Functional programming in μScheme

COMP 105 Assignment

Due Tuesday, February 13, 2018 at 11:59PM

Contents

Overview 2
Setup 2
Diagnostic tracing 2
Dire Warnings 3
Reading Comprehension (10 percent) 4
Programming and Proof Problems (90 percent) 6
  Problem Details (Theory) 6
  Our expectations for your code: Algebraic laws and unit tests 7
  How to write algebraic laws 7
    How to tell when laws are algorithmic 8
    Breaking list inputs down by cases 8
    One really bad mistake with lists 9
    Breaking S-expression inputs down by cases 9
    Breaking natural-number inputs down by cases 10
    You can’t break a function down by cases 10
    Laws must be consistent with the code 11
  Problem Details (Code) 11
  Extra credit: Merge sort 15

What and how to submit 15

How your work will be evaluated 16
  Programming in μScheme 16
    Code must be well structured 16
    Code must be well laid out, with attention to vertical space 16
    Code must load without errors 17
    Costs of list tests must be appropriate 18
  Explaining let 18
  Your proofs 18

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

This assignment has three purposes:

- To help you learn to design using algebraic laws:
  1. Look at the data
  2. Write algebraic laws
  3. Write code
- To give you extensive practice writing functions that work with lists and S-expressions
- To give you a little bit more practice with programming-language theory and proofs.

The assignment is based primarily on material from sections 2.1 to 2.6 of *Build, Prove, and Compare*. You will also need to know the syntax in section 2.11, which starts on page 144, and the initial basis (also in section 2.11). The table on page 156 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; 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. 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 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 73 on page 233 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:

\(^1\)http://www.cs.tufts.edu/comp/105/homework/scheme_solution_template.scm
Used carefully, \texttt{\&trace} can save you a lot of time and effort. \textbf{But do not leave even an unexecuted reference to \texttt{\&trace} in your submission.}

**Dire Warnings**

Since we are studying functional programming, the \muScheme programs you submit must not use any imperative features. \textbf{Banish set, while, println, print, printu, and begin from your vocabulary!} \textbf{If you break this rule for any problem, you will get No Credit for that problem.} You may find it useful to use \texttt{begin} and \texttt{println} while debugging, but they must not appear in any code you submit. As a substitute for assignment, use \texttt{let} or \texttt{let*}.

The primary purpose of the assignment is to teach you structural recursion. In particular, you are to learn how to work with lists under a “nil and cons” model, not a “length” model. Therefore, \textbf{you may use length for at most two problems on this assignment:}

- You may use \texttt{length} to test \texttt{arg-max}, but not to implement it.
- You may pick any one other problem for which you are free to use \texttt{length} in both implementation and in testing.

It is not necessary to use \texttt{length} at all, but there is a problem for which it is very convenient.

Helper functions may be defined at top level provided 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 \texttt{check-expect} or \texttt{check-assert}, and possibly \texttt{check-error}.

As an alternative to helper functions, you may read ahead and define local functions using \texttt{lambda} along with \texttt{let}, \texttt{letrec}, or \texttt{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.

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

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

\textit{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.

\textbf{Code you submit must not even mention \texttt{\&trace}.}

\(^2\)../coding-rubric.html
\(^3\)../coding-rubric.html
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. You may wish to use the template 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.

1. Review section 2.2 on list primitives and S-expression literals 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 H.

2. Review the first few pages of section 2.3, through the end of section 2.3.2. Which of the following expressions evaluate to #t for every list 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

3. Read section 2.3.2, then please explain in your own words the difference between simple-reverse and reverse.

   You are now mostly ready for exercise 35.

4. Read about association lists in section 2.3.6, which starts on page 103. 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 2.16.6, which starts on page 191. Imagine that μScheme is given the following definition:

   (record 3point (x y z))

---


5: [/cqscheme.txt](/cqscheme.txt)
This definition puts five functions into the environment $\rho$. What are their names?

You are now mostly ready for exercise E.

