CSC324 Fall 2010

Assignment 1: Basic Principles Of Functional Programming, Parameter Passing, Types.

General coding guidelines:

comments only required for unclear code, but try to change the code instead
don't submit your testing
assume valid uses: error handling is not required
all code must work in CDF's DrScheme or DrRacket 5.0.1.
style counts: refactor repeated code
use local variables for repeated expressions (lookup `let` or `local`), otherwise avoid them
 – comment the expression instead if the purpose of the variable was only to describe the value

(1) **Functional Programming**
Higher Order Functions — Returning Functions.
Closures.

This question is a simple illustration of the closure aspect of Scheme functions.
They can capture and manipulate a dynamically created environment, which persists between calls to the closure.
And then a function returning such functions is like a simple class/constructor.

Expected Maximum Time: 1 Hour.
This includes the non-coding portions not to be submitted.
If you've forgotten some prerequisites (drawing variables, objects, call stack), you might need more time to review.
The concepts are at CSC108 level, to get your feet wet with Scheme.

Preparation: First, make sure you can quickly and easily write a class Toggle in your favourite OO language, with a constructor taking a boolean, a single (private/unexposed) instance variable and a single (public) method, so that calling the method on an instance alternately returns false and true, starting with the boolean passed to the constructor.
Draw memory after two instances have been constructed and have had the method called twice on one instance and once on the other. Use your favourite object and variable diagramming/tracing approach you saw/used/learned in CSC108/148/150/207.
If you got stuck, go to the CS Help Centre, our Office Hours, or contact me ASAP.
Review your lecture notes, and then look through the posted Scheme #1 Part B to see if there's anything added (or you missed) that strengthens your understanding.

Code to write and submit: Write a Scheme function, name it `Toggle`, used as follows:

```
(define toggle1 (Toggle #f))
(toggle1) ;=> #f
(toggle1) ;=> #t
(toggle1) ;=> #f
(define toggle2 (Toggle #t))
(toggle2) ;=> #t
(toggle1) ;=> #f
(toggle2) ;=> #f
```

Design and then coding should take at most about a half hour total — if either is taking much longer, get in touch.

Trace the sample usage, showing the state of memory as in lecture, after each of lines: 1, 2, and 8.
Tracing questions like this (and more complex) could appear on the midterm(s) and/or exam, and for code with an unknown purpose. But the main reason for tracing is to check your understanding of your code, and presumably you already had a rough picture in your mind before writing `Toggle`.
It's beneficial now, for subsequent assignment questions, and to understand the tutorials and lectures. So doing this studying now is a nice time and effort saver, and makes the course more enjoyable.

For similar reasons, draw your source code as a tree, with each operation an internal node, components of an operation as subtrees; check against (or use) the s-expression collapsing and/or shading of DrScheme/Racket. Use your judgement for parts that don't fit this description exactly (e.g. non-operation compound expressions).

(2) **Functional Programming**
Higher Order Functions — Functions as Arguments.
Primitive/Natural Linear Recursion.
No Mutation.
Types
Records/Structs.
Types — Linear Recursive.
Comprehensions.

In this question you write a functional version of list comprehensions (which are appearing more and more in languages, in a special syntax), on a user-defined recursive datatype for lists.

Expected Maximum Time: 1 Hour.
This includes the non-coding portions not to be submitted.
It assumes an understanding of map from A0.
The concepts are at CSC148/150 level, to be mastered in Scheme to build on and get to new material in A2+.

Preparation: Review/lookup the CS terms: unary, binary and ternary function ; predicate.
Review your A0 map.
Write some unit tests for Scheme's map.
Try:

```
; Create a structure/record type named ‘List’, with two fields called ‘first’ and ‘rest’.
; (like a C struct, or Java/Python/C++ class with instance variables first and rest)
(define-struct List (first rest))
; Defines a syntax ’List’ (or in newer Racket versions, a function) for construction,
;  getter functions ‘List-first’ and ‘List-rest’ (check that they’re functions!),
;  and a type predicate ‘List?’.
; (instance variables with getters (and/or setters) are called Properties in OO)
;
; Another struct, with no fields, will model the empty list.
(define-struct (Empty))
(define l (List 3 (List 2 (List 4 (Empty)))))
(List-first l)
(List-rest l)
```

Code to write and submit: Write a ternary Scheme function (naming it) `map-such-that`, taking

a `List` or `Empty` structure as above,

a unary function,

and a unary predicate,

returning a `List/Empty` with each element of the original `List` such that the predicate is non-`#f`, then transformed by calling the unary function. Don't change the order, and don't include any other elements.
Do it with "natural/primitive" recursion (not accumulating tail-recursive), without helper functions.
Don't mutate any variables or values; in particular don't use functions whose name contains "`!`".

If you're stuck or taking a long time:

first trace your algorithm (expressed in your favourite human language)

then write just `map`

write a `such-that` in Scheme or your favourite non-Scheme language

write a `List-length` in Scheme for the `List/Empty` type

Then if you're still stuck or taking a long time, come for help (see (1)).

Trace your code on an example, and draw the source code (or just part of it, if this is getting easy) as a tree (see (1)).

(3) **Functional Programming**
Using Higher Order Functions.

This question starts using the basic CSC324 higher order list functions, avoiding some recursion/iteration.

