Functional programming in μScheme

COMP 105 Assignment

Due Tuesday, February 12, 2019 at 11:59PM

Contents

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

Overview

This assignment develops new skills that you can use to write one kind of code from scratch: code that inspects and manipulates lists, trees, or other linked data structures. You already know how to manipulate these structures using machine-level abstractions that operate on one word and one pointer at a time. You will start to develop a flexible, powerful vocabulary of functions that enable you to manipulate a whole list in just one or two operations. These skills come from the discipline of *functional programming*.

The key thing that's new this week is that no data structure is ever mutated—instead of changing an *existing* list or tree, code allocates a *new* list or tree with the desired values and structure. This discipline of programming has benefits for testing, specification, and coding:

- Tests are easy to write, require no setup, and can be repeated without fear of failure.
- Contracts are written without having to refer to multiple states of execution: a function's contract mentions only the inputs and the result.
- Because unchanging data structures can safely be shared, the functions in your vocabulary can easily be composed.

You will learn more about composition of functions next week.

In addition to programming, you will get more practice with programming-language theory and proofs. You will see that algebraic laws are built on top of operational semantics, and you will learn that on top of algebraic laws, we can build *calculational proofs* of program properties. This "cheap and cheerful" way of assuring program correctness is another benefit of functional programming.

This week's assignment is based primarily on material from sections 2.1 to 2.6 of *Programming Languages: Build, Prove, and Compare*. You will also need to know the syntax in section 2.11, which starts on page 147, and the initial basis (also in section 2.11). The table on page 159 lists all the functions found in the basis—**it is your lifeline**. Finally, although it is not necessary, you may find some problems easier to solve if you read ahead from section 2.7 to section 2.9.

You will define many functions and write a few proofs. The functions are small; most are in the range of 4 to 8 lines, and none of my solutions is more than a dozen lines. If you don't read ahead, a couple of your functions will be a bit longer, which is OK.

Setup

The executable μScheme interpreter is in `/comp/105/bin/uscheme`; once you have run `use comp105` (or set it up to run automatically on login), you should be able to run `uscheme` as a command. The interpreter accepts a `-q` ("quiet") option, which turns off prompting. When using the interpreter interactively, you may find it helpful to use `ledit`, as in the command

```
ledit uscheme
```

Please also download our template for solution.scm[1]. It contains a skeleton version of each function you

[1] http://www.cs.tufts.edu/comp/105/homework/solution-template.scm

must define, but the body of the function calls `error`. Each call to `error` should be replaced with a correct implementation.

Diagnostic tracing

μScheme does not ship with a debugger. But in addition to the `println` and `printu` functions, it does ship with a tracing facility. The tracing facility can show you the argument and results to every function call, or you can dial it back to show just a limited number.

The tracing facility is described in exercise 74 on page 237 of *Build, Prove, and Compare*. Our facility takes the approach sketched in part (b). Here are a couple of example calls for you to try:

```
-> (val &trace 5)
-> (append '(a b c) '(1 2 3))
-> (set &trace 500)
-> (append '(a b c) '(1 2 3))
```

Used carefully, `&trace` can save you a lot of time and effort. **But do not leave even an unexecuted reference to `&trace` in your submission.**

Dire Warnings

Since we are studying functional programming, the μScheme programs you submit must not use any imperative features. **Banish `set`, `while`, `println`, `print`, `printu`, and `begin` from your vocabulary! If you break this rule for any problem, you will get No Credit for that problem.** 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*`.

Helper functions may be defined at top level *only* if they meet these criteria:

- Each helper function has a meaningful name[2].
- Each helper function is given an explicit contract—or, as described in the general coding rubric[3], we can infer the contract by looking at the names of the function and its formal parameters.
- Each helper function is specified by algebraic laws.
- Each helper function is tested by `check-expect` or `check-assert`, and possibly `check-error`.

As an alternative to helper functions, you may read ahead and define local functions using `lambda` along with `let`, `letrec`, or `let*`. If you do define local functions, avoid passing them redundant parameters—a local function already has access to the parameters and let-bound variables of its enclosing function.

Except as specified, functions without algebraic laws will earn failing grades.

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

```
/comp/105/bin/uscheme -q < myfilename
```

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

[2] ../coding-rubric.html
[3] ../coding-rubric.html

and organization). If your file includes failing unit tests, you might possibly get some credit for functional correctness, but we cannot guarantee it.

Case analysis involving lists and S-expressions must be structural. That is, your case analysis must involve the results of functions like `null?`, `atom?`, `pair?`, and so on, all of which are found in the initial basis. Please note that the `length` function from the book is *not* in the initial basis, and **code submitted for this assignment must not compute the length of any list.**

Code you submit must not even *mention* `&trace`. We recommend that you use `&trace` only at the interactive prompt.

We will evaluate functional correctness by automated testing. **Because testing is automated, each function must be named be exactly as described in each question. Misnamed functions earn No Credit.** You may wish to use the template[4] provided above, which has the correct function names.