6. Review the first part of section 2.4, which starts on page 105, up to and including section 2.4.4. Now think up a new algebraic law that applies to some combination of append and length. Write your law in the style of section 2.4.4. Like the other list laws in that section, your law must mention a variable $x$, which must be allowed to be any arbitrary list.

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

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):

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

Scheme does not have local variables, so to translate this function into $\mu$Scheme, you must use let, let*, or letrec. Which of the following three translations sensibly captures the intent and behavior of the original function?

For each of the following parts, please answer “yes” or “no”:

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

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

You are now ready for exercise 30.
Programming and Proof Problems (90 percent)

For the “programming and proof” part of this assignment, you will do exercises 1, 2, 10, 30, and 35 in the book, plus the problems A through J below—but not in that order. There is also one extra-credit problem: problem M.

Problem Details (Theory)

1. *A list of S-expressions is an S-expression.* Do exercise 1 on page 203 of *Build, Prove, and Compare*. The hint in the book suggests using a proof by contradiction, but most students prefer to use structural induction. 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 are on page 113.

35. *Calculational proof.* Do exercise 35 on page 217 of *Build, Prove, and Compare*, proving that reversing a list does not change its length.

**Hint:** structural induction.

**Related Reading:** section 2.4.5, which starts on page 107

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

\[(\text{cdr} (\text{cons} \ x \ \text{xs})) == \text{xs}\]

b) 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 \((\text{cdr} \ (\text{cons} \ e_1 \ e_2))\) terminates, and
- \((\text{cdr} \ (\text{cons} \ e_1 \ e_2)) == e_2\)

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

If you believe the conjecture, you can establish it by proving that it’s true for *every* choice of \(e_1\), \(e_2\), and 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 154. Or if you prefer, you can use an Impcore-style semantics extended with rules from this handout.\(^6\)

---

\(^6\)handouts/list-opsem.pdf
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!

- There must be enough algebraic laws to cover all inputs that respect the function’s contract.
- No algebraic law may be completely redundant. That is, no law may be implied by a combination of other laws. However, 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.
- Every algebraic law must be tested by a check-expect or check-assert.

A law may have a side condition. A side condition might say that two values are different, or that a given value is found in a given set, or any number of other things.

For ideas on writing algebraic laws, consult the lecture slides and the example laws in section 2.4.4, which starts on page 107. You may also see a very simple example of such design for binary trees on pages 114 and 115.

For some problems, algebraic laws are not needed or are already given to you. Those problems are noted below.

How to write algebraic laws

We write algebraic laws as part of a design method: laws make it easy to write code, and also easy for readers to be confident that code is correct. To write such laws, you may need a little guidance.

To begin, we need to distinguish between algorithmic laws and property laws. A set of laws is algorithmic if it nails down exactly how a function is supposed to work. For example, here is a complete set of algorithmic laws for length:

\[
\begin{align*}
(length \ '()) & == 0 \\
(length \ (cons \ x \ xs)) & == (+ \ 1 \ (length \ xs))
\end{align*}
\]

These laws prescribe, completely, everything that the length function has to do. Here, for contrast is a property law for length, which says only that a nonempty list has a positive length:

\[
(length \ (cons \ x \ xs)) > 0 \quad ; \text{useful for testing, not design}
\]

Property laws like these are used for specification, for testing, and for simplifying code, but they are not enough to enable you to design an implement functions. Here is another property law:

\[
(length \ (append \ xs \ ys)) = (+ \ (length \ xs) \ (length \ ys))
\quad ; \text{useful for testing and simplification, not for design}
\]

This law makes for excellent test cases, and it is a superb way to make your code simpler and more efficient, but it is useless for design. Why? Because there is no way that a length function can know that its argument has the form (append \ xs \ ys).

Here are some more property laws, from page 107 of the textbook:
Using these laws, you can simplify and speed up code, but the laws won’t tell you how to write append. The algorithmic laws for append look like this:

; algorithmic laws
(append (cons x xs) ys) = (cons x (append xs ys))
(append '() ys) = ys

How to tell when laws are algorithmic

Algorithmic laws have these properties:

- The left-hand sides break the inputs down by cases.
- 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.
- In every recursive call on every right-hand side, some input is getting smaller.

