Computer Science 320 (Spring Term, 2006)
Concepts of Programming Languages
Problem Set 2: Higher-Order Functions

Out: Friday, January 27. 2006
Due: ***5:00 PM*** on Friday, February 3, 2006

There are 5 problems in this set, each worth as noted. The total number of points is 100. The harder problems are marked with a single * (average difficulty) or two ** (higher-than-average difficulty).

A hallmark of functional programming is the ease with which it allows the processing of *higher-order functions*. A higher-order function, sometimes called a *functional*, is a function which takes a function as argument and/or returns a function as result. This concept is similar to *callbacks* in C/C++ or *reflection* in Java.

Even if you did not call them by this name, you most cetainly encountered higher-order functions already. Examples of higher-order functions abound in calculus; e.g., the differential operator is a higher-order function which takes a function as argument and returns another function (the derivative) as result. The Scheme procedure `map` is an example of a higher-order function because its first argument is itself a function.

It is relatively simple, in functional programming, to make higher-order functions *first-class*. First-class elements in a programming language are those that may be:

- named by variables,
- passed as arguments to procedures,
- returned as the results of procedures, and
- included in data structures.

In an imperative language such as Fortran or C or Pascal, we take for granted that integer and boolean values, among others, are so manipulated. However, whatever else these languages are convenient for, they do not make it easy on programmers to use higher-order functions in the same way.

In this assignment you will practice programming with higher-order functions and learn some of their advantages, in particular, their suitability for producing transparent code for relatively complex computational tasks.

Problem 1 (30 points) The *composition* of two functions f after g results in a function $(f \circ g)(x) = f(g(x))$. We may define a procedure `compose` that implements function composition as the folowing:

```
(define (compose f g) (lambda (x) (f (g x))))
```

So, for example, if `inc` is a procedure that adds 1 to its argument, and `square` is a procedure that returns x^2 for an argument x, then `((compose square inc) 6)` would evaluate to `49`.

Please do the following exercises from the book in three parts.

1. Provide a procedure `double` as described in exercise 1.41, page 77, in the [A&S] book; write `double` in terms of `compose`.
2. Implement a procedure `repeated` as described in exercise 1.43, page 77, in the [A&S] book.
3. Exercise 1.44, page 77, in the [A&S] book. Note that procedure `smooth` should take two arguments in the following order: the function being smoothed and dx. The procedure implementing *n-fold smoothed function* should be called `n-smooth` and should take three arguments in the following order: the function being smoothed, dx, and a number of times the function is smoothed.

* **Problem 2** (20 points) Many useful list operators such as `map`, `append`, `length`, and `reverse` can be implemented in terms of two primitives, `fold-right` and `fold-left`. Please do the following exercises from the book in two parts.

1. Exercise 2.33, page 119, in the [A&S] book.
2. Exercise 2.39, page 122, in the [A&S] book.

Note that `fold-right` is also called `accumulate` in the book.

Injective, surjective, bijective

For later problems in this assignment, we recall some standard notions about functions.

Let $\mathbb{N}$ be the set of natural numbers (the non-negative integers). We restrict attention to unary (one-argument) total functions from some subset $A \subseteq \mathbb{N}$ (the *domain*) to some subset $B \subseteq \mathbb{N}$ (the *range*). The subsets A and B are not necessarily proper. A function $f : A \to B$ is said to be *injective* (or also *one-one*) if it satisfies the condition that $f(x_1) = f(x_2)$ implies $x_1 = x_2$, for all $x_1, x_2 \in A$. For every injective function $f : A \to B$, there is another function $g : B \to A$ (not necessarily unique), called the *inverse* of f such that $g(f(x)) = x$ for every $x \in A$. If g is unique, it is typically denoted f^{-1}.

If $f : A \to B$ is an injective function and $g : B \to A$ is an inverse of f, it is not necessarily the case that $f(g(y)) = y$ for every $y \in B$ unless f is *surjective* (or also *onto*) which means that for every $y \in B$ there is $x \in A$ such that $f(x) = y$. If f is both injective and surjective, its inverse g is uniquely defined and can be denoted f^{-1}.

A function $f : A \to B$ which is both injective and surjective is called *bijective*.

Representing sets with functions

There are different ways of representing sets in programming. Three different ways are discussed in Section 2.3.3 in the [A&S] book (*unordered lists*, *ordered lists*, and *binary trees*). Another altogether different way of representing a set A is by means of its *characteristic function*, which is a function $\chi : A \to \{\text{true}, \text{false}\}$ such that:

$$\chi(x) = \begin{cases} \text{true} & \text{if } x \in A, \\ \text{false} & \text{if } x \notin A. \end{cases}$$

The characteristic function of the set $\{10, 12, 14\}$ can be programmed in Scheme as:

```
(lambda (x) (or (= x 10) (= x 12) (= x 14)))
```

The characteristic function of all even natural numbers can be implemented as:

```
(lambda (x) (= (remainder x 2) 0))
```

Another way of representing a set A is by means of a *generating function* $g : \mathbb{N} \to A$ which must be surjective and, if at all useful, programmable. The generating function of all even natural numbers can be programmed as:

```
(lambda (x) (* 2 x))
```

Other useful sets for which we can program generating functions include the set of all prime numbers, the set of Fibonacci numbers, and many others. For the set of prime numbers, its generating function must satisfy:

$\mathsf{primes} : \mathbb{N} \to \{\text{all primes} \geqslant 2\}$ such that
$\mathsf{primes}(0) = 2,\ \mathsf{primes}(1) = 3,\ \mathsf{primes}(2) = 5,\ \mathsf{primes}(3) = 7,$ etc.

Curried and uncurried

We can program the addition function which consumes a pair of numbers and returns their sum as follows:

```
(define add-uncurried
    (lambda (x y) (+ x y)))
```

We call it "uncurried" in contrast to the following implementation:

```
(define add-curried
    (lambda (x)
       (lambda (y) (+ x y))))
```

The curried version of addition can be fed each of its 2 arguments separately. For example, `(add-curried 5)` is perfectly valid, whereas `(add-uncurried 5)` will cause an error. Note that the evaluation of `(add-curried 5)` returns a function, namely `(lambda (y) (+ 5 y))`.

Problem 3 (10 points) In two parts:

1. Define a Scheme procedure `curry` which consumes an uncurried function `f` and returns its curried version. Assume `f` takes a pair of arguments, i.e., `f` is invoked on arguments `x` and `y` by writing `(f x y)`.

2. Define a Scheme procedure `uncurry` which consumes a curried function `g` and returns its uncurried version. Assume that `g` takes two arguments in sequence, i.e., to invoke `g` on argument `x` and then on argument `y` we write `((g x) y)`.

* **Problem 4** (20 points) In two parts:

1. Define a Scheme procedure `from-char-to-gen` which consumes the characteristic function `f` of some set and returns the generating function of the same set. Assume `f` is the characteristic function of an infinite $A \subseteq \mathbb{N}$. The desired generating function of A is surjective from $\mathbb{N}$ to A.

2. Define a Scheme procedure `from-gen-to-char` which consumes the generating function `g` of some set and returns the characteristic function of the same set. Assume `g` generates an infinite set of natural numbers in strictly increasing order.

** **Problem 5** (20 points) Define a Scheme procedure `inv` which takes 3 arguments: a bijective function `f`: $A \to B$, where $A, B \subseteq \mathbb{N}$, together with the characteristic function `dom` of the domain A and the characteristic function `ran` of the range B. Evaluation of `(inv f dom ran)` should return the inverse function of `f`. The sets A and B can be finite or infinite (and, because `f` is a bijection, they must have the same cardinality).