Reading Comprehension (10 percent)

These problems will help guide you through the reading. Complete them before starting the other problems below. You can download the questions[5].

1. Read Sections 2.1 and 2.2 (the first part of the second lesson) in *Seven Lessons in Program Design*[6].

   You are tasked with writing a function that consumes a list of numbers:

   (a) How many cases must you consider?

   (b) To tell the cases apart, what condition or conditions will you use in `if` expressions? (List one fewer condition than cases.)

   You are tasked with writing a function that consumes an ordinary S-expression.

   (c) How many cases must you consider?

   (d) To tell the cases apart, what condition or conditions will you use in `if` expressions? (List one fewer condition than cases.)

   You are ready to write algebraic laws using Scheme data.

2. In the main textbook, review section 2.2 on values, S-expressions, and primitives, and say what is the value of each of the expressions below. If a run-time error would occur, please say so.

   ```
   (car '(a b 1 2))
   (cdr '(a b 1 2))
   (= 'a 'b)
   ```

   Write your answers as S-expression literals, like `'(a b c)`, `#t`, or `17`.

   *You are on your way to being ready for exercise **F**.*

[4] http://www.cs.tufts.edu/comp/105/homework/solution-template.scm
[5] ./cqs.scheme.txt
[6] ../design/lessons.pdf

3. In *Programming Languages: Build, Prove, and Compare*, review the first few pages of section 2.3, through the end of section 2.3.2, and also section 2.3.5, which starts on page 103. Which of the following expressions evaluates to `#t` for every *list of ordinary S-expressions* `xs`?

```
(=      (reverse (reverse xs)) xs)
(equal? (reverse (reverse xs)) xs)
```

   (a) Only the first
   (b) Only the second
   (c) Both the first and the second
   (d) None

4. Read about association lists in section 2.3.8, which starts on page 106. Given the definition

```
(val mascots
   '((Tufts Jumbo) (MIT Beaver) (Northeastern Husky) (BU Terrier)))
```

   Say what is the value of each of these expressions:

```
(find 'Tufts   mascots)
(find 'MIT     mascots)
(find 'Harvard mascots)
(find 'MIT (bind 'MIT 'Engineer mascots))
```

5. Read section 2.3 (another part of the second lesson) in *Seven Lessons in Program Design*[7], and also the first part of section 2.4 in the main textbook, up to and including section 2.4.4.

   Now complete the following law, which should represent a true *property* of the association-list functions `find` and `bind`:

```
(find x (bind ...)) = ...
```

   You may use variables, and you may use forms of data made with `'()` and with `cons`. You may not use any atomic literals. Write your property in the style of section 2.4.4.

   *You are now prepared for the algebraic laws in exercises **A**, **B**, and **C**.*

6. In *Programming Languages: Build, Prove, and Compare*, read the two laws for `append` (which we will call "append-nil" and "append-cons") on page 99, and then study the proof at the bottom of page 111, which shows that `(append (cons x '()) ys)` equals `(cons x ys)`.

   Now answer this question: The proof on page 111 proceeds by expanding the definition of `append`. Suppose that you simplify the proof by instead applying the "append-cons" law to the very first expression. How many steps in the original proof does this one step replace? (Count one step for each = sign.)

   Your answer:

7. Read section 2.5, which explains `let`, `let*`, and `letrec`. This question asks you to decide if any or all these forms can appropriately express the following function (written in C):

```
bool parity(int m) {
  int half_m     = m / 2;
  int other_half = m - half_m;
```

[7] ../design/lessons.pdf

```
  return half_m == other_half;
}
```

Scheme does not have local variables, so to translate this function into μScheme, you must use `let`, `let*`, or `letrec`. For each of these syntactic forms, we ask you if a translation sensibly and faithfully captures the intent and behavior of the original C function.

```
;; Translation A
(define parity (m)
   (let ([half_m     (/ m 2)]
         [other_half (- m half_m)])
     (= half_m other_half)))
```

Is translation A sensible and faithful (yes or no)?

```
;; Translation B
(define parity (m)
   (let* ([half_m     (/ m 2)]
          [other_half (- m half_m)])
     (= half_m other_half)))
```

Is translation B sensible and faithful (yes or no)?

```
;; Translation C
(define parity (m)
   (letrec ([half_m     (/ m 2)]
            [other_half (- m half_m)])
     (= half_m other_half)))
```

Is translation C sensible and faithful (yes or no)?

You are now ready to program using `let`, `let*`, and `letrec`.

8. Read section 2.16.6, which starts on page 194. Imagine that μScheme is given the following definition:

   ```
   (record 3point (x y z))
   ```

   This definition puts five functions into the environment ρ. What are their names?

