Functional programming in μScheme

COMP 105 Assignment

Due Tuesday, February 14, 2017 at 11:59PM

Contents

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

Overview

The purpose of this assignment is to give you *extensive* practice writing functions that work with lists and S-expressions, plus a little bit more practice with programming-language theory and proofs. The assignment is based primarily on Sections 2.1 to 2.6 of Ramsey. You will also need to know the syntax in Section 2.11 and the initial basis in Section 2.13—**the table on page 149 is your lifeline**. Finally, although it is not necessary, you may find some problems easier to solve if you read ahead into Sections 2.7 to 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. There are a lot of problems, but only one hard one: in problem D, there is enough code that it can be tricky to get everything right. (µScheme is so expressive that you can get yourself into trouble even in a 12-line function.)

Setup

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

```
  ledit uscheme
```

USEFUL: We provide a template for solution.scm[1]. The "solution.scm" template 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 `print` 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 225 of Ramsey. 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.

Dire Warnings

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

[1]http://www.cs.tufts.edu/comp/105/homework/scheme_solution_template.scm

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.
- Each helper function is tested by `check-expect` 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.

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.

Code you submit must not even *mention* `&trace`.

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. We recommend that you complete them before starting the other problems below. You can download the questions[3].

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 explain why.

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

   To write your answers, use S-expression literals.

2. Review the first few pages of section 2.3, through the end of section 2.3.2. Which of the following is true for every *list* `xs`? Why?

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

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

[2] ../coding-rubric.html
[3] ./cqs.scheme.txt

4. Review the first part of section 2.4, up to 2.4.4, and ~~write~~ discover a new algebraic law that applies to some combination of `append` and `length`. ~~Your law should be~~ 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 `xs`, which must be allowed to be any arbitrary list.

5. Review section 2.4.5 and demonstrate the proper form for a calculational proof by writing a one-step proof that `(+` e `0)` equals e.

6. Imagine you are tasked with translating the following C function into μScheme:

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

   Review section 2.5, and answer these questions:

   (a) Is it sensible to use `let`? Why or why not?
   (b) Is it sensible to use `let*`? Why or why not?
   (c) Is it sensible to use `letrec`? Why or why not?

7. The `cons` cell is used for more than just lists—it's the basic element of every μScheme data structure. Almost like the assembly language of functional programming. What a `cons` cell means depends on how it is used and what is the programmer's intent.

   Review section 2.6, especially the definitions of *SEXP* and *LIST*$(\cdots)$, and answer these questions:

   (a) Is the value `(cons 3 4)` a member of *SEXP*? Why or why not?
   (b) Given that 3 and 4 are members of set *NUM*, is the value `(cons 3 4)` a member of set *LIST(NUM)*? Why or why not?

   Your answers should appeal to the numbered equations or named proof rules at the beginning of section 2.6.

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 **I** below. There is also one extra-credit problem: problem **M**.

Problem Details

Related Reading: Many of the following problems will ask you to write recursive functions on lists. You can sometimes emulate examples from Section 2.3 on pages 91–99. And you will definitely want to take advantage of μScheme's predefined and primitive functions (the initial basis), which are listed in section 2.13 on pages 148–150.

1. *A list of S-expressions is an S-expression.* Do Exercise 1 on page 195 of Ramsey. 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 107.

2. *Recursive functions on lists*. Do all parts of Exercise 2 of Ramsey. 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.

As part of the design of each function, try writing algebraic laws for each function, as shown in lecture and in the example laws in Section 2.4.4 on page 101. You can see a very simple example of such design for binary trees on pages 108 and 109.

Each function you define should be accompanied by test cases using `check-expect` or `check-error`. In comments, explain the purpose of each test.

10. *Taking and dropping a prefix of a list*. Do Exercise 10 on page 200 of Ramsey.

Each function you define should be accompanied by test cases using `check-expect` or `check-error`. In comments, explain the purpose of each test.

