Wellesley College ◊ CS251 Programming Languages ◊ Spring, 2000

PROBLEM SET 2
Due Friday, February 25, 2000

Notes:

This is the final version of Problem Set 2. The due date has been extended from Friday, February 18 to Friday February 25. The problem set is now worth 200 points rather than the usual 100 points.

This problem set contains 5 mandatory problems and 3 (optional) extra credit problems
- Problem 1 is the same as posted previously.
- Problem 2 is the same as posted previously except:
  1. the description of bits (part g) notes that (bits 0) is a special case;
  2. the previous definition of fast-expt had a bug: (* power ...) should have been (* base ...). This has been corrected;
  3. there are now example applications of fast-expt in part h;
  4. there are five new parts (i, j, k, l, and m);
  5. The previous definition of foldl has been modified to be consistent with the definition presented in class. In particular, (op init (car lst)) in the original PS2 has now been replaced by (op (car lst) init);
  6. the higher-order list operations map2 and map-append have been removed from Appendix A (if you need them, they are easy to define in terms of the other operators).
- Problems 3 through 5 are new.
- There are 2 extra credit problems worth 60 points of extra credit.

Reading: First-Class Functions handout; SICP 1.3, 2.2.3 – 2.2.4.

Problem 1 [20]: Using the Substitution Model to Reason About Higher-Order Functions

Consider the following definitions:

```
(define apply-to-5 (lambda (f) (f 5)))
(define create-subtracter (lambda (n) (lambda (x) (- x n))))
```

Use the substitution model to show the evaluation of the following expressions:

a. (apply-to-5 (create-subtracter 2))
b. (apply-to-5 create-subtracter)
c. ((apply-to-5 create-subtracter) 2)
d. (create-subtracter apply-to-5)
Problem 2 [100]: Aggregate Data Paradigm

Implement the following functions in terms of the higher-order list operations in Appendix A. (These can be found on-line in the cs251 download folder in ps2/higher-order-list-ops.scm) You should not use recursion in any of your definitions, though you may want to define some auxiliary functions. You will recognize some of these functions from PS1.

a [5]  (append lst1 lst2)
   Return a list containing all the elements of lst1 followed by the elements of lst2.

   > (append '(1 2 3) '(4 5 6))
   (1 2 3 4 5 6)

   > (append '(((a b) (c d)) '((e (f g) h)))
   ((a b) (c d) e (f g) h)

b [5]  (reverse lst)
   Return a list containing the elements of lst in reverse order.

   > (reverse '(a b c d))
   (d c b a)

   > (reverse '(((a b) (c d)))
   ((c d) (a b))

   > (reverse '())
   ()

c [5]  (unzip lst)
   Assume that lst is a list of length len whose ith element is a list of the form (ai bi). Return a list of the form (lst1 lst2) where lst1 and lst2 are length len lists whose ith elements are ai and bi, respectively.

   > (unzip '(((1 a) (2 b) (3 c)))
   ((1 2 3) (a b c))

   > (unzip '(((1 a)))
   ((1) (a))

   > (unzip '())
   () ()
d [6] (sum-multiples-of-3-or-5 m n)

Assume \( m \) and \( n \) are integers. Returns the sum of all integers from \( m \) up to \( n \) (inclusive) that are multiples of 3 and/or 5.

\[
\begin{align*}
> & \ (\text{sum-multiples-of-3-or-5} \ 0 \ 10) \\
& \ 33 \ ; \ 3 + 5 + 6 + 9 + 10 \\
> & \ (\text{sum-multiples-of-3-or-5} \ -9 \ 12) \\
& \ 22 \\
> & \ (\text{sum-multiples-of-3-or-5} \ 18 \ 18) \\
& \ 18 \\
> & \ (\text{sum-multiples-of-3-or-5} \ 10 \ 0) \\
& \ 0 \ ; \ The \ range \ "10 \ up \ to \ 0" \ is \ empty.
\end{align*}
\]

e [7] (all-contain-multiple? n intss)

Assume that \( n \) is an integer and \( \text{intss} \) is a list of lists of integers. Returns \#t if each list of integers in \( \text{intss} \) contains at least one integer that is a multiple of \( n \); returns \#f if some list of integers in \( \text{intss} \) does not contain a multiple of \( n \). (Note that some Scheme interpreters use the empty list () to stand for \#f.)

\[
\begin{align*}
> & \ (\text{all-contain-multiple?} \ 5 \ '((17 \ 10 \ 12) \ (25) \ (3 \ 7 \ 5))) \\
& \ #t \\
> & \ (\text{all-contain-multiple?} \ 3 \ '((17 \ 10 \ 12) \ (25) \ (3 \ 7 \ 5))) \\
& \ #f \\
> & \ (\text{all-contain-multiple?} \ 3 \ '()) \\
& \ #t
\end{align*}
\]

f [8] (cartesian-product lst1 lst2)