   *You are now mostly ready for exercise **E**.*

9. Read section 2.3 in the second *Lesson in Program Design*[8]—in particular, the last part, on understanding and using properties. Assuming that `x` is different from `y`, complete the following property:

   ```
   (member? x (add-element y xs)) == ...,
                                     where x differs from y
   ```

   You are ready to use properties to test `split-list`.

[8] ../design/lessons.pdf

Programming and Proof Problems (90 percent)

For the "programming and proof" part of this assignment, you will do exercises **1**, **2**, **10**, and **37** in the book, plus the problems **A** through **H**, **LP**, and **N** below—but not in that order. There are also two extra-credit problems: problems **M** and **TDP**.

Problem Details (Theory)

1. *A list of S-expressions is an S-expression.* Do exercise 1 on page 207 of *Build, Prove, and Compare*. Do this proof before tackling exercise 2; the proof should give you ideas about how to implement the code.

Related Reading: The definitions of *LIST (A)* and $SEXP_{FG}$ are on page 116.

37. *Calculational proof.* Do exercise 37 on page 221 of *Build, Prove, and Compare*, proving that appending lists is an associative operation.

This problem yields to structural induction, but there are three lists involved. The hard part is to identify which list or lists have to be broken down by cases and handled inductively, and which ones can be treated as variables and not scrutinized. *Hint:* it is not necessary to break down all three lists.

Related Reading:

- The proof technique is described in section 2.4.5, which starts on page 110.
- Section 2.3.1, which starts on page 98, develops `append`, and it states these two laws:

  ```
  (append '()         ys) == ys
  (append (cons z zs) ys) == (cons z (append zs ys))
  ```

  You will find additional laws for append on page 110, but you may not use those additional laws—in particular, the third law is what you are trying to prove.
- We have summarized basic laws of μScheme[9] in a web page[10].

A. *From operational semantics to algebraic laws.* This problem has two parts:

a) The operational semantics for μScheme includes rules for `cons`, `car`, and `cdr`. Assuming that `x` and `xs` are variables and are defined in ρ (rho), use the operational semantics to prove that

   ```
   (cdr (cons x xs)) == xs
   ```

b) The preceding law applies only to variables `x` and `xs`. In this part, you determine if a similar law applies to expressions.

   Use the operational semantics to prove or disprove the following conjecture: if e_1 and e_2 are arbitrary expressions, in any context where the evaluation of e_1 terminates and the evaluation of e_2 terminates, then both of the following are true:

   - The evaluation of `(cdr (cons` e_1 e_2`))` terminates, and
   - `(cdr (cons` e_1 e_2`))` == e_2

   The conjecture says that **two independent evaluations**, starting from the **same initial state**, produce the same *value* as a result.

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

If you believe the conjecture, you can establish it by proving that it's true for *every* choice of e_1 and e_2, in any context in which both e_1 and e_2 terminate. If you disbelieve the conjecture, you need only to find *one* choice of e_1, e_2, and context such that both e_1 and e_2 terminate but at least one of the desired conclusions does not hold.

Related Reading: The operational semantics for `cons`, `car`, and `cdr` can be found on page 157.

Our expectations for your code: Algebraic laws and unit tests

For each function you define, you must specify not only a contract but also algebraic laws and unit tests. Even helper functions! For some problems, algebraic laws are not needed or are already given to you. Those problems are noted below.

Laws and tests make it easy to write code and easy for readers to be confident that code is correct. To get your laws, code, and tests right, use the checklists below.

A checklist for your laws

A good set of algorithmic laws satisfies all these requirements:

- The left-hand sides break the inputs down by cases. In each case, each argument is a variable or is a form of data such as `(cons y ys)`. A good left-hand side never has a call to a non-primitive function like `list2` or `append`.
- Cases are mutually exclusive. Mutual exclusion is usually accomplished by using mutually exclusive forms of data on distinct left-hand sides, but occasionally, mutual exclusion may be accomplished via side conditions.
- You can tell which case is which via a constant-time test, like `(null? xs)` or `(= n 0)`.
- Each left-hand side is equal to some right-hand side, and the right-hand side can be computed as a function of the variables named on the left-hand side. Every variable that appears on a right-hand side also appears on the corresponding left-hand side.
- If a variable on the left-hand side stands for a *part* of an argument, then on the right-hand side that variable stands for the same part of the same argument—not the whole argument.
- No algebraic law is completely redundant. That is, no law is fully implied by a combination of other laws. (It is OK if some inputs are covered by more than one law, which we call "overlapping." Overlapping laws are handy, but you must be sure that on the overlapping inputs, all laws agree on the result.)
- If, given a particular input, the function's contract says that a value is returned, there must be some algebraic law that specifies what the value is.
- In every recursive call on every right-hand side, some input is getting smaller.

A checklist for your code

Your laws will be evaluated not just in isolation but in the context of your code. (The whole purpose of laws is to help write code.) In particular, your laws must be consistent with your code.

- The number of cases in your code is equal to the number of algebraic laws.

