Computer Science 324 — 4 October, 2004
St. George Campus — University of Toronto

Homework Assignment #2

Due: Wednesday, 20 October, 2004, by 5pm

No submissions will be accepted after 5pm Wednesday, 20 October — no exceptions, no grace days

---

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 85 marks available in this assignment, but it will be marked out of 75 marks. In other words, if you get 75 marks, you get 100%.

Handing in this Assignment

What to hand in on paper: Please use an **unsealed** 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. Put inside:

1. A printout of your solutions for all questions.
2. For each procedure that you write, you must describe the testing strategy that you used. This should include a table listing the test cases that you designed for your procedure, what output your procedure returned, and an explanation of why the test case and output are significant in verifying the correctness of the procedure. Read the following for a discussion of good testing practices:

   `http://www.cs.toronto.edu/~gpenn/csc324/software.testing.pdf`

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. The procedures (including helper procedures) for each part of each individual question are to be submitted separately, with the filename as stated in the question. To submit these files electronically, use the CDF secure website:
`https://www.cdf.utoronto.ca/students`
or use the CDF **submit** command. Type **man submit** for more information.

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 procedure names and argument(s) (including the order of arguments) specified,
- use the exact file names specified in the questions,
- not use `load` in any of your submitted files,
- not display anything but the function output (no text messages to the user, fancy formatting, etc. — just what is in the assignment handout).

Other Information You must read the requirements for code and marking information on the following web page: `http://www.cs.toronto.edu/~anya/a2guidelines.html` This document constitutes part of the **official requirements** for this assignment.

FAQ Page and Newsgroup Important corrections (hopefully few or none) and clarifications to the assignment will be posted on the Assignment 2 FAQ page, linked from your section's CSC324 home page. You are also responsible for monitoring the CSC324 newsgroup.

Question 1. (1 mark) Fill out the cover sheet correctly.

Question 2. (9 marks)
a. [4 marks] Define a procedure **mult-num** that takes a list as input and returns the product of all numbers in the list. List elements that are not numbers should be ignored. If the input list contains no numbers, then 1 should be returned. Do not use the Scheme built-in `reduce`. For example:

```
]=> (mult-num '(1 2 3 f 4))
;Value: 24
]=> (mult-num '(14 () (100) 2.5))
;Value: 35.
```

Call this file **mult-num.scm**.

b. [5 marks] Define a procedure **min-max** that takes a non-empty list of numbers as input and returns a list of two elements: the smallest number that appears in the input list and the largest number that appears in the input list. Example:

```
]=> (min-max  '(-10 0 3 -4))
;Value: (-10 3)
```

Call this file **min-max.scm**.

Question 3. (10 marks)
a. [3 marks] Define a procedure **select-even** that takes a list of numbers as input and returns the list that contains its even numbers, in the same order as they appear in the input list. You can use the built-in predicate **even?**. Do not use **select** from part (b). Example:

```
]=> (select-even '(1 2 3 4))
;Value: (2 4)
```

Call this file **select-even.scm**.

b. [5 marks] Define a procedure **select** that takes a unary predicate and a list as input and returns the list containing all elements from the input list for which the predicate is true, in the same order as they appear in the input list. You can assume that the input predicate is applicable to every element in the input list. Example:

```
]=> (select null? '( () (1 2 3) ))
;Value: ( () )
```

Call this file **select.scm**.

c. [2 marks] Using **select**, define **select-three**, which takes a list of numbers as input and returns the list of elements from the input list that are divisible by 3, in the same order as they appear in the input list. Example:

```
]=> (select-three `(1 3 0 -75 4))
;Value: (3 0 -75)
```

The first argument in your call to **select** must be an anonymous procedure.
Call this file **select-three.scm**. Do not include the definition of **select** in this file.

Question 4. (5 marks) Find a non-recursive procedure that will not terminate when applied to some (but not necessarily all) inputs. Your procedure should not call any built-in recursive procedures either. Also provide one input that causes it not to terminate. Hint: Are there functions that can be applied to themselves?
Do not submit anything electronically for this question.

Question 5. (10 marks) **Revised October 9, 2004.** **"Lazy Evaluation"**. Normally, when Scheme evaluates a procedure that returns a list, it attempts to build the entire list, and then, if the procedure is called from the read-eval-print loop, to display the entire list. But there are circumstances in which it is conceptually more elegant to imagine the output of a procedure as an infinite list, the elements of which are referenced only as they are needed. This procedure, for example, creates a list of all of the natural numbers greater than or equal to n, but it never terminates for any n, because there are infinitely many of them:

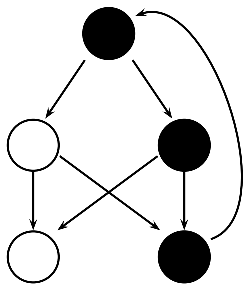
```
(define up-set (lambda (n) (cons n (up-set (+ n 1)))))
```

Define a function **lazy** which, given a number n and a unary successor procedure, returns a function that enumerates the infinite list consisting of n and its successors, one element at a time. Example:

```
]=> (define up-set
       (lambda (n)
          (lazy  n  (lambda (x) (+ x 1)))))
;Value: up-set

]=> (define f (up-set 10))
;Value: f

]=> (f ())
;Value: 10

]=> (f ())
;Value: 11

]=> (f ())
;Value: 12

...
```

You may use `set!` in your solution.
Call this file **lazy.scm**.