The key skill here is breaking inputs down by cases. Cases typically ask an input “how were you formed?”, and they must have these properties:

- You can tell which case is which via a constant-time test, like (null? xs) or (= n 0).
- If an input was formed from other inputs, like a cons cell or a nonempty association list, you must be able to get at the parts from which the input was formed. For example, you can get the parts of a cons cell using car and cdr.

Let’s look at case analysis in more detail.

Breaking list inputs down by cases

If an input is in LIST(A), then a typical function treats it as one of two cases:

'()
(cons y ys), where y is in A and ys is in LIST(A)

These cases can always be distinguished by calling null?, and when the input was formed using cons, we find y and ys using car and cdr.

Another possibility is that a list input is nonempty and in LIST(A). Cases for a nonempty list are a little different:

(cons y '()), where y is in A
(cons y (cons z zs)), where y and z are in A, and zs is in LIST(A)

If xs is one of these two cases, we can distinguish the cases by calling (null? (cdr xs)). Notice that if xs = (cons y (cons z zs)), then (cons z zs), which is the same as (cdr xs), is also a nonempty list of A, so it can be used in a recursive call.
Finally, if a list could be either nil or cons, and the law holds regardless, you should write the list just as a variable:

xs, which is in LIST(A)

As an example, the second parameter to append is like this. When you write a variable, you are promising never to apply null?, car, or cdr to that variable.

Here’s a common mistake with variables:

```lisp
(sublist? xs '()) == #f ;; WRONG
(sublist? '() ys) == #t
... more cases below ...
```

The student who wrote xs and ys meant for them to be nonempty. But a variable could be any list, including the empty list. In this example, if both xs and ys are empty, the laws give inconsistent results. That’s how we’re certain it’s wrong. Here’s a correct version:

```lisp
(sublist? (cons x xs) '()) == #f ;; RIGHT
(sublist? '() (cons y ys)) == #t
... more cases below ...
```

These left-hand sides can’t possibly be confused.

This version can be refined: if the first argument is empty, we don’t really care what the second argument is—the empty list is a sublist of any list. So we could write the laws this way:

```lisp
(sublist? (cons x xs) '()) == #f ;; RIGHT
(sublist? '() zs) == #t ;; SPLENDID
... more cases below ...
```

The advantage of this form is that we might then have to consider fewer alternatives in the “more cases below.”

One really bad mistake with lists

Misusing variables is wrong, but we can usually tell what you meant, and you’ll probably get the code right. Where you can go off the rails is if you put something like list or append on the left-hand side of an algebraic law. There is no way to know whether or how a list may have been formed using list or append—all we can tell in the code is “nil or cons?”—so such a law cannot possibly be algorithmic.

Breaking S-expression inputs down by cases

A list can be formed in only two ways. But an S-expression can be formed in three ways:

'()

(cons z zs), where z is an S-expression and zs is a list of S-expressions

a, where a is an atom and not '()

These cases can be distinguished using a combination of null? and pair?.

A common mistake here is to forget the side condition on a. Here are some mistaken laws for counting the number of atoms in an S-expression:
\[(\text{atom-count } '()) = 0\]
\[(\text{atom-count } (\text{cons } z \ zs)) = (+ (\text{atom-count } z) (\text{atom-count } zs))\]
\[(\text{atom-count } a) = 1 ;; \text{WRONG}\]

The last law needs a side condition:
\[(\text{atom-count } a) = 1, \text{where } a \text{ is a non-null atom}; ;; \text{RIGHT}\]

**Breaking natural-number inputs down by cases**

As you might guess from the proof-system handout, there are a zillion ways to break natural numbers down by cases. But the most common way is using the Peano system, which says that a natural number is formed in one of two ways:

\[0\]
\[(+ n 1), \text{where } n \text{ is a natural number}; ;; \text{RIGHT}\]

Writing \[(+ n 1)\] everywhere gets old quickly, so we usually prefer to do it this way:

\[0\]
\[n, \text{where } n \neq 0; ;; \text{ALSO RIGHT}\]