Returns a list of all duples \((a \ b)\) where \( a \) ranges over the elements of \( \text{lst1} \) and \( b \) ranges over the elements of \( \text{lst2} \). The duples should be sorted first by the \( a \) entry (relative to the order in \( \text{lst1} \)) and then by the \( b \) entry (relative to the order in \( \text{lst2} \)).

\[
\begin{align*}
> & \ (\text{cartesian-product} \ '(1 \ 2) \ '(a \ b \ c)) \\
& \ '((1 \ a) \ (1 \ b) \ (1 \ c) \ (2 \ a) \ (2 \ b) \ (2 \ c)) \\
> & \ (\text{cartesian-product} \ '(2 \ 1) \ '(c \ a \ b)) \\
& \ '((2 \ c) \ (2 \ a) \ (2 \ b) \ (1 \ c) \ (1 \ a) \ (1 \ b)) \\
> & \ (\text{cartesian-product} \ '(c \ a \ b) \ '(2 \ 1)) \\
& \ '((c \ 2) \ (c \ 1) \ (a \ 2) \ (a \ 1) \ (b \ 2) \ (b \ 1)) \\
> & \ (\text{cartesian-product} \ '(1) \ '(a)) \\
& \ '((1 \ a)) \\
> & \ (\text{cartesian-product} \ '() \ '(a \ b \ c)) \\
& \ ()
\end{align*}
\]
g [10]  (bits int)

Assume int is a non-negative integer. Returns a list of bits (i.e, binary digits -- 0s and 1s) in the binary representation of int, where the bits are ordered from most significant digit to least significant.

> (bits 0)  
  (0)

> (bits 1)  
  (1)

> (bits 2)  
  (1 0)

> (bits 3)  
  (1 1)

> (bits 10)  
  (1 0 1 0)

> (bits 20)  
  (1 0 1 0 0)

> (bits 26)  
  (1 1 0 1 0)

> (bits 42)  
  (1 0 1 0 1 0)

> (bits 52)  
  (1 1 0 1 0 0)

**Hint:** Consider the sequence of numbers obtained by successive integer division by 2 (using Scheme's quotient function) until reaching a number less than 1, as shown in the following examples. Do you see a relationship to bits?

0: ()  ; Need a special case for 0.
1: (1)
2: (2 1)
3: (3 1)
10: (10 5 2 1)
20: (20 10 5 2 1)
26: (26 13 6 3 1)
42: (42 21 10 5 2 1)
52: (52 26 13 6 3 1)
The fast exponentiation procedure fast-expt can be defined recursively as follows:

```scheme
;; Assume power is a non-negative integer.
(define fast-expt
  (lambda (base power)
    (if (= power 0)
      1
      (if (even? power)
        (square (fast-expt base (/ power 2)))
        (* base (square (fast-expt base (quotient power 2)))))
    )))
```

Give a non-recursive definition of `fast-expt` using the higher-order list operators. *Hint:* use bits from above.
i [7]  (repeated fun n)

Return the \( n \)-fold composition of the function \( \text{fun} \).

\[
\begin{align*}
\texttt{> } & \quad ((\text{repeated } \lambda x. (+ \ x \ 1) \ 5) \ 0) \\
\texttt{> } & \quad ((\text{repeated } \lambda x. (* \ 2 \ x) \ 3) \ 1)
\end{align*}
\]

j [7]  (inner-product \( \text{nums1} \ \text{nums2} \))

Assume that \( \text{nums1} \) is the list of numbers \( (a_1 \ a_2 \ \ldots \ a_n) \) and \( \text{nums2} \) is the list of numbers \( (b_1 \ b_2 \ \ldots \ b_n) \). (Note that both lists are assumed to have length \( n \).) Return the sum of the products of the corresponding elements of the two lists – i.e., the value \( (a_1*b_1) + (a_2*b_2) + \ldots + (a_n*b_n) \).

\[
\begin{align*}
\texttt{> } & \quad (\text{inner-product } '(1 \ 2 \ 3) \ '(4 \ 5 \ 6)) \\
\texttt{> } & \quad (\text{inner-product } '() \ '())
\end{align*}
\]

k [10]  (splits \( \text{lst} \))

Returns a list of all duples \( (a \ b) \) such that appending \( a \) and \( b \) gives \( \text{lst} \). The order of the duples does not matter.

\[
\begin{align*}
\texttt{> } & \quad \text{splits '}(1 \ 2 \ 3)\text{'} \\
\texttt{> } & \quad \text{splits '}(2 \ 3)\text{'}
\end{align*}
\]
1[10] \((\text{insert-all} \; \text{elt} \; \text{lst})\)

Assume that \textit{lst} is a list of elements not containing \textit{elt}. Returns a list of all the distinct ways that \textit{elt} can be inserted into \textit{lst}, maintaining the relative order of the elements of \textit{lst}. The order of elements in the result does not matter.

