COMP 105 Assignment: Functional programming in µScheme

Due Monday, February 11 at 11:59PM.

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

Preliminaries and Setup

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 3.1 through 3.6 of Ramsey. You will also need to know the syntax in Section 3.11 and the initial basis in Section 3.13—**the table on page 124 is your lifeline**. One question uses material from Section 3.12. Finally, although it is not necessary, you may find some problems easier to solve if you read ahead into Sections 3.7 through 3.9.

You will write about seventeen functions and do 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. (Scheme is so expressive that you can get yourself into trouble even in a 12-line function.)

The executable micro-Scheme 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
```

Dire Warnings

Since we're talking about functional programming, **the Scheme 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 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*`.

Helper functions may be defined at top level provided they have meaningful names and their contracts are documented. You may also 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.

Your solutions should be valid micro-Scheme; in particular, they must pass the following test:

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

without any error messages. 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 readability).

Overview, organization, and what to submit

For this assignment, you will do Exercises **1**, **2**, **14**, **5**, **30**, and **37** in the book, plus the problems **A** through **G** below. You will submit three files: `README`, `theory.pdf` (containing the solutions to 1, 30, 37, and G) and `solution.scm` (containing

the solutions to all the other exercises). You can create `theory.pdf` using LaTeX or Lyx another mathematical word processor, or you can write your solution by hand and scan it.

Details of all the problems

1. *A list of S-expressions is an S-expression.* Do Exercise 1 on page 154 of Ramsey. Do this proof before tackling Exercise 2; the proof should give you ideas about how to implement the code.
My estimate of difficulty: medium, because you haven't seen this kind of proof before.

2. *Recursive functions on lists*. Do all parts of Exercise 2 on page 154 of Ramsey. Expect to write some recursive functions, but you may also read ahead and use the higher-order functions in Sections 3.7 through 3.9.
My estimate of difficulty: if you exploit the result in Exercise 1, this problem is relatively easy. If not, you can get tangled up in case analyses.

5. *Taking and dropping a prefix of a list*. Do Exercise 5 on page 156 of Ramsey.
My estimate of difficulty: easy.

14. *Let-binding*. Do Exercise 14 on page 159 of Ramsey. You should be able to answer the questions in at most a few sentences. Place your answer as comments in file `solutions.scm`.
My estimate of difficulty: easy.

30. *Calculational proof*. Do Exercise 30 on page 164 of Ramsey, proving that reversing a list does not change its length. Put your solution in file `theory.pdf`.
My estimate of difficulty: medium.
Hint: structural induction.

37. *Operational semantics and language design*. Do all parts of Exercise 37 on page 165 of Ramsey. Be sure your answer to part (b) compiles and runs under `uscheme`. Put your answers to all parts in file `theory.pdf`.
My estimate of difficulty: easy (parts a and b) and medium (part c).

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`.
My estimate of difficulty: easy, provided you read the specification carefully

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`.
My estimate of difficulty: medium, provided you don't mind what happens to unspecified arguments. (If you insist on nailing

down the behavior in the unspecified cases, you may find that your code grows uncomfortably complex.)

C. *Arg max*.
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
-> (arg-max (lambda (x) (- 0 (square (- x 3)))) '(5 4 3 2 1))
3
```

Implement `arg-max`.

Hint: the specification says that list argument to `arg-max` is not empty. Exploit this part of the specification.
My estimate of difficulty: easy

D. *Graph functions*.
From COMP 15, you should be familiar with graphs and graph algorithms. In this problem you will write code that changes representations *directed* graphs. You will work with two representations:

- The first representation is a *list of edges*, where a single edge is represented by a two-element list. For example, the list `(A B)` represents an edge from `A` to `B`.
- The second representation uses a *successors map*: a graph is represented by an association list in which each node is associated with a list of its successors.

For example, the ASCII-art graph

```
A --> B --> C
|           ^
|           |
+-----------+
```

could be represented as an edge list by `'((A B) (B C) (A C))` and as a successors map by `'((A (B C)) (B (C)) (C ()))`.

1. Write function `successors-map-of-edge-list`, which accepts a graph in edge-list representation and returns a representation of the same graph in successors-map representation.
2. Write function `edge-list-of-successors-map`, which accepts a graph in successors-map representation and returns a representation of the same graph in edge-list representation.

