

CSC324 Spring 2007  
St. George Campus

February 15, 2007  
University of Toronto

Homework Assignment #3  
**Due: Tuesday, February 27, 2007, by 1pm**

---

### Silent Policy

A silent policy will take effect 24 hours before this assignment is due. This means that no question about this assignment will be answered, whether it is asked on the newsgroup, by email, or in person.

### Total Marks

There are 70 marks available in this assignment. This assignment represents 10% of the course grade.

### Handing in this Assignment

*What to hand in on paper:*

Please use an *unsealed* letter-sized envelope, attaching the cover page provided to the front. Note that without a properly completed and signed cover page your assignment will not be marked. Please place your solutions to all questions inside the envelope. You must hand in this part of the assignment in the CSC324 drop box in the Bahen Computer Lab, BA2220.

*What to hand in electronically:*

In addition to your paper submission, you must submit your code electronically. Use the file names specified in the questions. To submit the files electronically, use the CDF secure Web site:

<https://www.cdf.utoronto.ca/students>

*Warning:* marks will be deducted for incorrect submission. Note that if the code submitted electronically differs from the code submitted on paper, we will only mark the electronically submitted version (if, in such a case, you put comments, etc. only on paper, we will mark the question as if no comments, etc. were provided).

Since we will test your code electronically, you must:

- *make certain that your code runs on CDF,*
- use the exact file names specified,
- use the exact procedure names and argument(s) (including the order of arguments) specified,
- not load any files,
- not display anything but the procedure output (no text messages to the user, fancy formatting, etc.)

### Clarification Page and Newsgroup

Important corrections (hopefully few or none) and clarifications to the assignment will be posted on the Assignment 3 Clarification page, linked from the CSC324 home page. You are also responsible for monitoring the CSC324 newsgroup.

### How you will be marked

Code will be marked with respect to correctness, style, and documentation.

# Assignment #3

---

**Due Date:** This assignment is due on Tuesday, February 27, 2007 at 1pm.

**Please include this cover sheet, when handing in your paper assignment.**

**Last Name:**

**First Name:**

**Email:**

**CDF Login:**

**Student Number:**

**Grace days used to date:**

**Grace days used for this assignment:**

**Date and Time you are handing this in:**

All answers are my own, written in isolation, without help from others. This submission is in accordance with the University of Toronto Code of Behaviour on Academic Matters (<http://www.artsandscience.utoronto.ca/ofr/calendar/rules.htm#behaviour>).

Signature \_\_\_\_\_

**Question 1.** (20 marks)

Procedure (**conditional f g h lst**) applies a predicate procedure **f** to every element of the list **lst**. If **f** is true of every element in **lst**, then it applies **g** to every element and returns the resulting list. If **f** is false of every element in **lst**, then it applies **h** to every element and returns the resulting list. Otherwise, the list **lst** is returned. Assume that **f**, **g**, and **h** are applicable to every element of **lst**. For example,

```
1 ]=> (conditional null? length reverse '( ()()() ))
;Value: (0 0 0)
```

```
1 ]=> (conditional null? length reverse '( (1 2) (3 4) () ))
;Value: ((1 2) (3 4) ())
```

```
1 ]=> (conditional null? length reverse '( (1 2) (3 4) (5 6 7) ))
;Value: ((2 1) (4 3) (7 6 5))
```

**a. (5 marks)** Implement (**conditional f g h lst**). Whenever you have a choice between using recursion and using higher-order procedures, use recursion. You may use helper procedures. Filename: *a3-1-a.scm*.

**b. (5 marks)** Re-implement (**conditional f g h lst**). Do not use recursion in your solution. Do not use any helper procedures. Use the built-in procedure `eval`. Filename: *a3-1-b.scm*.

**c. (5 marks)** Re-implement (**conditional f g h lst**). Do not use recursion in your solution. Do not use any helper procedures. Do not use `eval`. Use the built-in procedure `apply`. Filename: *a3-1-c.scm*.

