extra credit, it must be impossible for an adversary to write a μScheme program that produces different output with your version than with a standard version. However, the adversary is not permitted to change the names in the initial basis.

Related reading: Examples of `foldl` and `foldr` in sections 2.8.1 and 2.8.2 starting on page 121. You may also find it helpful to study the implementations of `foldl` and `foldr` at the end of section 2.8.3 on page 125. Information on `lambda` can be found in section 2.7. to the top of page 115.

19. *Functions as values*. Do Exercise 19 on page 204 of Ramsey. **You cannot represent these sets using lists.** If any part of your code uses `cons`, `car`, `cdr`, or `null?`, you are doing the problem wrong.

Do all four parts:

- Parts (a) and (b) require no special instructions.
- In part (c), your `add-element` function must take two parameters: the element to be added as the first parameter and the set as the second parameter.

  Also in part (c), compare values for equality using the `equal?` function.
- In part (d), when you code the third approach to polymorphism, write a function `set-ops-from` which places your set functions in a record. Use the syntactic sugar described in the book in Section 2.16.6 on page 183.

  In particular, your submission must include this record definition:

  ```
  (record set-ops (empty member? add-element union inter diff))
  ```

  Code your solution to part (d) as a function `set-ops-from`, which will accept one argument (an equality predicate) and will return a record created by calling `make-set-ops`. Your function might look like this:

[2] In your README, please identify this credit as EXACT-EXISTS.

```
(define set-ops-from (eq?)
  (let ([empty   ...]
        [member? ...]
        [add     ...]
        [union   ...]
        [inter   ...]
        [diff    ...])
    (make-set-ops empty member? add union inter diff)))
```

Fill in each `...` with your code.

To help you get part (d) right, we recommend that you use these unit tests:

```
(check-expect (procedure? set-ops-from)   #t)
(check-expect (set-ops? (set-ops-from =)) #t)
```

And to write your own unit tests for the functions in part (d), you may use these definitions:

```
(val atom-set-ops (set-ops-from =))
(val nullset      (set-ops-empty atom-set-ops))
(val member?      (set-ops-member? atom-set-ops))
(val add-element  (set-ops-add-element atom-set-ops))
(val union        (set-ops-union atom-set-ops))
(val inter        (set-ops-inter atom-set-ops))
(val diff         (set-ops-diff atom-set-ops))
```

Related reading: For functions as values, the examples of `lambda` in the first part of section 2.7 on page 112. Also function composition and currying in section 2.7.2. For polymorphism, section 2.9, which starts on page 125.

Relating imperative code to functional code

A. *Good functional style.* The Impcore function

```
(define f-imperative (y) (locals x) ; x is a local variable
  (begin
    (set x e)
    (while (p? x y)
       (set x (g x y)))
    (h x y)))
```

is in a typical imperative style, with assignment and looping. Write an equivalent μScheme function `f-functional` that doesn't use the imperative features `begin` (sequencing), `while` (goto), and `set` (assignment).

- Assume that `p?`, `g`, and `h` are free variables which refer to externally defined functions.
- Assume that `e` is an arbitrary expression.
- Use as many helper functions as you like, as long as they are defined using `let` or `letrec` and not at top level.

Hint #1: If you have trouble getting started, rewrite `while` to use `if` and `goto`. Now, what is like a `goto`?

Hint #2: `(set x e)` binds the value of `e` to the name `x`. What other ways do you know of binding the value of an expression to a name?

Don't be confused about the purpose of this exercise. The exercise is a *thought experiment.* We don't want you to write and run code for some *particular* choice of `g`, `h`, `p?`, `e`, `x`, and `y`. Instead, we want you write a function that works the same as `f-imperative` given *any* choice of `g`, `h`, `p?`, `e`, `x`, and `y`. So for example, if `f-imperative` would loop forever on some inputs, your `f-functional` must also loop forever on exactly the same inputs.

Once you get your mind twisted in the right way, this exercise should be easy. The point of the exercise is not only to show that you can program without imperative features, but also to help you develop a technique for eliminating such features.

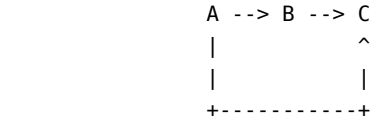
Related reading: No part of the book bears directly on this question. You're better off reviewing your experience with recursive functions and perhaps the solutions for the Scheme assignment.

Graph problems

From COMP 15, you should be familiar with graphs and graph algorithms. In the next few problems you will work with two different representations of *directed* graphs:

- The first representation is a *list of edges*, where a single edge is represented by a two-element list. For example, the list `(A B)` represents an edge from `A` to `B`.
- The second representation is a *successors map*: a graph is represented by an association list in which each node is associated with a list of its successors.

For example, the ASCII-art graph



could be represented as an edge list by `'((A B) (B C) (A C))` and as a successors map by `'((A (B C)) (B (C)) (C ()))`.

