Higher-Order Functions

COMP 105 Assignment

Due Tuesday, February 19, 2019 at 11:59PM

Contents

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

Overview

Higher-order functions are a cornerstone of functional programming. And they have migrated into all of the web/scripting languages, including JavaScript, Python, Perl, and Lua. This assignment will help you incorporate first-class and higher-order functions 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 *Build, Prove, and Compare*.

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
```

We don't give you a template—by this time, you know how to identify solutions and where to put contracts, algebraic laws, and tests.

Dire Warnings

The μScheme programs you submit must not use any imperative features. **Banish `set`, `while`, `print`, `println`, `printu`, 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 `println` while debugging, but they must not appear in any code you submit. As a substitute for assignment, use `let` or `let*`.

Except for the implementation of `length` in exercise 15, no code may compute the length of a list.

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. (An example of what to avoid appears under "Avoid common mistakes" below.)

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.

Every function should be accompanied by a short contract and by unit tests. If the function does case analysis, it must also be accompanied by algebraic laws. Submissions without algebraic laws will earn **No Credit**.

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)

Answer these questions before starting the rest of the assignment. 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, which you will use a lot. Suppose you need a list, or a Boolean, or a function—what can you call?

   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
   ```

   Put each function into exactly one of the following four categories:

   (B) Always returns a Boolean
   (F) Always returns a function
   (L) Always returns a list
   (A) Can return anything (including a Boolean, a function, or a list)

   After each function, write (B), (F), (L), or (A):

   ```
   map

   filter

   exists?

   all?

   curry

   uncurry

   foldl

   foldr
   ```

2. Here are the same functions again:

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

   For each function, say which of the following five categories best describes it. Pick the most specific category (e.g., (S) is more specific than (L) or (M), and all of these are more specific than (?)).

   (S) Takes a list & a function; returns a list of *exactly* the same size
   (L) Takes a list & a function; returns a list of *at least* the same size
   (M) Takes a list & a function; returns a list of *at most* the same size
   (?) Might return a list
   (V) Never returns a list

   After each function, write (S), (L), (M), (?), or (V):

[1] ./cqs.hofs.txt

`map`

`filter`

`exists?`

`all?`

`curry`

`uncurry`

`foldl`

`foldr`

3. Here are the same functions again:

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

   Put each function into exactly one of the following categories. Always pick the most specific category (e.g. (F2) is more specific than (F)).

   (F) Takes a single argument: a function
   (F2) Takes a single argument: a function *that itself takes two arguments*
   (+) Takes more than one argument

   After each function, write (F), (F2), or (+):

   `map`

   `filter`

   `exists?`

   `all?`

   `curry`

   `uncurry`

   `foldl`

   `foldr`

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

4. Review the difference between `foldr` and `foldl` in section 2.8.1. You may also find it helpful to look at their implementations in section 2.8.3, which starts on page 133; the implementations are at the end.

   (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, which is summarized on 159. 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.

5. Read the third *Lesson in Program Design*[2]: Higher-Order Functions. The lesson mentions a higher-order function `flip`, which can convert `<` into `>`, among other tricks. Write as many algebraic laws as are needed to specify `flip`:

6. Review function composition and currying, as described in section 2.7.2, which starts on page 128. Then judge the *proposed* properties below, which propose equality of functions, according to these rules:

   - Assume that names `curry`, `o`, `<`, `*`, `cons`, `even?`, and `odd?` have the definitions you would expect, but that `m` may have any value.
   - Each property proposes to equate two functions. If the functions are equal—which is to say, when both sides are applied to an argument, they always produce the same result—then mark the property **Good**. But if there is any argument on which the left-hand side produces *different* results from the right, mark the property **Bad**.

   Mark each property **Good** or **Bad**:

```
((curry <) m)     == (lambda (n) (< m n))

((curry <) m)     == (lambda (n) (< n m))

((curry cons) 10) == (lambda (xs) (cons 10 xs))

(o odd?  (lambda (n) (* 3 n))) == odd?

(o even? (lambda (n) (* 4 n))) == even?
```

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

Programming and Proof (90 percent)

Overview

For this assignment, you will do Exercises **14 (b-f,h,j)**, **15**, and **19**, from pages 212 to 216 of *Build, Prove, and Compare*, plus the exercises **A**, **F**, **V1** to **V6**, **M**, and **O** below.