It is always possible to structure your code so it has one case per law. But it is acceptable to take shortcuts with things like short-circuit && and ||. It is also acceptable, if unusual, to use `if` on the right-hand side of an algebraic law, in which case that law would cover two cases in the code.

- The names of formal parameters are consistent with the names used in algebraic laws. If there is no case analysis on a parameter, its name is the same everywhere it appears. If there is case analysis, a parameter's name is different from the names of its parts. Example: parameter `xs` might take the form `(cons y ys)`.

A checklist for your tests

While it is often useful to write additional tests for corner cases, here is a checklist for our minimum expectations.

- Every algebraic law is tested.
- If the function returns a Boolean, each algebraic law is tested using `check-assert`. Otherwise, each algebraic law is tested using `check-expect`.
- If the function returns a Boolean, then when possible, each algebraic law is tested twice: once with a true result and once with a false result. (Such testing is not always possible; for example, the empty list is always a sublist of any other list, and it is not possible to test that case with a false result.)
- A function is tested using `check-error` *if and only if* the function's contract says that certain inputs cause a checked run-time error.

Problem Details (Code)

Related Reading: Many of the following problems ask you to write recursive functions on lists. You can sometimes emulate examples from section 2.3, which starts on page 98. And you will definitely want to take advantage of μScheme's predefined and primitive functions (the initial basis). These functions are listed in section 2.13, which starts on page 158.

Lists and S-expressions

In this set of problems, you work with lists, and you also work with ordinary S-expressions, which are "lists all the way down."

2. *Recursive functions on lists of S-expressions*. Do parts c, d, and e of exercise 2 on page 207 of *Build, Prove, and Compare* (`mirror`, `flatten`, and `contig-sublist?`). Expect to write some recursive functions, but you may also read ahead and use the higher-order functions in sections 2.7 through 2.9.

Each function you define, including helper functions, must be accompanied by algebraic laws and by unit tests written using `check-expect` or `check-assert`, with this exception:

- The algebraic laws for `contig-sublist?` may be too challenging for beginners, so you may omit them. But do write laws for all other functions, including helper functions.

Related reading:

- The rules for ordinary S-expressions are shown in Figure 2.1 on page 95.

- The first section of the second *Lesson in Program Design*[11] describes several ways to break down S-expressions. Look at the "expanded" version at the end of the section.

Hints:

- It is acceptable to extend the contracts of the *LIST(SEXP)* functions so that they can also accept an *SEXP*. For example, you might extend the contract of `mirror` so that if it receives an atom, it returns that atom. Extending a contract in this way can simplify code. But in some cases it may be unnecessary or even counterproductive.
- Once you extend a contract, you can profitably break *SEXP* down by three cases: empty list, `cons`, and atom different from empty list.
- The most difficult function here is probably `contig-sublist?`. Think how you would implement it in C++: probably with a doubly nested loop. In Scheme, therefore, you probably will implement it using two recursive functions: one corresponding to the outer loop and one corresponding to the inner loop. The additional function, like every function, will need a good name and contract.

Classic list functions

In this set of problems, you write some classic functions for manipulating whole lists, or for chopping lists into big pieces.

10. *Taking and dropping a prefix of a list* (`takewhile` and `dropwhile`). Do exercise 10 on page 212 of *Build, Prove, and Compare*.

Each function you define, including helper functions, must be accompanied by algebraic laws and by unit tests written using `check-expect` or `check-assert`.

B. *Take and drop*. Function `(take n xs)` expects a natural number and a list of values. It returns the longest prefix of `xs` that contains at most `n` elements.

Function `(drop n xs)` expects a natural number and a list of values. Roughly, it removes `n` elements from the front of the list. When acting together, `take` and `drop` have this property: for any list of values `xs` and natural number `n`,

```
(append (take n xs) (drop n xs)) == xs
```

Implement `take` and `drop`.

Each function you define, including helper functions, must be accompanied by algebraic laws and by unit tests written using `check-expect` or `check-assert`. Be aware that *the property above (the "append/take/drop" law) is not algorithmic*. Therefore, it cannot be used as the sole guide to implementations of `take` and `drop`. Before defining `take`, you must write laws that define only what `take` does. And before defining `drop`, you must write more laws that define only what `drop` does.

C. *Zip and unzip*. Function `zip` converts a pair of lists to a list of pairs by associating corresponding values in the two lists. (If `zip` is given lists of unequal length, its behavior is not specified.) Function `unzip` converts a list of pairs to a pair of lists. In both functions, a "pair" is represented by a list of length two, e.g., a list formed using predefined function `list2`.

```
-> (zip '(1 2 3) '(a b c))
((1 a) (2 b) (3 c))
```

[11] ../design/lessons.pdf

```
-> (unzip '((I Magnin) (U Thant) (E Coli)))
((I U E) (Magnin Thant Coli))
```

The standard use cases for `zip` and `unzip` involve association lists, but these functions are well defined even when keys are repeated:

```
-> (zip '(11 11 15) '(Guyer Sheldon Korman))
((11 Guyer) (11 Sheldon) (15 Korman))
```

As further specification, provided lists `xs` and `ys` are the same length, `zip` and `unzip` satisfy these properties:

```
(zip (car (unzip pairs)) (cadr (unzip pairs))) ==  pairs
(unzip (zip xs ys))                            ==  (list2 xs ys)
```

Neither of these properties is algorithmic. You are excused from writing algebraic laws for `unzip`, but you must write *algorithmic* laws for `zip`.