Expected Maximum Time: 1 Hour.
This includes the non-coding portions not to be submitted.
The recursion in the algorithms is at CSC148/150 level: concentrate on seeing any good uses of map and/or apply.

Preparation: Review map and apply from Lecture.
Review your A0 `subsequences`.
Lookup and practice using Scheme's append.
Write out all rearrangements of

(a)

(b a)

(c b a)

(d c b a)

Write out all rearrangements of

(b c)

(a c)

(a b)

(a b c)

Code to write and submit

(a) Write a Scheme function `subsequences` that takes a (regular) Scheme list, returning a list of all its subsequences.

(b) Write a Scheme function `rearrangements` that takes a (regular) Scheme list, returning a list of all rearrangements of its elements (treat them as sequences, don't treat duplicate elements in the original list specially).

Use map and/or apply where useful. Don't use any other Scheme/Racket higher order functions.
Use only the list and boolean datatypes. Don't mutate. Use only the list functions covered so far, and/or append.
You may write helper functions (higher order or not).

Explain your algorithms in your favourite human language.

Draw them with 'rectangle' diagrams as in lecture: such diagrams might be on the midterm(s)/exam.

Determine which are the first and last subsequences/rearrangements:

created by your code

in the result

Trace a good example. Explain from the algorithm/code, why `subsequences` returns $2^{(\text{length l})}$ elements, and `rearrangements` returns (length l)! elements.

(4) **Functional Programming**

Primitive/Natural Tree Recursion.

Types

Boxes/References/Cells.

Types — Tree Recursive.

Parameter Passing

Call by Reference.

Design Patterns

Interpreter

Domain Specific Languages / Little Languages.

This question starts you working on data described by a recursive data description.

The handling of arithmetic, and the use of boxes to 'return' a value, prepares for Prolog.

While not technically call-by-reference, much of the same effect is achieved by passing a mutable object.

Expected Maximum Time: 1 Hour.

This includes the non-coding portions not to be submitted.

The recursion in the algorithm is at CSC148/150 level, to be mastered in Scheme to build on later.

Preparation: Read about and get comfortable with Racket's `box` type.

Notice they're like a single anonymous/unnamed instance variable, or one-element array/list/vector.

Code to write and submit: Model arithmetic expressions as:

a box containing a number,

or a list with arithmetic expressions as 1st and 3rd element, and 2nd element one of the symbols +, -, * or /.

Write a function '`is`' that takes a box containing a number or void, and an arithmetic expression as defined above.

If the box contains a number, return a boolean for whether it's the number represented by the expression.

If the box contains void, return true and put the value of the arithmetic expression in the box.

Do not use eval. Assume the expression doesn't contain division by zero.

For testing, you can produce the void value by calling `(void)`.

Updating a box is a side-effect, so you might want a sequence of statements in a place where only one is allowed: you can bundle a sequence of statements into one with `begin`.

Draw the algorithm with tree diagrams as in lecture.

Determine which are the first and last boxes to have their values extracted, and the first and last arithmetic operations performed.

(5) **Functional Programming**

Primitive/Natural vs Tail Linear Recursion.

Parameter Passing

Call by Reference.

This question compares natural vs tail recursion, by passing results both in and out via a parameter.
It also continues practicing this style to prepare for Prolog.

Expected Maximum Time: 1 Hour.
This includes the non-coding portions not to be submitted.
The recursion in the algorithm is at CSC148/150 level, to focus on the parameter passing.

Preparation: Read about and get comfortable with Racket boxes (if you didn't do (4) yet).
Write list-length in Scheme in both primitive and tail-recursive accumulating style; trace, and also draw the algorithms. Is the addition done pre-order or post-order?
Write sum-of-squares in Scheme in both styles, that takes a natural number and adds up the sum of the squares from 0 to the number; trace; which are the first and last additions; write out $1^2 + 2^2 + 3^2 + 4^2$ fully-parenthesized to show the order of evaluations, and write them out as Scheme expressions.

Code to write and submit
(Be sure to read the note about begin in (4), in case you need it).

(a) Write `length-primitive` that can be used in the following way:

```
(define Len (box (void)))
(length-primitive '(a b c) Len)
Len ;=> #&3
```

It must recurse, doing all numeric computation on the way out of the recursion (when normally a value would be returned). Do not make any helpers.

(b) Write `length-tail` that can be used in the following way:

```
(define Len (box 0))
(length-tail '(a b c) Len)
Len ;=> #&3
```

It must recurse, doing all numeric computation on the way into the recursion. Do not make any helpers.

(c) Write `reverse-tail` that can be used in the following way:

```
(define L (box '()))
(reverse-tail '(a b c) L)
L ;=> #&(c b a)
```

It must recurse, doing all list building on the way into the recursion. Do not make any helpers.

(d) Write `running-sum-primitive` that can be used in the following way:

```
(define NumberList (box (void)))
(running-sum-primitive '(3 2 4) NumberList)
NumberList ;=> #&(3 5 9) ; i.e. 3, 3+2, 3+2+4
```

It must recurse, doing all work on the box's value on the way out of the recursion. Do not make any helpers.

Draw and trace the algorithms.
Write out all the additions done by `running-sum-primitive`, in the order they're executed.
Determine why reverse is natural as tail recursion, running-sum is natural as primitive recursion.