[2] ../design/lessons.pdf

A summary of the initial basis can be found on page 159. While you're working on this homework, *keep it handy*.

Each top-level function you define must be accompanied by a contract and unit tests. Each *named* internal function written with `lambda` should be accompanied by a contract, but internal functions cannot be unit-tested. (Anonymous `lambda` functions need not have contracts.) Algebraic laws are required only where noted below; each problem is accompanied by a **Laws** section, which says what is needed in the way of algebraic laws.

Book problems

14. *Higher-order functions*. Do exercise 14 on page 212 of *Build, Prove, and Compare*, parts (b) to (f), part (h), and part (j). Note which functions accept only *nonempty* lists, and code accordingly. **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 defined in the initial basis, may use recursion.)

Because you are not defining recursive functions, you need not write any algebraic laws.

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 131. For material on `curry`, see section 2.7.2, which starts on page 128.

Laws: These functions must not be recursive, should not do any case analysis,[3] and do not return functions. Therefore, no algebraic laws are needed.

15. *Higher-order functions*. Do exercise 15 on page 214. **You must *not* use recursion—solutions using recursion will receive No Credit.** As above, this restriction applies only to code you write.

Because you are not defining recursive functions, you need not write any algebraic laws.

For this problem, you get full credit if your implementations return correct results. You get *extra* credit[4] 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` are in sections 2.8.1 and 2.8.2 starting on page 131. You may also find it helpful to study the implementations of `foldl` and `foldr` in section 2.8.3, which starts on page 133; the implementations are at the end. Information on `lambda` can be found in section 2.7, on pages 121 to 124.

Laws: These functions must not be recursive, should not begin with case analysis, and do not return functions. Therefore, no algebraic laws are needed.

19. *Functions as values*. Do exercise 19 on page 216 of *Build, Prove, and Compare*. **You cannot represent these sets using lists.** If any part of your code to construct or to interrogate a set uses `cons`, `car`, `cdr`, or `null?`, you are doing the problem wrong.

Do all four parts:

[3] Case analysis may be happening, but on this problem, it will be happening inside functions like `map` and `foldr`, not in any code that you write.

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

- 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.

  When you code part (c), compare values for equality using the `equal?` function.

  To help you design part (c), put comments in your source code that complete the right-hand sides of the following *properties*:

  ```
  (member? x (add-element x s)) == ...
  (member? x (add-element y s)) == ..., where (not (equal? y x))
  (member? x (union s1 s2))     == ...
  (member? x (inter s1 s2))     == ...
  (member? x (diff  s1 s2))     == ...
  ```

  The properties are not quite algorithmic, but they should help anyway.
- 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. To define record functions, use the syntactic sugar described in the book in Section 2.16.6 on page 194. In particular, be sure your code includes 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:

  ```
  (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 own implementations. Each implementation is like one you wrote in part (c), except instead of using the predefined `equal?`, it uses the parameter `eq?`—that is what is meant by "the third approach to polymorphism."

  No additional laws are needed for part (d).

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

```
(check-assert (procedure? set-ops-from))
(check-assert (set-ops? (set-ops-from =)))
```

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 atom-emptyset      (set-ops-empty atom-set-ops))
(val atom-member?      (set-ops-member? atom-set-ops))
(val atom-add-element  (set-ops-add-element atom-set-ops))
```

```
(val atom-union        (set-ops-union atom-set-ops))
(val atom-inter        (set-ops-inter atom-set-ops))
(val atom-diff         (set-ops-diff atom-set-ops))
```

Hint: The recitation for this unit includes an "arrays as functions" exercise. Revisit it.

Related reading: For functions as values, see the examples of `lambda` in the first part of section 2.7 on page 121, and also the array exercise from recitation. For function composition and currying, see section 2.7.2. For polymorphism, see section 2.9, which starts on page 135.