Here’s an example in context. You can use this template to write laws for `take` and `drop`:

\[(\text{take } 0 \ ...) = ...\]
\[(\text{take } n \ ...) = ... n \ ...; \text{where } n \neq 0; ;; \text{RIGHT}\]

The side condition on \(n\) is required; forgetting it is a common mistake:

\[(\text{take } 0 \ ...) = ...\]
\[(\text{take } n \ ...) = ... n \ ...; ;; \text{WRONG}\]

**You can’t break a function down by cases**

Some of the problems below, like `takewhile`, `dropwhile`, and `arg-max`, take functions as inputs. You can’t break a function down by cases, because there’s no way to ask a function how it was formed. All you can do with a function is apply it. How, then, should a function appear in an algebraic law? As a variable. Here’s an example, for `takewhile`, which takes two arguments, a predicate \(p?\) and a list of values. A function has one case and a list has two, and multiplied together there are two in total:

\[(\text{takewhile } p? '()) = ...\]
\[(\text{takewhile } p? (\text{cons } x \ xs)) = ...\]

That’s not the end of the story, however: once we have both \(p?\) and an \(x\) that we could apply \(p?\) to, we could have extra cases depending on whether \((p? \ x)\) is true or false. Those cases would be written as side conditions.

One final example: function `arg-max` takes a function and a *nonempty* list of values. The left-hand sides look like this:

\[(\text{arg-max } f (\text{cons } x '())) = ...\]
\[(\text{arg-max } f (\text{cons } x1 (\text{cons } x2 \ xs))) = ...\]
Laws must be consistent with the 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. We look especially hard at case analysis: the number of cases in your code should be 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.

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 97. 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 155.

30. Let-binding. Do parts (a) and (b) of exercise 30 on page 217 of Build, Prove, and Compare. You should be able to answer the questions in at most a few sentences. Your answers go in comments in file solution.scm.

In this problem, you don’t define any functions, so you don’t write any algebraic laws or any unit tests.

Related Reading: Information on let can be found in section 2.5 (pages 111–112)

2. Recursive functions on lists. Do all parts of exercise 2 on page 203 of Build, Prove, and Compare. 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? are perhaps too challenging for beginners, so you may omit them. But do write laws for all other functions, including helper functions.

Hints:

• It is acceptable to extend the contracts of the LIST(SEXP) functions so that they can also accept an SEXP. Extending a contract in this way can simplify the code.

• 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.


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. It returns the longest prefix of xs that contains at most n elements.
Function \((\text{drop } n \text{ } \text{xs})\) expects a natural number and a list. Roughly, it removes \(n\) elements from the front of the list. When acting together, take and drop obey this algebraic law: for any list \(\text{xs}\) and natural number \(n\),

\[
(\text{append } (\text{take } n \text{ } \text{xs}) \text{ } (\text{drop } n \text{ } \text{xs})) = \text{xs}
\]

Implement take and drop.

Each function you define, including helper functions, must be accompanied by algebraic laws and by unit tests written using \texttt{check-expect} or \texttt{check-assert}. Be aware that the algebraic law above (the “append/take/drop” law) is necessary but not sufficient to characterize 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 \texttt{zip} converts a pair of lists to an association list; \texttt{unzip} converts an association list to a pair of lists. If \texttt{zip} is given lists of unequal length, its behavior is not specified.

\[
\begin{align*}
\texttt{zip '}(1 \text{ } 2 \text{ } 3) \text{ '}(a \text{ } b \text{ } c) &\rightarrow ((1 \text{ a}) \text{ } (2 \text{ b}) \text{ } (3 \text{ c})) \\
\texttt{unzip '((I Magnin) \text{ } (U Thant) \text{ } (E Coli))) &\rightarrow ((I \text{ U E}) \text{ } (Magnin \text{ Thant} \text{ Coli}))
\end{align*}
\]

Provided lists \(\text{xs}\) and \(\text{ys}\) are the same length, \texttt{zip} and \texttt{unzip} satisfy these algebraic laws:

\[
(\text{zip } (\text{car } (\text{unzip pairs})) \text{ } (\text{cadr } (\text{unzip pairs}))) = \text{pairs}
\]

\[
(\text{unzip } (\text{zip } \text{xs } \text{ys})) = (\text{list2 } \text{xs } \text{ys})
\]

You must write additional algebraic laws for \texttt{zip}.