**d. (5 marks)** Re-implement (**conditional f g h lst**). Do not use recursion in your solution. Do not use any helper procedures. Do not use `eval`. Do not use `apply`. Use the built-in procedure `reduce`. Filename: *a3-1-d.scm*.

**Question 2.** (10 marks)

Write a procedure (**compose f n**) that, given a unary procedure **f** and a natural number **n**, returns a procedure that applies **f** to its argument **n** times. 0 is a natural number. A procedure applied 0 times is an identity procedure (a unary procedure which returns its argument). Do not use any helper procedures. Filename: *a3-2.scm*. For example,

```
1 ]=> ((compose list 0) 'a)
;Value: a
```

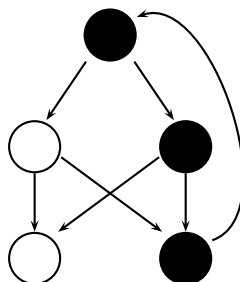
```
1 ]=> ((compose list 1) 'a)
;Value: (a)
```

```
1 ]=> ((compose list 2) 'a)
;Value: ((a))
```

```
1 ]=> ((compose list 3) 'a)
;Value: (((a)))
```

**Question 3.** (20 marks)

A directed graph is a finite set of *nodes* with directed *edges* that point from one node to another. Here is a picture of a directed graph with five nodes:



Directed graphs can be represented in Scheme using an *adjacency representation*. This can be a list of 3-element lists, for example, in which each node is represented by three items. The first item names the node, the second item is some data associated with that node, and the third item is a list of names of all of the nodes reachable by one edge from the node named in the first item. The following is a representation of the directed graph depicted above:

```

((1 black (2 3))
 (2 white (4 5))
 (3 black (4 5))
 (4 white ())
 (5 black (1)))

```

where 1 names the topmost node. The data (second item) may not necessarily be colours. In fact, because the second item could be a list, you can store as much data there as you like. The empty list, `()`, corresponds to the directed graph with no nodes.

**a. (10 marks)** Write a procedure (**insert data next prev g**) that returns a directed graph **g** with a new node inserted. This node contains data **data**, has edges to nodes listed in list **next** and edges from nodes listed in list **prev**. If the largest label of the nodes is  $n$ , the new node should be labeled  $n + 1$ . You can assume that any number listed in **next** or **prev** corresponds to a node in **g**. For example, if **g** is a directed graph above, then

```
(insert 'black '(1) '(4 5) g)
```

returns the graph

```

((1 black (2 3))
 (2 white (4 5))
 (3 black (4 5))
 (4 white (6))
 (5 black (1 6))
 (6 black (1)))

```

Do not use recursion. Do not use helper procedures.

**b. (10 marks)** Write a procedure (**degree g**) that returns the degree of graph **g** defined as follows. The degree of a node is the number of outgoing edges of that node. The degree of a graph is the maximum of degrees of its nodes. The degree of a graph with no nodes is 0. Do not use recursion. Do not use helper procedures. You may use the built-in procedure **max**.

Filename (one file for both parts): *a3-3.scm*.

**Question 4.** (10 marks)

Recall that the binomial distribution is the discrete probability distribution of the number of successes in a sequence of  $n$  independent yes/no experiments, each of which yields success with probability  $p$ . The probability of getting exactly  $k$  successes is given by the probability mass function:

$$f(k, n, p) = \binom{n}{k} p^k (1-p)^{n-k}, \text{ where } \binom{n}{k} = \frac{n!}{k!(n-k)!}$$

Write a procedure **binomial(k,n,p)** that implements the probability mass function of the binomial distribution. The number of recursive calls your program makes should be at most  $n + 1$ . You may use the built-in procedure `expt`. Do not use helper procedures. Filename: *a3-4.scm*.