Laws: Complete the right-hand sides of the properties listed above. These properties say what happens when `member?` is applied to any set created with any of the other functions. No other laws are needed.

Relating imperative code to functional code

A. *Good functional style*. The Impcore-with-locals function

```
(define f-imperative (y) [locals x]
  (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.
- You need not write any algebraic laws.
- You need not write any unit tests. (And we recommend against trying to unit-test this function.)

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.

Laws: This problem doesn't need laws.

A function that returns a function

F. The third lesson in program design[5] ("Higher-order functions") mentions a higher-order function `flip`, which can convert `<` into `>`, among other tricks. Using your algebraic law or laws from the comprehension questions, define `flip`. Don't forget unit tests.

Related reading: *Seven Lessons in Program Design*[6], lesson 3.

Laws: Use your law or laws from the comprehension questions.

Input validation

In the following set of problems, you use higher-order functions to create a fault detector for web forms. To see what such a fault detector does, go to the course regrade form at https://www.cs.tufts.edu/comp/105/regrade, and click the Submit button without filling in the form. Now ask for a grade review without giving a problem or assignment, and Submit again. How does the software know the response to the form is faulty?

The regrade form uses higher-order functions to detect fields that have not been filled out and radio buttons that have not been checked. The browser's response to the web form is represented as an association list, and the fault detector, which uses functions you will implement, looks something like this:

```
(val regrade-analyzer  ;; as you read the problem, refer back to me
  (faults/switch 'why
    (bind         'photo
                  faults/none
      (bind       'badsubmit
                   (faults/both (faults/equal 'badsubmit_asst '...)
                                (faults/equal 'info #f))
        (bind     'badgrade
                  (faults/both
                      (faults/equal 'badgrade_asst '...)
                      (faults/both
                         (faults/equal 'info #f)
                         (faults/equal 'problem #f)))
            (bind '#f
                  (faults/always 'nobutton)
                  '()))))))
```

You can download some integration tests[7] for this analyzer.

Before we see the specifications of the functions, here is how we represent the input-validation problem in μScheme:

- A *response* to a web form is represented by an association list: each field in the form has a name, which is represented by a symbol, and in the response, that name is bound to the value the user responded with. If the user did not supply a value, the name is bound to the value `#f`.

  Here's an example response, representing a student who wants something regraded because they accidentally submitted the wrong PDF:

[5] ../design/lessons.pdf
[6] ../design/lessons.pdf
[7] ./analyzer-tests.scm

```
(val sample-response
  '([why badsubmit]
    [badsubmit_asst scheme]
    [info (I accidentally submitted the opsem PDF again.)]
    [badgrade_asst ...]
    [problem #f]))
```

- A *fault* is represented by a symbol. The symbol is typically the name of the field that was filled out incorrectly, or that was not filled out, but *any* symbol can be a fault.

  Here are some of the faults used by the regrade analyzer:

  - `nobutton` if no radiobutton was selected
  - `problem` if "review my grade" was selected but the "problem" field was blank
  - `badsubmit_asst` if "I submitted the wrong PDF" was selected but the "assignment" dropdown was left at `...`

- A *fault set* is represented by a **list** of symbols without duplicates.[8]

  For example, if I select "review my grade" on the regrade form, but I don't fill out anything else, the analyzer will return a fault set containing symbols `problem`, `badgrade_asst`, and `info`, like this one:

  ```
  '(problem badgrade_asst info)
  ```

- An *analyzer* is a function that takes one argument (a response) and returns a fault set.

In the problems below, you create a "little language" for writing analyzers.[9] The key elements of this language are higher-order functions: functions that build analyzers from other analyzers.

Expectations for these functions:

- Each function you write must be accompanied by algebraic laws and unit tests.
- Every analyzer must treat the response as an abstraction. An analyzer must never interrogate a response about its form of data; the analyzer should restrict itself to function `find` and possibly function `bound-in?`:

  ```
  (define bound-in? (key pairs)
    (if (null? pairs)
        #f
        (|| (= key (alist-first-key pairs))
            (bound-in? key (cdr pairs)))))
  ```

Problem **V** is to *implement the five functions* described in **V1** to **V5** below, plus the "travel validator" described in **V6**. My five functions total less than 20 lines of μScheme.

Related reading:

- Except for `faults/none`, the functions below are "function factories": each one takes in some arguments and returns a function. The main function factories in the book are `curry` and `o` ("compose"); to review them, study section 2.7.2, which starts on page 128.

[8] Unlike exercise 19, the fault set here is represented by a *list* of values, not by a function.

[9] It's actually just a library, but a library like this is called a "little language" because of the way functions compose. The composition of fault analyzers resembles the syntactic composition of expressions in an actual programming language.

- For `faults/both`, revisit the `combine` and `divvy` examples from the lecture on `lambda`. (These functions are intended to be called `conjoin` and `disjoin`.)
- For a review of association lists and `find`, consult section 2.3.8, which starts on page 106.

Laws. Write laws for each of the five functions **V1** to **V5** below. Most of your algebraic laws, like the algebraic laws for `curry` and `flip`, will need to specify what happens when the result of calling a function is itself applied. If written well, your laws will be clearer and easier to follow than the informal text below. Which is one reason why we write them.

For the travel validator in **V6**, laws are neither necessary nor useful.

V1. Function `faults/none` is an analyzer that always returns the empty list of faults, no matter what the response.

V2. Function `faults/always` takes one argument (a fault F), and it returns an analyzer that finds fault with every response. No matter what the response, the analyzer returns a singleton list containing the fault F.

V3. Function `faults/equal` takes two arguments, a key k and a value v, and it returns an analyzer that finds fault if the response binds k to v. That is, when given a response R, the analyzer returns an empty set of faults *unless* key k is bound to value v in R. If key k *is* bound to value v, the analyzer returns a singleton list containing the fault k.

V4. Function `faults/both` takes two analyzers `a1` and `a2` as arguments. It returns an analyzer that, when applied to a response, finds all the faults found by analyzer `a1` and also all the faults found by analyzer `a2`, returning them together in a single set.

To avoid conflicts with the set code in exercise 19, you are welcome to use the following template to define `faults/both`:

```
(val faults/both
   (let* ([member?  (lambda (x s) (exists? ((curry =) x) s))]
          [add-elem (lambda (x s) (if (member? x s) s (cons x s)))]
          [union    (lambda (faults1 faults2) (foldr add-elem faults2 faults1))])
      ...))
```

V5. Function `faults/switch` takes two arguments, a key k and an *analyzer table*, and it returns an analyzer that uses field k of the response to analyze the response. The analyzer table is like the list of cases in a C `switch` statement: each case is labeled with a value, and the action to be performed in the case is itself an analyzer. The analyzer table is represented as an association list.

The use case for `faults/switch` is to create an analyzer that uses some of the input data to figure out how to analyze the rest. For example, in the regrade-form validator, if a student submits a photo for regrade, `faults/switch` figures out that no more analysis required. But if a student asks for a given problem to be regraded, `faults/switch` figures out that we need an assignment, a problem number, and an explanation ("info").

Here's the contract of `faults/switch`: key k determines which field of the response is used to make a decision. When the analyzer is given a response, the value v associated with key k *in the response* is looked up in the analyzer table, and the resulting analyzer is used to find faults in the response. If I had a language like Lua, Python, or JavaScript, where table lookup is primitive, I might write simply