Implement `zip` and `unzip`.

Each function you define, including helper functions, must be accompanied by algebraic laws and by unit tests written using `check-expect` or `check-assert`, with this exception:

- The algebraic laws for `unzip` are too challenging for beginners, so you may omit them.

Related Reading: Information on association lists can be found in section 2.3.8, which starts on page 106.

Properties of lists

Programmers who are first working with lists are often tempted to ask questions about a list's length. But if you have a list of a million elements, computing the length is expensive, and in many situations, a property of even a very long list can be computed quickly (sometimes in constant time). The problem below asks you to write predicates that test properties about the length of a list—without ever computing a length.[12]

LP. *Length predicates*. Here are four predicates that give information about length, of which you will implement the last three:

- When `xs` is a list of values, `(null? xs)` tells if the length of `xs` is 0.

  (You don't implement `null?`; it is part of the initial basis.)

- When `xs` is a list of values, `(singleton? xs)` tells if the length of `xs` is 1.

  Implement `singleton?`, and make sure it runs in constant time.

- When `xs` is a list of values and `n` is a natural number, `(has-n-elements? xs n)` tells if the length of `xs` is `n`. That is, it tells if `xs` has *exactly* `n` elements.

  Implement `has-n-elements?`, and make sure its running time is proportional to the *smaller* of these two numbers: the length of `xs`, and the magnitude of the natural number `n`.

- When `xs` and `ys` are both lists of values, `(nearly-same-lengths? xs ys)` tells if the length of `xs` and the length of `ys` differ by at most 1.

[12] As noted above, on this assignment, any code that computes a length will earn **No Credit**.

Implement `nearly-same-lengths?`, and make sure its running time is proportional to length of the shorter list.

If the caller of any of these functions violates a contract, say by passing an non-list or a negative number, the function has no obligations, either toward running time or anything else.

Each function you define, including helper functions, must be accompanied by algebraic laws and by unit tests written using `check-expect` or `check-assert`.

Hints: All these functions yield to the standard method of breaking the inputs down by cases, with one case for each form of data. If you follow the design process, you will wind up with two laws for `singleton?` and four each for `has-n-elements?` and `nearly-same-lengths?`. But when you go to write the code, some funny things might happen:

- You might find yourself using `&&` or `||`, which makes the number of cases in the code less obvious than using `if`.
- You might find yourself writing an expression of the form `(if` e `#t #f)`. Such an expression is never acceptable; it should always be written as just e. This rewrite might reduce the total number of cases in your code so that your code has fewer cases than laws. If so, you can consider rewriting your laws so that you have fewer laws—or you could just leave the laws alone. Either tactic is OK.

Programming with nonempty lists

There are many computations, like "find the smallest," that work only on *nonempty* lists. In this set of problems, you define the forms of a nonempty list (two different ways), and you implement two functions that work only on nonempty lists.

N. *Define nonempty lists*. Many useful functions operate on *nonempty* lists. A nonempty list of A's is notated *LIST1(A)*.[13] The usual forms of data don't work here: `'()` is not a nonempty list. In this problem, you define nonempty lists in two different ways:

1. Define *LIST1(A)* in terms of *LIST(A)*. You may use set notation, a proof system, or the style of "An informal alternative" in the first section of the second lesson on program design. This definition must *not* be inductive.[14]

   If there are multiple cases in your definition, say what code you would write to distinguish the cases of a value `xs` in *LIST1(A)*.

   Warning: a useful definition says what *LIST1(A) is*, not what it isn't. Saying "a *LIST1(A)* is a *LIST(A)* that is not empty" is not useful. Your definition must be useful.

2. Define *LIST1(A)* inductively,[15] *without* any reference to *LIST(A)*. You may use set notation, a proof system, or the style of "An informal alternative" in the first section of the second lesson on program design. This definition must *not* mention *LIST(A)*.

   If there are multiple cases in your definition, say what code you would write to distinguish the cases of a value `xs` in *LIST1(A)*.

Both definitions are useful for writing code—different definitions for different functions.

Place your definitions with your code, in file `solution.scm`.

[13] Because it has at least one element.

[14] That's math talk for "the definition must not be recursive." That is, this definition of *LIST1(A)* must not refer to *LIST1(A)*.

[15] That's math talk for "recursively".