Hint: your new best friend is `let*`.
My estimate of difficulty: hard (there is little conceptual difficulty, but by 105 standards, there is a lot of code)

E. *Merging sorted lists*
Implement function `merge`, which expects two sorted lists of numbers and returns a single sorted list 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)
```

My estimate of difficulty: medium (you will have to think about the structure of a function that consumes *two* lists)

F. *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.
My estimate of difficulty: easy to get a solution that works, medium to avoid unnecessary case analysis

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

1. The operational semantics for uScheme 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
   ```
2. 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.

My estimate of difficulty: medium, because working with formal proofs is tedious

How your work will be evaluated

Programming in µScheme

The criteria we will use to assess your µScheme code are mostly the same as the criteria we used to assess your Impcore core. Be aware that there are a few new criteria.

We will evaluate the correctness of your code by extensive testing.

	Exemplary	Satisfactory	Must improve
Cost	• **New:** Empty lists are distinguished from non-empty lists in constant time.		• **New:** Distinguishing an empty list from a non-empty list might take longer than constant time.
Documentation	• The contract of each function is clear from the function's name, the names of its parameters, and perhaps a one-line comment describing the result. • When names are not enough, each function is documented with	• A function's contract omits some parameters. • A function's documentation mentions every parameter, but does not specify a contract.	• A function is not named after the thing it returns, and the function's documentation does not say what it returns. • A function's documentation includes a narrative description of what happens in the body of the

	a contract that explains what the function returns, in terms of the parameters, which are mentioned by name. • From the name of a function, the names of its parameters, and the accompanying documentation, it is easy to determine how each parameter affects the result. • The contract of each function is written without case analysis, or case analysis was unavoidable. • Documentation appears consistent with the code being described. • **New:** As an alternative to internal documentation, a function's documentation may refer the reader to the problem specification where the function's contract is given.	• A function's documentation includes information that is redundant with the code, e.g., "this function has two parameters." • A function's contract omits some constraints on parameters, e.g., forgetting to say that the contract requires nonnegative parameters. • A function's contract includes a case analysis that could have been avoided, perhaps by letting some behavior go unspecified.	function, instead of a contract that mentions only the parameters and result. • A function's documentation neither specifies a contract nor mentions every parameter. • There are multiple functions that are not part of the specification of the problem, and from looking just at the names of the functions and the names of their parameters, it's hard for us to figure out what the functions do. • A function's contract includes a *redundant* case analysis. • Documentation appears inconsistent with the code being described.
Naming	• Each function is named either with a noun describing the result it returns, or with a verb describing the action it does to its argument. (Or the function is a predicate and is named as suggested below.) • A function that is used as a predicate (for `if` or `while`) has a name that is formed by writing a property followed by a question mark. Examples might include `even?` or `prime?`. (Applies only if the language permits question marks in names.) • *Or*, the code defines no predicates. • In a function definition, the name of each parameter is a noun saying what, in the world of ideas, the parameter represents. • Or the name of a parameter is the name of an entity in the problem statement, or a name from the underlying mathematics.	• Functions' names contain appropriate nouns and verbs, but the names are more complex than needed to convey the function's meaning. • Functions' names contain some suitable nouns and verbs, but they don't convey enough information about what the function returns or does. • A function that is used as a predicate (for `if` or `while`) does not have a name that ends in a question mark. (Applies only if the language permits question marks in names.) • The name of a parameter is a noun phrase formed from multiple words. • Although the name of a parameter is not short and conventional, not an English noun, and not a name from the math or the problem, it is still recognizable---perhaps as an abbreviation or a compound of abbreviations.	• Function's names include verbs that are too generic, like "calculate", "process", "get", "find", or "check" • Auxiliary functions are given names that don't state their contracts, but that instead indicate a vague relationship with another function. Often such names are formed by combining the name of the other function with a suffix such as `aux`, `helper`, `1`, or even `_`. • Course staff cannot identify the connection between a function's name and what it returns or what it does. • The name of a parameter is a compound phrase phrase which could be reduced to a single noun. • The name of some parameter is not recognizable---or at least, course staff cannot figure it out. • **New:** The name of a list