Note: The graph problems below can be solved using only first-order functions. But you will find problem **G3** much easier if you use `let`, `lambda`, and either of the fold functions.

Related reading: The previous assignment. The definitions of `equal?` in section 2.3.1 (basic recursive functions on lists). Material on association lists in section 2.3.6. The definition of `=alist?` in section 2.9, which starts on page 125.

G1. *Edge-list representations*. A single graph may have many different representations as an edge list. For example, the graph pictured above can be represented by either of the following edge lists:

```
'((A B) (B C) (A C))
'((A B) (A C) (B C))
```

Define a function `=edge-list?` which takes as arguments two edge lists and returns a Boolean indicating whether they represent the same graph. To test your function, choose three different graphs, and use `check-expect` to demonstrate that your function works on each one.

Hint: You have already written this function, but you know it by another name.

G2. *Successors-map representations*. A single graph may have many different representations as a successors map. For example, the graph pictured above can be represented by any of the following successors maps:

```
'((A (B C)) (B (C)) (C ()))
'((A (C B)) (B (C)) (C ()))
'((B (C)) (A (C B)) (C ()))
```

Define a function `=successors-map?` which takes as arguments two successors maps and returns a Boolean indicating whether they represent the same graph. To test your function, choose two different graphs, and use `check-expect` to demonstrate that your function works on each one.

Hint: There are at least two good ways to approach this problem. You can code it directly, or you can generalize the book function `=alist?` using the third approach to polymorphism. If you use the polymorphic approach, you will find yourself able to reuse more code.

G3. *Converting representations*.

(a) Write function `successors-map-of-edge-list`, which accepts a graph in edge-list representation and returns a representation of the same graph in successors-map representation.

(b) Write function `edge-list-of-successors-map`, which accepts a graph in successors-map representation and returns a representation of the same graph in edge-list representation. You must assume that in the argument graph, every node has at least one incoming edge or one outgoing edge. Otherwise the graph cannot be represented using an edge list.

Write unit tests for each function. In your unit tests, you may find it convenient to use `=edge-list?` and `=successors-map?`.

Hint: By 105 standards, the solution to this problem requires a lot of code. Your new best friend is `let*`.

Calculational reasoning about functions

M. *Reasoning about higher-order functions*. Using the calculational techniques from Section 2.4.5, which starts on page 101, prove that

```
(o ((curry map) f) ((curry map) g)) == ((curry map) (o f g))
```

To prove two functions equal, prove that when applied to equal arguments, they return equal results.

Take the following laws as given:

```
((o f g) x) == (f (g x))        ; apply-compose law
(((curry f) x) y) == (f x y)    ; apply-curried law
```

Using these laws should keep your proof relatively simple.

Related reading: Section 2.4.5. The definitions of composition and currying in section 2.7.2. Example uses of `map` in section 2.8.1. The definition of `map` in section 2.8.3.

Ordered lists

O. *Ordered lists*. I said in class that in most cases, a function that consumes lists uses the obvious inductive structure on lists: a list either empty or is made with `cons`. Here is a problem that requires a more refined inductive structure.

Define a function `ordered-by?` that takes one argument—a comparison function that represents a transitive relation—and returns a predicate that tells if a list is totally ordered by that relation. Assuming the comparison function is called `precedes?`, here is an inductive definition of a list that is ordered by `precedes?`:

- The empty list is ordered by `precedes?`.
- A singleton list is ordered by `precedes?`.
- A list of the form `(cons x (cons y zs))` is ordered by `precedes?` if the following properties hold:
  - `x` is related to `y`, which is to say `(precedes? x y)`.
  - List `(cons y zs)` is totally ordered by `precedes?`.

Here are some examples. Note the parentheses surrounding the calls to `ordered-by?`.

```
-> ((ordered-by? <) '(1 2 3))
#t
-> ((ordered-by? <=) '(1 2 3))
#t
-> ((ordered-by? <) '(3 2 1))
#f
-> ((ordered-by? >=) '(3 2 1))
#t
-> ((ordered-by? >=) '(3 3 3))
#t
-> ((ordered-by? =) '(3 3 3))
#t
```

Hints:

- The structure of your function should be informed by the structure of the inductive definition of what it means for a list to be ordered by a relation.
- You will need `letrec`.
- We recommend that your submission include the following unit tests, which help ensure that your function has the correct name and takes the expected number of parameters.

```
(check-expect (procedure? ordered-by?) #t)
(check-expect (procedure? (ordered-by? <)) #t)
(check-error (ordered-by? < '(1 2 3)))
```

Related reading: Section 2.9, which starts on page 125. Especially the polymorphic sort in section 2.9.2—the `lt?` parameter to that function is an example of a transitive relation.