D. *Arg max.* This problem gives you a taste of higher-order functions, which we'll explore in more detail in the next homework assignment. Function `arg-max` expects two arguments: a function `f` that maps a value in set A to a number, and a *nonempty* list `as` of values in set A. It returns an element `a` in `as` for which `(f a)` is as large as possible. This function is commonly used in machine learning to predict the most likely outcome from a model.

```
-> (define square (a) (* a a))
-> (arg-max square '(5 4 3 2 1))
5
-> (arg-max car '((105 PL) (160 Algorithms) (170 Theory)))
(170 Theory)
```

Implement `arg-max`. *Be sure your implementation does not take exponential time.*[16]

Each function you define, including helper functions, must be accompanied by algebraic laws and by unit tests written using `check-expect` or `check-assert`.

E. *Rightmost point.* Page 194 of the book defines a `point` record. Copy that definition into your code. Define a function `rightmost-point` that takes a *nonempty* list of `point` records and returns the one with the largest x coordinate. Break ties arbitrarily.

For this problem, you need not write any algebraic laws. Write unit tests as usual.

To earn full credit for this problem, define `rightmost-point` without defining any *new* recursive functions (which means that `rightmost-point` itself must not be recursive).

A toolkit for sorting

Sorting is classic. In this set of problems, you define *properties* that can be used to test sorting functions ("a sorted list is a permutation of the original list") as well as *functions* that can be useful in sorting ("split a list into two nearly equal parts"). These tools can be combined into one of the most efficient sorting algorithms known: merge sort. (The merge sort itself is extra credit.)

F. *Copy removal.* Function `(remove-one-copy sx sxs)` expects an S-expression and a list of S-expressions. The list `sxs` contains one or more copies of `sx`. The function returns a new list which is like `sxs` except that one copy of `sx` is removed.

- An S-expression is considered a copy if it is `equal?` to another S-expression.
- If the caller violates the contract by calling `(remove-one-copy sx sxs)` where `sxs` does *not* contain a copy of `sx`, `remove-one-copy` causes a checked run-time error. (One option is to call the primitive function `error`.)

  This behavior should be tested using `check-error`.
- If there are multiple copies, the specification does not say which copy is removed.

```
-> (remove-one-copy 'a '(a b c))
(b c)
-> (remove-one-copy 'a '(a a b b c c))
(a b b c c)
-> (remove-one-copy 'a '(x y z))
Run-time error: removed-an-absent-item
```

[16] It is sufficient (but not necessary) to ensure that the body of `arg-max` contains only one call to `arg-max`.

```
-> (remove-one-copy '(b c) '((a b) (b c) (c d)))
((a b) (c d))
```

Implement `remove-one-copy`.

Each function you define, including helper functions, must be accompanied by algebraic laws and by unit tests written using `check-expect` or `check-assert`. In addition, you must write at least one unit test which verifies that in response to the contract violation mentioned above, `remove-one-copy` correctly signals a checked run-time error. For that test, use `check-error`.

G. *Permutations.* Lists `xs` and `ys` are permutations if and only if they have exactly the same elements—but possibly in different orders. Repeated elements must be accounted for. Write function `permutation?`, which tells if two lists of atoms are permutations.

```
-> (permutation? '(a b c) '(c b a))
#t
-> (permutation? '(a b b) '(a a b))
#f
-> (permutation? '(a b c) '(c b a d))
#f
```

Hint: Plan to use `remove-one-copy`.

Each function you define, including helper functions, must be accompanied by algebraic laws and by unit tests written using `check-expect` or `check-assert`.

H. *Splitting a list of values in two.* Function `split-list` takes a list of values `xs` and splits it into two lists of nearly equal length. More precisely `(split-list xs)` returns a *two-element list* `(cons ys (cons zs '()))` such that these properties hold:

- `(append ys zs)` is a permutation of `xs`
- `ys` and `zs` have nearly the same length; that is, their lengths differ by at most 1

You can choose *how* to split the `xs`. Here are a couple of examples:

```
-> (split-list '())
(() ())
-> (split-list '(a b))
((b) (a))   ;; ((a) (b)) would be equally good here
```

No matter how you choose to split the list, you should be able to test it with the following three properties, each of which is embodied in a function:

```
(define split-list-returns-two? (xs)
  (let ([result (split-list xs)])
    (has-n-elements? result 2)))

(define split-list-splits? (xs)
  (&& (split-list-returns-two? xs)
      (let ([result (split-list xs)])
         (permutation? xs (append (car result) (cadr result))))))

(define split-list-splits-evenly? (xs)
  (&& (split-list-splits? xs)
```

```
(let ([result (split-list xs)])
   (nearly-same-lengths? (car result) (cadr result))))
```

With these properties, you can write tests that track down failure, as in

```
(check-assert (split-list-splits-evenly? '(a b c)))
```

Implement `split-list`.