Implement \texttt{zip} and \texttt{unzip}.

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

- The algebraic laws for \texttt{unzip} are too challenging for beginners, so you may omit them.

**Related Reading:** Information on association lists can be found in section 2.3.6, which starts on page 103.

D. Arg max. This problem gives you a taste of higher-order functions, which we’ll be covering in more detail in the next homework assignment. Function \texttt{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 \((f \text{ } a)\) is as large as possible.

\[
\begin{align*}
\texttt{define square (a) (* a a)} &\rightarrow (\text{define square '}(5 \text{ } 4 \text{ } 3 \text{ } 2 \text{ } 1)) \\
\texttt{(define invert (a) (/ 1000 a)) &\rightarrow (\text{define invert '}(5 \text{ } 4 \text{ } 3 \text{ } 2 \text{ } 1))}
\end{align*}
\]

Implement \texttt{arg-max}. Be sure your implementation does not take exponential time.

Each function you define, including helper functions, must be accompanied by algebraic laws and by unit tests written using \texttt{check-expect} or \texttt{check-assert}. 
**Hint:** A nonempty list of values has one of the following two forms:

\[
(\text{cons } x \ '())
\]

\[
(\text{cons } x1 \ (\text{cons } x2 \ xs)), \text{ where } xs \text{ is a list of values}
\]

This information should be reflected in your algebraic laws and in your code.

**Hint:** Elements of the list as might not be numbers. Find a way to test your code with a list of non-numbers.

**E. Most distant point.** Page 191 of the book defines a point record. The distance of a point from the main diagonal through the origin is given by \(|x - y|\) (absolute value of the difference of the coordinates). Define a function `maximally-distant-point` that takes a nonempty list of point records and returns the one that is most distant from the main diagonal. That is, it returns a point from the list such that the distance of that point from the main diagonal is as large as possible.

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

You are welcome to copy and paste this definition of absolute value:

```scheme
;; laws: (abs n) == (- 0 n), when n < 0
;; (abs n) == n, when n >= 0
(check-expect (abs -12) 12)
(check-expect (abs 3) 3)

;; absolute value
(define abs (n)
  (if (< n 0) (- 0 n) n))
```

**F. Merging sorted lists.** Implement 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:

```scheme
-> (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)
```

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

**G. Interleaving lists.** Implement function `interleave`, which expects as arguments two lists `xs` and `ys`, and returns a single list obtained by choosing elements alternately, first from `xs` and then from `ys`. When either `xs` or `ys` runs out, `interleave` takes the remaining elements from the other list, so that the elements of the result are exactly the elements of the two argument lists taken together.

```scheme
-> (interleave '(1 2 3) '(a b c))
(1 a 2 b 3 c)
-> (interleave '(1 2 3) '(a b c d e f))
(1 a 2 b 3 c d e f)
-> (interleave '(1 2 3 4 5 6) '(a b c))
(1 a 2 b 3 c 4 5 6)
```

N.B. This is another function that consumes two lists.
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. Copy removal. Function remove-one-copy takes an S-expression and a list which includes one or more copies of that S-expression. It returns a new list which is like the original list except that one copy of the S-expression 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 on a list that does not contain the given S-expression, remove-one-copy causes a checked run-time error. (One option is to call the primitive function 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
-> (remove-one-copy '(b c) '((a b) (b c) (c d)))
((a b) (c d))

Implement remove-one-copy.

I. 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 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

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

J. Splitting a list in two. Function split-list takes a list 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
- (length ys) and (length zs) differ by at most 1

You have a lot of freedom to choose how you want to split the xs. Here are a couple of examples:

-> (split-list '())
Define \texttt{split-list}.

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

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

\textbf{Extra credit: Merge sort}

\textit{M. Merge sort.} For extra credit, use your functions \texttt{split-list} and \texttt{merge} to define a recursive function \texttt{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.