What and how to submit

You must submit four files:

- A `README` file containing
  - The names of the people with whom you collaborated
  - The numbers of the problems that you solved
  - A note identifying any extra-credit work you did
  - The number of hours you worked on the assignment
- A `cqs.hofs.txt` containing the reading-comprehension questions[3] with your answers edited in
- A PDF files `semantics.pdf` containing the solutions to Exercise **M**. If you already know LaTeX[4], by all means use it. Otherwise, write your solution by hand and scan it. Do check with someone else who can confirm that your work is legible—if we cannot read your work, we cannot grade it.
- A file `solution.scm` containing the solutions to Exercises **14 (b–f,h,j)**, **15**, **19**, **A**, **G1**, **G2**, **G3**, and **O**. You must precede each solution by a comment that looks like something like this:

```
;;
;; Problem A
;;
```

As soon as you have the files listed above, run `submit105-hofs` to submit a preliminary version of your work. Keep submitting until your work is complete; we grade only the last submission.

Avoid common mistakes

Listed below are some common mistakes, which we encourage you to avoid.

Passing unnecessary parameters. In this assignment, a very common mistake is to pass unnecessary parameters to a nested helper function. Here's a silly example:

```
(define sum-upto (n)
  (letrec ([sigma (lambda (m n) ;;; UGLY CODE
                    (if (> m n) 0 (+ m (sigma (+ m 1) n))))])
    (sigma 1 n)))
```

The problem here is that **the `n` parameter to `sigma` never changes**, and it is already available in the environment. To eliminate this kind of problem, don't pass the parameter:

```
(define sum-upto (n)
  (letrec ([sum-from (lambda (m) ;;; BETTER CODE
                    (if (> m n) 0 (+ m (sum-from (+ m 1)))))])
    (sum-from 1)))
```

I've changed the name of the internal function, but the only other things that are different is that I have removed the formal parameter from the `lambda` and I have removed the second actual parameter from the call sites. I can still use `n` in the body of `sum-from`; it's visible from the definition.

[3] ./cqs.hofs.txt
[4] http://www.latex-project.org/

An especially good place to avoid this mistake is in your definition of `ordered-by?` in problem **O**.

Another common mistake is to fail to redefine functions `length` and so on in Exercise 15. Yes, we really want you to provide new definitions that replace the existing functions, just as the exercise says.

How your work will be evaluated

Structure and organization

The criteria in the general coding rubric[5] apply. As always, we emphasize **contracts** and **naming**. In particular, unless the contract is obvious from the name and from the names of the parameters, **an inner function defined with `lambda` and a `let` form needs a contract**.

There are a few new criteria related to `let`, `lambda`, and the use of basis functions. The short version is **use the functions in the initial basis; don't redefine them.**

	Exemplary	**Satisfactory**	**Must Improve**
Structure	• Short problems are solved using simple anonymous `lambda` expressions, not named helper functions. • When possible, inner functions use the parameters and `let`-bound names of outer functions directly. • The initial basis of µScheme is used effectively.	• Most short problems are solved using anonymous lambdas, but there are some named helper functions. • An inner function is passed, as a parameter, the value of a parameter or `let`-bound variable of an outer function, which it could have accessed directly. • Functions in the initial basis, when used, are used correctly.	• Most short problems are solved using named helper functions; there aren't enough anonymous `lambda` expressions. • Functions in the initial basis are redefined in the submission.

Functional correctness

In addition to the usual testing, we'll evaluate the correctness of your translation in problem **A**. We'll also want appropriate list operations to take constant time.

[5] ../coding-rubric.html

	Exemplary	**Satisfactory**	**Must Improve**
Correctness	● The translation in problem A is correct. ● Your code passes every one of our stringent tests. ● Testing shows that your code is of high quality in all respects.	● The translation in problem A is almost correct, but an easily identifiable part is missing. ● Testing reveals that your code demonstrates quality and significant learning, but some significant parts of the specification may have been overlooked or implemented incorrectly.	● The translation in problem A is obviously incorrect, ● Or course staff cannot understand the translation in problem A. ● Testing suggests evidence of effort, but the performance of your code under test falls short of what we believe is needed to foster success. ● Testing reveals your work to be substantially incomplete, or shows serious deficiencies in meeting the problem specifications (**serious fault**). ● Code cannot be tested because of loading errors, or no solutions were submitted (**No Credit**).
Performance	● Empty lists are distinguished from non-empty lists in constant time.		● Distinguishing an empty list from a non-empty list might take longer than constant time.

Proofs and inference rules

For your calculational proof, **use induction correctly** and **exploit the laws that are proved in the book**.

	Exemplary	**Satisfactory**	**Must Improve**
Proofs	● Proofs that involve predefined functions appeal to their definitions or to laws that are proved in the book. ● Proofs that involve inductively defined structures, including lists and S-expressions, use structural induction exactly where needed.	● Proofs involve predefined functions but do not appeal to their definitions or to laws that are proved in the book. ● Proofs that involve inductively defined structures, including lists and S-expressions, use structural induction, even if it may not always be needed.	● A proof that involves an inductively defined structure, like a list or an S-expression, does **not** use structural induction, but structural induction is needed.