Question 6. (20 marks) **Directed Graphs**. 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:



Some configurations of edges can form a *cycle*, a path from a node to itself that is at least one edge long. There is a cycle among the three black nodes in the above picture (actually, there are two cycles — the other passes through the upper white node). Notice that cycles must follow the direction indicated by their edges. Multiple edges can point to a single node without creating a cycle, as is the case with the bottom left node in the above picture.

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. It has no cycles.

a. [15 marks] Define a predicate, **cyclic?**, which, given such an adjacency representation of a directed graph, indicates whether the directed graph has a cycle. You do not need to exhibit a cycle, if there is one; you should give only a true or false answer.
Call this file **digraph.scm**.

b. [5 marks] What data did you need to store at each node as you traversed the graph? Use as little as possible.
Do not submit anything electronically for this question.

Question 7. (20 marks) **Software Verification**. 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 and sample executions:

```
(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))))))

]=> (append '(1 2 3) '(4 5))
;Value: (1 2 3 4 5)

]=> (length '(1 2 3))
;Value: 3
```

We now prove that for *all* lists, `X` and `Y`, the following equation holds:

`(length (append X Y)) = (+ (length X) (length Y))`.

This means that `(length (append X Y))` always evaluates to the sum of the value of `(length X)` and the value of `(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 that we can read directly from the source code:

1. `(append () Y) = Y`
2. `(append (cons E L) Y)` = `(cons E (append L Y))`
3. `(length ())` = 0
4. `(length (cons E L))` = `(+ 1 (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 = (cons E L)
Inductive hypothesis (IH): (length (append L Y)) = (+ (length L) (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 functions:

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

(define increment-list
   (lambda (L)
       (if (null? L) L
           (cons (+ (car L) 1) (increment-list (cdr L))))))
```

Using the following six facts about these functions:

1. `(sum ())` = 0
2. `(sum (cons N L))` = `N` + `(sum L)`
3. `(length ())` = 0
4. `(length (cons N L))` = `(+ 1 (length L)`
5. `(increment-list ())` = `()`
6. `(increment-list (cons N L))` = `(cons (+ N 1) (increment-list L))`,

prove by induction that, if `L` is a list of integers, then:

`(sum (increment-list L)) = (+ (sum L) (length L))`.

Each step of your proof must be justified with `case`, `arith`, `IH` or one of the above six facts, just as in the example above.
Do not submit anything electronically for this question.

Question 8. (10 marks) In this exercise, you will implement a calculator for infixed arithmetic expressions with operator precedence. But you will do so in a special way: your calculator will be implemented as a tail-recursive procedure which makes a single linear pass through the input expression. In each call, your procedure will pass to the next call a *continuation*, which represents the state of the (possibly unfinished) computation. Because you are using Scheme, your continuation for unfinished computations can (and should) be a lambda-expression that is waiting for more input to finish.

Here is some code and a few examples to get you started:

```
(define calculate
   (lambda expression
       (reduce-cont calc-initiate expression 'calc-terminate)))

(define reduce-cont
   (lambda (continuation list nularg)
       (if (null? list) (continuation nularg)
           (reduce-cont (continuation (car list)) (cdr list) nularg))))

(define id (lambda (x) x))

(define paren list)

; Operator op1 precedes operator op2 in the standard order of
; arithmetic operations
(define precedes?
   (lambda (op1 op2)
         (member op2 (cdr (member op1 (list * / + -))))))

]=> (calculate 3 / 4 + 5)
;Value: 23/4

]=> (calculate 3 + 4 / 5)
;Value: 19/5

]=> (calculate (paren 3 + 4) / 5)
;Value: 7/5
```

Notice that your continuation never actually gets to peek at the entire arithmetic expression — only the `car`, or next token. The implementation of `reduce-cont` above assumes that your continuation will be a unary function. `calc-initiate` initializes your continuation at the beginning of computation, and the value `calc-terminate` is used as a signal to your continuation that there will be no more input. `paren` is used in expressions to tell Scheme that these are your parentheses, and not Scheme's.

Hint: Because of the infix ordering of the expressions, the input tokens to your calculator alternate between numbers/values and operators. Generalize `calc-initiate` to a more general procedure, `calc-continue`, which iteratively looks for an expression-followed-by-operator sequence. Then define `calc-initiate` using `calc-continue` and `id`. In this case, `calc-continue` will also be recursive, calling itself tail-recursively to consume more input, and calling `calculate` to evaluate parenthesized expressions.

a. [3 marks] Start by implementing a calculator that handles only addition and subtraction. No operator precedence is involved here, so you will not need `precedes?`. You can apply operators to their successive arguments in the order that you encounter them. This is equivalent to taking `+` and `-` to be of the same precedence and left-associative.
Call this file **calca.scm**.

b. [5 marks] Now add multiplication and division. You must use the following order of arithmetic operations: `*`, `/`, `+`, `-`. This differs from the standard order, in which `*` and `/` have the same precedence, and `+` and `-` have the same precedence. If you did not use the hint for part (a), you definitely should now — you'll have to look at each pair of successive operator to determine which one has higher precedence.
Call this file **calcb.scm**. Do not include procedures from **calca.scm** in this file.

c. [2 marks] Now add support for parenthesized sub-expressions. You will not see `paren` in your input — Scheme evaluates that away, so all that remains is an ordinary list.
Call this file **calcc.scm**. Do not include procedures from **calca.scm** and **calcb.scm** in this file.