Functional programming in μScheme

CS 105 Assignment

Due Tuesday, September 27, 2022 at 11:59PM

Contents

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

Introduction

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. From CS 15, you know how to manipulate these structures using machine-level abstractions that operate on one word and one pointer at a time. In 105 you will 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 here 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 2.4 of *Programming Languages: Build, Prove, and Compare*. You will also need to know the syntax in figure 2.2 (page~95), and the initial basis (also in section~2.2). The initial basis is summarized in table 2.3 on page~99—**that table 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\text{ 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 the model 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`. 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 specified in exercise~60 on page~200 of *Build, Prove, and Compare*. This exercise asks for an implementation of the tracing facility. But we have already implemented it for you! We used the approach sketched in part (b) of the exercise.

If the specification is a bit murky, you can probably master the tracing facility just by playing around with it. Here are a couple of examples:

```
-> (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.
- Each helper function is given an explicit contract—or, as described in the general coding rubric, 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 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.** It's best to use the template provided above, which has the correct function names.

Reading comprehension

Before starting the programming problems, answer the reading-comprehension questions for this module, which you will find online. It is also OK to alternate between reading-comprehension questions and related homework questions.

Theory problems

The problems are organized into two groups: theory and programming. Theory problems include exercise **22** in the book, plus problem **A** below. There are also two extra-credit theory problems: problems **TDP** and **CCL**.

22. *Calculational proof.* Do exercise~22 on page~185 of *Build, Prove, and Compare*: prove 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~2.5.7, which starts on page~116.
- Section~2.3.1, which starts on page~100, develops `append`, and it states two laws that you should use in your proof:

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

  You will find additional laws for `append` on page~115, but you may not use those additional laws—in particular, the third law is what you are trying to prove.
- The laws in the book are interrupted by explanations. We have condensed the basic laws of μScheme into a summary, which is online.

A. *From operational semantics to algebraic laws.* The contribution of operational semantics to program design is to justify algebraic laws. In this problem, you use the operational semantics of μScheme—which is simpler than the operational semantics of Impcore—to understand how and when a valid algebraic law can be used to simplify code. The 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`

   As a reminder, the judgment form for μScheme evaluation is $\langle e, \rho, \sigma \rangle \Downarrow \langle v, \sigma' \rangle$.

b) The preceding law can be used to simplify code in which `cdr` is applied to `cons` applied to variables. In this part, you show that when `cons` is applied to general expressions, the two sides aren't always equal.

   Find two expressions e_1 and e_2 and a context (that is, ρ and σ) such that all of the following are true:

   - The evaluation of e_1 terminates.
   - The evaluation of e_2 terminates.
   - The evaluation of `(cdr (cons` e_1 e_2`))` terminates, and
   - `(cdr (cons` e_1 e_2`))` $\neq e_2$

   For this problem, you may use *any* syntactic form of μScheme, including those that are forbidden in the programming parts of the assignment.

Related Reading: The operational semantics for `cons`, `car`, and `cdr` can be found on page~153.

Expectations for programming problems: 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

As described in the second lesson on program design, laws used to design a function must be algorithmic. 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. We expect the following:

- 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.

Programming problems

Programming problems include exercises **1**, **8**, and **31** in the book, plus the problems **B** through **E** below.

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~100. And you will definitely want to take advantage of μScheme's predefined and primitive functions (the initial basis). These functions are listed in section~2.2, which starts on page~93, and they are summarized in table 2.3 on page~99.

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."

1. *Recursive function on lists of values.* Do part (a) of exercise~1 on page~180 of *Build, Prove, and Compare* (`contig-sublist?`).

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. And each function you define, including `contig-sublist?` and any helper functions, must be accompanied by unit tests written using `check-expect` or `check-assert`.

Hint:

- If you're having trouble thinking about how to write `contig-sublist?` in Scheme, think about how you would implement it in C++: probably with a doubly nested loop. For this homework, then, 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.

8. *Recursive functions on lists of S-expressions.* Do parts (c) and (d) of exercise~8 on page~182 of *Build, Prove, and Compare* (`mirror` and `flatten`). 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`.

Related reading:

- Ordinary S-expressions are defined in in Figure 2.1 on page~93.
- The first section of the second *Lesson in Program Design* describes several ways to break down S-expressions. Look at the "expanded" version at the end of the section.

Hints:

- Every list of S-expressions is itself an S-expression—or formally, *LIST(SEXP)* $\subseteq$ *SEXP*. It is therefore 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. Use your judgment.
- Once you extend a contract, you can profitably break *SEXP* down by three cases: empty list, `cons`, and atom different from empty list.

Classic list functions

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

31. *Taking and dropping a prefix of a list* (`takewhile` and `dropwhile`). Do exercise~31 on page~187 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))
-> (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~107.

Programming with nonempty lists

Many useful functions, like "find the smallest," work only on *nonempty* lists. In this set of problems, you implement two such functions.

A nonempty list of A's is notated *LIST1(A)*.[1] The usual forms of data don't work here: `'()` is not a nonempty list. On the course web site you will find a short handout that defines nonempty lists in two different ways:

1. We can define *LIST1(A)* in terms of *LIST(A)*. This definition is *not* inductive.[2]
2. We can define *LIST1(A)* inductively,[3] *without* any reference to *LIST(A)*.

Both definitions are useful, because they say what *LIST1(A) is*, not what it isn't. (Saying "a *LIST1(A)* is a *LIST(A)* that is not empty" is not very useful.) And over time, you'll use both definitions for writing code—different definitions are good for different functions. For this homework, you'll use one or both to solve the following two problems.

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.*[4]

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

Avoid this common mistake: It's all too easy to return the *result* of applying `f`. But that's not what's in the contract: `arg-max` returns an *argument* `x` that, when passed to `f`, maximizes the value of `(f x)`.

E. *Rightmost point.* Here is a μScheme definition that defines a record for a point in the plane, with `x` and `y` coordinates:

```
(record point [x y])
```

[1]Because it has at least one element.
[2]That's math talk for "the definition is not recursive." That is, this definition of *LIST1(A)* does not refer to *LIST1(A)*.
[3]That's math talk for "recursively".
[4]It is sufficient (but not necessary) to ensure that the body of `arg-max` contains only one call to `arg-max`.

Copy this 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). Using recursive functions defined for other problems is OK.

Extra credit: theory of algebraic laws

Here are two extra-credit problems about justification of algebraic laws.

TDP. *Proof of a take-drop law.* Using the same techniques you used to solve exercise~22, 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
```

CCL. *Repair of a cdr-cons law.* Answer all three parts:

1. For what class of expressions e_1 and e_2 is it true that
   - `(cdr (cons` e_1 e_2`))` $= e_2$?
2. How would you prove it?
3. We know the law works if the *forms* of e_1 and e_2 are restricted to variables. Is there a less severe *syntactic* restriction (on the *forms* of expressions) that is sufficient to make the law work?

What and how to submit

Please submit three files:

- A `README` file containing
  - The names of the people with whom you collaborated
  - A list identifying which problems that you solved
- A PDF file `theory.pdf` containing the solutions to Exercises 22 and **A**. If you already know LaTeX, by all means use it. 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.

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, which we used to assess your Impcore code. But some additional criteria appear below.

Laws must be well formed and algorithmic

	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 (like a closing bracket on a line by itself). • 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.