\section*{What and how to submit}

Please submit four files:

\begin{itemize}
  \item A \texttt{README} file containing
    \begin{itemize}
      \item The names of the people with whom you collaborated
      \item A list identifying which problems that you solved
    \end{itemize}
  \item A text file \texttt{cqs.scheme.txt} containing your answers to the reading-comprehension questions (you can start with our file\footnote{\texttt{cqs.scheme.txt}})
  \item A PDF file \texttt{theory.pdf} containing the solutions to Exercises 1, 35, and A. 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.
    \begin{itemize}
      \item As long as your name is in your README, you can leave it out of your PDF—that will enable your work to be graded anonymously.
    \end{itemize}
  \item A file \texttt{solution.scm} containing the solutions to all the other exercises, including your written answers in comments to the questions posed by exercise 30.
\end{itemize}

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

\footnote{\texttt{cqs.scheme.txt}}
\footnote{http://www.latex-project.org/}
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\(^9\), which we used to assess your Impcore code. But some additional criteria appear below.

Code must be well structured

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

<table>
<thead>
<tr>
<th>Exemplary</th>
<th>Satisfactory</th>
<th>Must Improve</th>
</tr>
</thead>
<tbody>
<tr>
<td>Structure</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• The assignment does not use set, while, print, or begin.</td>
<td>• 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.</td>
<td>• Some code uses set, while, print, or begin (No Credit).</td>
</tr>
<tr>
<td>• Wherever Booleans are called for, code uses Boolean values #t and #f.</td>
<td>• 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.</td>
<td>• Code uses integers, like 0 or 1, where Booleans are called for.</td>
</tr>
<tr>
<td>• The code has as little case analysis as possible (i.e., the course staff can see no simple way to eliminate any case analysis)</td>
<td></td>
<td>• The code contains superfluous case analysis that is not mandated by the structure of the data.</td>
</tr>
<tr>
<td>• When possible, inner functions use the parameters and let-bound names of outer functions directly.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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.

\(^9\)../coding-rubric.html
<table>
<thead>
<tr>
<th>Form</th>
<th>Exemplary</th>
<th>Satisfactory</th>
<th>Must Improve</th>
</tr>
</thead>
</table>
| ● 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 from 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.

<table>
<thead>
<tr>
<th>Correctness</th>
<th>Exemplary</th>
<th>Satisfactory</th>
<th>Must Improve</th>
</tr>
</thead>
</table>
| ● 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.

<table>
<thead>
<tr>
<th>Exemplary</th>
<th>Satisfactory</th>
<th>Must Improve</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cost</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Empty lists are distinguished from</td>
<td></td>
<td>• Distinguishing an empty list from</td>
</tr>
<tr>
<td>non-empty lists in constant time.</td>
<td></td>
<td>a non-empty list might take longer</td>
</tr>
<tr>
<td></td>
<td></td>
<td>than constant time.</td>
</tr>
</tbody>
</table>

Explaining let

Here is what we expect from your explanation of the strange let in exercise 30.

<table>
<thead>
<tr>
<th>Exemplary</th>
<th>Satisfactory</th>
<th>Must Improve</th>
</tr>
</thead>
<tbody>
<tr>
<td>Let</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Your explanation of the strange</td>
<td></td>
<td>• Your explanation of the strange</td>
</tr>
<tr>
<td>let code is accurate and appeals to</td>
<td></td>
<td>let code does not identify which</td>
</tr>
<tr>
<td>the relevant semantic rules by name.</td>
<td></td>
<td>rules of the ( \mu )Scheme semantics</td>
</tr>
<tr>
<td>The meanings of the rules are</td>
<td></td>
<td>must be used to explain the code.</td>
</tr>
<tr>
<td>explained informally.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Your proofs

The proofs for this homework are different from the derivations and metatheoretic proofs from the operational-semantics homework, and different criteria apply.
<table>
<thead>
<tr>
<th>Proofs</th>
<th>Exemplary</th>
<th>Satisfactory</th>
<th>Must Improve</th>
</tr>
</thead>
</table>
|        | • 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 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. |