**Question 5.** (10 marks)

In large-scale programming projects, it is important to document properties that are believed to hold of their sub-components. These properties not only document the intended behaviour of a component, but can establish conditions on the intended context of its invocation to ensure its portability. Formally proving these properties is thus a way of verifying the correctness of a software component in all of its potential invocations. Test suites, while convenient, at best improve our confidence in a component's correctness, by evaluating it on a finite collection of possible inputs.

In the case of functional programming, functional procedures can naturally be thought of as the components, and the properties that we need to prove correct often consist of algebraic equations that establish certain invariants over their possible invocations. Below, we will be looking at invariants that pertain to lists and their lengths.

*Example:* Consider the following program:

```
(define append
  (lambda (X Y)
    (if (null? X) Y
        (cons (car X) (append (cdr X) Y))))

(define length
  (lambda (X)
    (if (null? X) 0
        (+ 1 (length (cdr X)))))
```

We now prove that for *all* (proper) lists,  $X$  and  $Y$ , the following equation holds:

$$(\text{length } (\text{append } X \ Y)) = (+ (\text{length } X) (\text{length } Y)).$$

This means that  $(\text{length } (\text{append } X \ Y))$  always evaluates to the sum of the value of  $(\text{length } X)$  and the value of  $(\text{length } Y)$ . If this equation holds, then even if we have made a mistake in defining `length` and `append`, our definitions at least *act like* `length` and `append` in an important respect.

Because these procedures are recursively defined, this claim admits a fairly straightforward inductive proof. With reference to four basic properties (facts) that we can read directly from the source code:

1.  $(\text{append } () \ Y) = Y$
2.  $(\text{append } (\text{cons } E \ L) \ Y) = (\text{cons } E \ (\text{append } L \ Y))$
3.  $(\text{length } ()) = 0$
4.  $(\text{length } (\text{cons } E \ L)) = (+ 1 (\text{length } L))$

we can formulate the following proof:

Proof by mathematical induction on length of X:

Case:  $X = ()$ :

```
(length (append X Y))
= (length (append () Y)) [case]
= (length Y) [1]
= 0 + (length Y) [arith]
= (length ()) + (length Y) [3]
= (length X) + (length Y) [case]
```

Case:  $X = (\text{cons } E \text{ } L)$

Inductive hypothesis (IH):  $(\text{length } (\text{append } L \text{ } Y)) = (+ (\text{length } L) (\text{length } Y))$

```
(length (append X Y))
= (length (append (cons E L) Y)) [case]
= (length (cons E (append (L Y)))) [2]
= 1 + (length (append L Y)) [4]
= 1 + (length L) + (length Y) [IH]
= (length (cons E L)) + (length Y) [4]
= (length X) + (length Y) [case]
```

Notice that the `if` statement in the definition of `append` corresponds to the two cases in the proof: one for the “then” branch, and one for the “else” branch. Each step in the proof has a justification, which is one of the following:

- `case` — uses a fact assumed to be true in the current case,
- `arith` — uses a basic fact of arithmetic (you can assume that `+` works correctly),
- `IH` — uses the inductive hypothesis,
- $n$  — in reference to one of the above facts, where  $1 \leq n \leq 4$ .

Now it’s your turn! Consider the following two procedures:

```
(define sum
  (lambda (lst)
    (if (null? lst) 0
        (+ (car lst) (sum (cdr lst))))))
```

```
(define mult
  (lambda (c lst)
    (if (null? lst) ()
        (cons (* c (car lst)) (mult c (cdr lst))))))
```

Your task is to prove that, if `lst` is a list of integers and `k` is an integer, then:

$$(\text{sum } (\text{mult } k \text{ } \text{lst})) = (* k (\text{sum } \text{lst})) \quad (1)$$

**a. (4 marks)** Examine the procedures `sum` and `mult`. From inspection of the code, write four facts (similar to the ones in the above example) that you will use in your proof.

**b. (6 marks)** Prove (1) using induction. Each step of your proof must be justified with `case`, `arith`, `IH` or one of the four facts from part a.