Each *recursive* function you define, including helper functions, must be accompanied by algebraic laws. Your algebraic laws for `split-list` are likely to be *more* specific than the problem definition—they describe not the problem as a whole, but your particular implementation.

Each *top-level* function, whether recursive or not, must be accompanied by unit tests written using `check-expect` or `check-assert`. (Inner functions defined with `letrec` can have algebraic laws, but they cannot be unit tested.) If you have a working version of `permutation?`, it is acceptable for your test cases to call it.

Your test cases for `split-list` should include one of the following:

- Tests showing that `split-list-splits-evenly?` is satisfied on multiple lists of S-expressions
- A note explaining why your code cannot pass such tests (e.g., explaining which functions are not working)

Extra credit, programming: Merge sort

M. *Merge and Merge sort.* For extra credit, implement merge sort. Begin with function `merge`, which expects two lists of numbers sorted in increasing order and returns a single list sorted in increasing order containing exactly the same elements as the two argument lists together:

```
-> (merge '(1 2 3) '(4 5 6))
(1 2 3 4 5 6)
-> (merge '(1 3 5) '(2 4 6))
(1 2 3 4 5 6)
```

Function `merge`, plus any helper functions, must be accompanied by algebraic laws and by unit tests written using `check-expect` or `check-assert`.

Now use `split-list` and `merge` to define a recursive function `merge-sort`, which is given a list of numbers and returns a sorted version of that list, in increasing order. You need not write any algebraic laws for `merge-sort`.

Extra credit, theory: take and drop

TDP. *Proof of a take-drop law.* Using the same techniques you used to solve exercise 37, plus your algebraic laws for `take` and `drop`, prove the append-take-drop law: for any list of values `xs` and natural number `n`,

```
(append (take n xs) (drop n xs)) == xs
```

What and how to submit

Please submit four files:

- A `README` file containing
  - The names of the people with whom you collaborated
  - A list identifying which problems that you solved
- A text file `cqs.scheme.txt` containing your answers to the reading-comprehension questions (you can start with our file[17])
- A PDF file `theory.pdf` containing the solutions to Exercises 1, 37, and **A**. If you already know LaTeX[18], by all means use it. You may benefit by emulating our LaTeX source code for a simple proof system[19] or Sam Guyer's LaTex source code for typesetting operational semantics[20]. Otherwise, write your solution by hand and scan it or photograph it. Do check with someone else who can confirm that your work is legible—if we cannot read your work, we cannot grade it.

  Please leave your name out of your PDF—that will enable your work to be graded anonymously.
- A file `solution.scm` containing the solutions to all the other exercises, including your written answers in comments to the questions posed by exercise N

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

How your work will be evaluated

Programming in μScheme

The criteria we will use to assess the structure and organization of your μScheme code, which are described in detail below, are mostly the same as the criteria in the general coding rubric[21], which we used to assess your Impcore code. But some additional criteria appear below.

Laws must be well formed and algorithmic

[17] ./cqs.scheme.txt
[18] http://www.latex-project.org/
[19] ../handouts/noset.tex
[20] ../handouts/latexexample.tex
[21] ../coding-rubric.html

	Exemplary	**Satisfactory**	**Must Improve**
Laws	• When defining function f, each left-hand side applies f to one or more *patterns*, where a pattern is a form of input (examples: `(+ m 1)`, `(cons x xs)`). • When a law applies only to equal inputs, those inputs are notated with the same letter. • The left-hand side of each algebraic law applies the function being defined. • On the left-hand side of each algebraic law, the number and types of arguments in the law are the same as the number and types of arguments in the code. • The only variables used on the right-hand side of each law are those that appear in arguments on the left-hand side. • When a variable on a left-hand side is part of a form-of-data argument, that variable is used on the right-hand side as a part of the argument. • For every permissible form of the function's input or inputs, there is an algebraic law with a matching left-hand side (and a matching side condition, if any). • The patterns of the left-hand sides of laws defining function f are all mutually exclusive, *or* • The patterns of the left-hand sides of laws defining function f are either mutually exclusive or are distinguished with side conditions written on the right-hand side.	• On a left-hand side, f is applied to a form of input, but the form of input is written in a way that is not consistent with code. • When a law applies only to equal inputs, the equality is written as a side condition. • Once or twice in an assignment, a variable appears on the right-hand side of a law without also appearing on the left-hand side. The variable appears to name an argument. • Once or twice, a variable on a left-hand side is part of a form-of-data argument, but on the right-hand side, it is used as if it were the whole argument. • For every permissible form of the function's input or inputs, there is an algebraic law with a matching left-hand side, but some inputs might inadvertently be excluded by side conditions that are too restrictive. • Laws are distinguished by side conditions, but the side conditions appear on the left-hand side. • There are some inputs that match more than one left-hand side, and these inputs are not distinguished by side conditions, but the laws contain a note that the ambiguity is intentional, and for such inputs, the right-hand sides all specify the same result.	• One or more left-hand sides contain laws that are not applications of f. • On a left-hand side, f is applied to something that is not a form of input, like an arbitrary sum `(+ j k)` or an `append`. • The left-hand side of an algebraic law applies some function *other* than the one being defined. • The left-hand side of an algebraic law the function being defined to the wrong number of arguments, or to arguments of the wrong types. • The right-hand side of a law refers to a variable that is not part of the left-hand side and which appears not to refer to an argument. • The assignment shows a pattern of using argument variables on right-hand sides, instead of or in addition to the variables that appear on left-hand sides. • The assignment shows a pattern of using part-of-data variables as if they were whole arguments. • There is permissible input whose form is not matched by the left-hand side of any algebraic law. • There is at least one input to which it is ambiguous which law should apply: the input matches more than one left-hand side, and either there are no side conditions, or the side conditions are insufficient to distinguish the ambiguous laws. *And* there is no note explaining that the ambiguity is intentional and OK.