30. *Let-binding*. Do parts (a) and (b) of Exercise 30 on page 209 of Ramsey. You should be able to answer the questions in at most a few sentences.

Related Reading: Information on *let* can be found in Section 2.5 (pages 105–106)

35. *Calculational proof.* Do Exercise 35 on page 209 of Ramsey, proving that reversing a list does not change its length.

Hint: structural induction.

Related Reading: Section 2.4.5 (pages 101–104)

A. *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 `(drop n xs)` expects a natural number and a list. Roughly, it removes `n` elements from the front of the list. The exact semantics are given by this algebraic law: for any list `xs` and natural number `n`,

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

Implement `take` and `drop`.

Each function you define should be accompanied by test cases using `check-expect` or `check-error`. In comments, explain the purpose of each test.

B. *Zip and unzip*. Function `zip` converts a pair of lists to an association list; `unzip` converts an association list to a pair of lists. If `zip` is given lists of unequal length, its behavior is not specified.

```
-> (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))
```

Provided lists `xs` and `ys` are the same length, `zip` and `unzip` satisfy these algebraic laws:

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

Implement `zip` and `unzip`.

Related Reading: Information on association lists can be found in Section 2.3.6 (pages 97–99).

Each function you define should be accompanied by test cases using `check-expect` or `check-error`. In comments, explain the purpose of each test.

C. *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 `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.

```
-> (define square (a) (* a a))
-> (arg-max square '(5 4 3 2 1))
5
-> (define invert (a) (/ 1000 a))
-> (arg-max invert '(5 4 3 2 1))
1
```

Implement `arg-max`. **Hint:** the specification says that the list argument to `arg-max` is not empty. Take advantage.

Each function you define should be accompanied by test cases using `check-expect` or `check-error`. In comments, explain the purpose of each test.

D. *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:

```
-> (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 should be accompanied by test cases using `check-expect` or `check-error`. In comments, explain the purpose of each test.

E. *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.

```
-> (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 should be accompanied by test cases using `check-expect` or `check-error`. In comments, explain the purpose of each test.

F. *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.

- 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.
- 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`. Define test cases using `check-expect` and `check-error`, and explain in comments the purpose of each test. Take care that your tests do not rely on unspecified behavior, such as which copy is removed when there are multiple copies.

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 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 should be accompanied by test cases using `check-expect` or `check-error`. In comments, explain the purpose of each test.

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

Define `split-list`.

Each function you define should be accompanied by test cases using `check-expect` or `check-error`. In comments, explain the purpose of each test. If you have a working version of `permutation?`, it is acceptable for your test cases to call it.

I. *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) 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, 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.

Related Reading: The operational semantics for *cons*, *car*, and *cdr* can be found on pg 147. Or if you prefer, you can use an Impcore-style semantics extended with rules from this handout[4].

Extra credit: Merge sort

M. *Merge sort.* For extra credit, use your functions `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.

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
  - The number of hours you worked on the assignment
- A text file `cqs.scheme.txt` containing your answers to the reading-comprehension questions (you can start with our file[5])
- A PDF file `theory.pdf` containing the solutions to Exercises 1, 35, and **I**. If you already know LaTeX[6], by all means use it. Otherwise, write your solution by hand and scan it. Do check with someone else who can confirm that your work is legible—if we cannot read your work, we cannot grade it.
- A file `solution.scm` containing the solutions to all the other exercises, including your written answers in comments to the questions posed by Exercise 30.

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.

[4] ../handouts/list-opsem.pdf
[5] ./cqs.scheme.txt
[6] 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[7], 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 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.

[7] ../coding-rubric.html

	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.

Explaining `let`

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

	Exemplary	Satisfactory	Must Improve
Let	• Your explanation of the strange `let` code is accurate and appeals to the relevant semantic rules by name. The meanings of the rules are explained informally.	• Your explanation of the strange `let` code is accurate and appeals to the relevant semantic rules by name, but it does not explain the rules.	• Your explanation of the strange `let` code does not identify which rules of the μScheme semantics must be used to explain the code.

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.