```
analyzer_table[response[k]](response)
```

When you write your algebraic laws, you will notate a similar computation in μScheme.

This problem is simplest if key k is bound in the response and value v is bound in the analyzer table. When you write your algebraic laws, make this simplifying assumption. And when you write your code, you may simply assume that k is bound in the response. But if v is *unbound* in the analyzer table, the analyzer must halt with a checked run-time error. (Any error will do.)

V6. *Travel validator*. In this problem, you use some of the functions above to define a validator for a travel form. A response to the travel form has three fields:

- Field `type` is either `'one-way`, `'round-trip`, or `#f`.
- Field `out_date` is either `#f` (if empty) or a symbol (if not empty).
- Field `return_date` is either `#f` (if empty) or a symbol (if not empty).

Your validator must detect the following faults:

- If field `type` is `#f`, then field `type` is at fault.
- if field `type` is `round-trip` and field `out_date` is `#f`, field `out_date` is at fault.
- if field `type` is `round-trip` and field `return_date` is `#f`, field `return_date` is at fault.
- if field `type` is `one-way` and field `out_date` is `#f`, field `out_date` is at fault.

(A one-way trip with a return date is also faulty, but that fault can't easily be detected with the functions you have. You're not required to detect it.)

Using the `val` definition form, *define an analyzer* `travel-validator`, which implements the fault-detection rules above. Your analyzer should be implemented using only the functions above.

Here is a test case:

```
(check-expect
   (travel-validator '([type round-trip]
                       [out_date 4/11/2019]
                       [return_date #f]))
   '(return_date))
```

Calculational reasoning about functions

M. *Reasoning about higher-order functions*. Using the calculational techniques from Section 2.4.5, which starts on page 110, 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.

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.

Laws: In this problem you don't write new laws; you reuse existing ones. You may use any law in the Basic Laws[10] handout, which includes laws for `o`, `curry`, and `map`. (If it simplifies your proof, you may also introduce new laws, provided that you prove each new law is valid.)

[10] ../handouts/initial-laws.html

Ordered lists

O. *Ordered lists*. Like natural numbers, the forms of a list can be viewed in different ways. In almost all functions, we examine just two ways a list can be formed: `'()` and `cons`. But in some functions, we need a more refined view. Here is a problem that requires us to divide a list of values into *three* forms.

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 of values 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 of values is ordered by `precedes?`.
- A singleton list containing one value is ordered by `precedes?`.
- A list of values in 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 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 entire 9-step software-design process applies, and it starts with the *three* forms of data in the definition of "list ordered by" above.
- For the code itself, 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-assert (procedure? ordered-by?))
(check-assert (procedure? (ordered-by? <)))
(check-error (ordered-by? < '(1 2 3)))
```

Related reading: Section 2.9, which starts on page 135. Especially the polymorphic sort in section 2.9.2—the `lt?` parameter to that function is an example of a transitive relation. Section 2.7.2. Example uses of `map` in section 2.8.1. The definition of `map` in section 2.8.3.

Laws: Write algebraic laws for `ordered-by?`, including at least one law for each of the *three* forms of data used in the definition of "list ordered by" above.

Extra credit

VX. For extra credit, answer the questions below.

(a) Which of the following equations are valid properties of the fault-validation functions?

- `(faults/both faults/none` A`)` $\equiv$ A?
- `(faults/both (faults/always` F`)` A`)` $\equiv$ `(faults/always` F`)`?

(b) For each of the equations in part (a),

- If the equation is a valid property, present a calculational proof using your laws from problems **V1**, **V2**, and **V4**.
- If the equation is not a valid property, present a counterexample. That is, present examples of A, F (if necessary), and a response such that, when applied to the response, the two analyzers produce different fault sets.

What and how to submit

You must submit four files:

- A `README` file containing
  - The names of the people with whom you collaborated
  - A list identifying which problems you solved
  - A note identifying any extra-credit work you did
- A `cqs.hofs.txt` containing the reading-comprehension questions[11] with your answers edited in
- A PDF files `semantics.pdf` containing the solutions to Exercises **M** and **VX**. (Exercise VX is extra credit and is optional.) If you already know LaTeX, 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**, **F**, **V1** to **V6**, 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.

[11] ./cqs.hofs.txt

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.

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 predefined functions like `map` and `filter` 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[12] 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**. (An anonymous lambda that is returned from a function like `faults/both` does not need a contract—the behavior of that lambda is part of the contract of the function that returns it.)

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; except when we specifically ask you to, don't redefine initial-basis functions.**

[12] ../coding-rubric.html

	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.

	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.