Exemplary	Satisfactory	Must Improve

Code must be well structured

We're looking for functional programs that use Boolean and name bindings idiomatically. Case analysis must be kept to a minimum.

	Exemplary	Satisfactory	Must Improve
Structure	• The assignment does not use `set`, `while`, `print`, or `begin`. • Wherever Booleans are called for, code uses Boolean values `#t` and `#f`. • The code has as little case analysis as possible (i.e., the course staff can see no simple way to eliminate any case analysis) • When possible, inner functions use the parameters and `let`-bound names of outer functions directly.	• The code contains case analysis that the course staff can see follows from the structure of the data, but that could be simplified away by applying equational reasoning. • 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.	• Some code uses `set`, `while`, `print`, or `begin` (**No Credit**). • Code uses integers, like 0 or 1, where Booleans are called for. • The code contains superfluous case analysis that is not mandated by the structure of the data.

Code must be well laid out, with attention to vertical space

In addition to following the layout rules in the general coding rubric (80 columns, no offside violations), we expect you to use vertical space wisely.

	Exemplary	Satisfactory	Must Improve
Form	• Code is laid out in a way that makes good use of scarce vertical space. Blank lines are used judiciously to break large blocks of code into groups, each of which can be understood as a unit.	• Code has a few too many blank lines. • Code needs a few more blank lines to break big blocks into smaller chunks that course staff can more easily understand.	• Code wastes scarce vertical space with too many blank lines, block or line comments, or syntactic markers carrying no information. • Code preserves vertical space too aggressively, using so few blank lines that a reader suffers from a "wall of text" effect. • Code preserves vertical space too aggressively by crowding multiple expressions onto a line using some kind of greedy algorithm, as opposed to a layout that communicates the syntactic structure of the code. • In some parts of code, every single line of code is separated form its neighbor by a blank line, throwing away half of the vertical space (**serious fault**).

Code must load without errors

Ideally you want to pass all of *our* correctness tests, but at minimum, your own code must load and pass its own unit tests.

	Exemplary	Satisfactory	Must Improve
Correctness	• Your μScheme code loads without errors. • Your code passes all the tests we can devise. • *Or*, your code passes all tests but one.	• Your code fails a few of our stringent tests.	• Loading your μScheme into `uscheme` causes an error message (**No Credit**). • Your code fails many tests.

Costs of list tests must be appropriate

Be sure you can identify a nonempty list in constant time.

	Exemplary	**Satisfactory**	**Must Improve**
Cost	• 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.

Your proofs

The proofs for this homework are different from the derivations and metatheoretic proofs from the operational-semantics homework, and different criteria apply.

	Exemplary	**Satisfactory**	**Must Improve**
Proofs	• Course staff find proofs short, clear, and convincing. • Proofs have exactly as much case analysis as is needed (which could mean no case analysis) • Proofs by induction explicitly say what data is inducted over and clearly identify the induction hypothesis. • Each calculational proof is laid out as shown in the textbook, with each term on one line, and every equals sign between two terms has a comment that explains why the two terms are equal.	• Course staff find a proof clear and convincing, but a bit long. • *Or*, course staff have to work a bit too hard to understand a proof. • A proof has a case analysis which is complete but could be eliminated. • A proof by induction doesn't say explicitly what data is inducted over, but course staff can figure it out. • A proof by induction is not explicit about what the induction hypothesis is, but course staff can figure it out. • Each calculational proof is laid out as shown in the textbook, with each term on one line, and most of the the equals signs between terms have comments that explain why the two terms are equal.	• Course staff don't understand a proof or aren't convinced by it. • A proof has an incomplete case analysis: not all cases are covered. • In a proof by induction, course staff cannot figure out what data is inducted over. • In a proof by induction, course staff cannot figure out what the induction hypothesis is. • A calculational proof is laid out correctly, but few of the equalities are explained. • A calculational proof is called for, but course staff cannot recognize its structure as being the same structure shown in the book.