	• Or the name of a parameter is short and conventional. For example, a magnitude or count might be `n` or `m`. An index might be `i`, `j`, or `k`. A pointer might be `p`; a string might be `s`. A variable might be `x`; an expression might be `e`. **New:** A list might be `xs` or `ys`. • **New:** Names that are visible only in a very small scope are short and conventional.	• **New:** Names that are visible only in a very small scope are reasonably short.	parameter is neither a plural noun form nor a conventional name like `xs` or `ys`. • **New:** Long names are used in very small scopes (exception granted for some function parameters). • **New:** Very short names are used with global scope.
Structure	• **New:** The assignment does not use `set`, `while`, `print`, or `begin`. • The code of each function is so clear that, with the help of the function's contract, course staff can easily tell whether the code is correct or incorrect. • There's only as much code as is needed to do the job. • Helper functions are used only where needed. • **New:** Code uses Boolean values `#t` and `#f` where Booleans are called for. • 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. • In every case analysis, all cases are necessary. • **New:** Expressions cannot be made any simpler by application of algebraic laws.	• Course staff have to work to tell whether the code is correct or incorrect. • There's somewhat more code than is needed to do the job. • The code contains unnecessary helper functions, but the course staff find them simple and easy to read. • 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. • In some case analyses, there are cases which are redundant (i.e., the situation is covered by other cases which are also present in the code).	• **New:** Some code uses `set`, `while`, `print`, or `begin` (**No Credit**). • From reading the code, course staff cannot tell whether it is correct or incorrect. • From reading the code, course staff cannot easily tell what it is doing. • There's about twice as much code as is needed to do the job. • The code contains unnecessary helper functions, and the course staff find them complex or and difficult to read. • **New:** 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. • A significant fraction of the case analyses in the code, maybe a third, are redundant. • **New:** Code can be simplified by applying algebraic laws. For example, the code says `(+ x 0)`, but it could say just `x`.
Form	• **New:** 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	• **New:** Code has a few too many blank lines. • **New:** Code needs a few more blank lines to break big blocks into smaller	• **New:** Code wastes scarce vertical space with too many blank lines, block or line comments, or syntactic markers carrying no information.

	of which can be understood as a unit. • All code fits in 80 columns. • The submitted code contains no tab characters. • All code respects the offside rule • Indentation is consistent everywhere. • In Impcore, if a construct spans multiple lines, its closing parenthesis appears at the end of a line, possibly grouped with one or more other closing parentheses. • No code is commented out. • Solution file contains no distracting test cases or print statements.	chunks that course staff can more easily understand. • One or two lines are wider than 80 columns. • The code contains one or two violations of the offside rule • In one or two places, code is not indented in the same way as structurally similar code elsewhere. • Solution file may contain clearly marked test *functions*, but they are never executed. It's easy to read the code without having to look at the test functions.	• **New:** Code preserves vertical space too aggressively, using so few blank lines that a reader suffers from a "wall of text" effect. • **New:** 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. • **New:** 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**). • Three or more lines are wider than 80 columns. • An ASCII tab character lurks somewhere in the submission. • The code contains three or more violations of the offside rule • The code is not indented consistently. • The closing parenthesis of a multi-line construct is followed by more code (or by an open parenthesis) on the same line. • A closing parenthesis appears on a line by itself. • Solution file contains code that has been commented out. • Solution file contains test cases that are run when loaded. • When loaded, solution file prints test results.
Correctness	• **New:** Your *Î¼*Scheme 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.	• **New:** Loading your *Î¼*Scheme into `uscheme` causes an error message (**No Credit**). • Your code fails many tests.

Theory

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
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 *Î¼*Scheme semantics must be used to explain the code.
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.

What to submit

Provide a `README` file, and in it, please do as follows:

- Please tell us with whom you collaborated
- Please tell us what problems you solved
- On each of the dimensions *form*, *documentation*, *naming*, *structure*, *cost*, and *correctness*, please let us know whether you believe your work was Exemplary, Satisfactory, or whether it needs improvement.
- Please tell us how many hours you spent on the assignment

If you want, include any insights you may have had about the problems, but detailed remarks about your solutions are best left to comments in the source code.

When you are ready, run `submit105-scheme` to submit your work, which should include the following files:

- `README`: This documentation file is mandatory.
- `solution.scm`: This source file is mandatory.
- `theory.pdf`: This source file is mandatory; you may prepare it by computer, or you can scan a handwritten solution.