\[
\begin{array}{c}
> \ (\text{insert-all} \ 1 \ '2\ 3\ 4') \\
\ ((1\ 2\ 3\ 4) \ (2\ 1\ 3\ 4) \ (2\ 3\ 1\ 4) \ (2\ 3\ 4\ 1)) \\
> \ (\text{insert-all} \ 1 \ '3\ 4') \\
\ ((1\ 3\ 4) \ (3\ 1\ 4) \ (3\ 4\ 1)) \\
> \ (\text{insert-all} \ 1 \ '4') \\
\ ((1\ 4) \ (4\ 1)) \\
> \ (\text{insert-all} \ 1 \ '()) \\
\ ((1)) \\
\end{array}
\]

\textbf{Hint:} use \textit{splits} from above.

m [10] \((\text{permutations} \; \text{lst})\)

Assume that \textit{lst} is a list of distinct elements (i.e., no duplicates). Returns a list of all the permutations of the elements of \textit{lst}. The order of the permutations does not matter.

\[
\begin{array}{c}
> \ (\text{permutations} \ '()) \\
\ (()()) \\
> \ (\text{permutations} \ '(1)) \\
\ (()1()) \\
> \ (\text{permutations} \ '(1\ 2)) \\
\ (()1\ 2)) \ ; \ \text{Order doesnt matter} \\
> \ (\text{permutations} \ '(1\ 2\ 3)) \\
\ (()1\ 2\ 3) \ (1\ 3\ 2) \ (2\ 1\ 3) \ (2\ 3\ 1) \ (3\ 1\ 2) \ (3\ 2\ 1)) \ ; \ \text{Order doesnt matter}
\end{array}
\]

\textbf{Hint:} use \textit{insert-all} from above.
Problem 3 [20]: Lexical Ordering

Consider the following Scheme sorting function (which can be found in the CS251 download folder in ps2/lexord.scm):

\[
\text{(define insertion-sort}
\text{(lambda (less-than? elts)
    (if (null? elts)
        '()
        (insert less-than?
            (car elts)
            (insertion-sort less-than? (cdr elts))))))}
\]

\[
\text{(define insert}
\text{(lambda (less-than? elt lst)
    (if (null? lst)
        (list elt)
        (if (less-than? elt (car lst))
            (cons elt lst)
            (cons (car lst) (insert less-than? elt (cdr lst))))))})
\]

The \text{insertion-sort} procedure takes a binary \text{less-than?} predicate and a list of elements and returns a list of all the elements in sorted order from smallest to largest according to the \text{less-than?} predicate. For example:

\[
> \text{(insertion-sort < '(6 1 8 10 3 5))}
\]
\[
(1 3 5 6 8 10)
\]

\[
> \text{(insertion-sort > '(6 1 8 10 3 5))}
\]
\[
(10 8 6 5 3 1)
\]

\[
> \text{(insertion-sort}
\text{;; Sort elements by their distance from 5}
\text{(lambda (a b)
    (< (abs (- a 5))
        (abs (- b 5)))))}
\text{'} (6 1 8 10 3 5))
\]
\[
(5 6 3 8 1 10)
\]

For any binary predicate \text{P} of two elements of type \text{T}, it is possible to "lift" \text{P} to an ordering \text{P'} that compares two lists of elements of type \text{T}. The lifted ordering \text{P'} is called a lexical ordering predicate. Informally, \text{P'} compares two lists \text{L1} and \text{L2} elementwise from left to right using \text{P}. It returns true if it discovers an element of \text{L1} less than the corresponding element of \text{L2} or if \text{L1} is a proper prefix of \text{L2}; it returns false if it discovers an element of \text{L1} greater than the corresponding element of \text{L2} or if \text{L2} is a prefix of \text{L1}.

More formally, suppose list \text{L1} has the \text{m} elements \text{a1, a2, ..., am} and list \text{L2} has the \text{n} elements \text{b1, b2, ..., bn}. Then \text{(P' L1 L2)} is true if and only if one of the following two conditions holds:

1. There is a \text{k} in the range \text{[1..m]} such that \text{ai} and \text{bi} are equal for all \text{i} in the range \text{[1..k-1]} and \text{(P ak bk)} is true.

2. \text{m < n} and \text{ai} and \text{bi} are equal for all \text{i} in the range \text{[1..m]}

Lexical ordering is the standard way that alphabetic comparisons on characters is extended to compare two strings. It is often called dictionary ordering, since it determines how words are arranged in a dictionary.
In this problem, you are to write a Scheme function \texttt{lexord} that returns the lexical ordering predicate \( P' \) when given the predicate \( P \). To test two elements for equality, you should \textbf{not} use =, eqv?, eq?, or equal?. Rather, observe that \( ai \) and \( bi \) are equal if both \((P ai \ bi)\) and \((P bi \ ai)\) are false.

Here are some examples of \texttt{lexord} in action:

\begin{verbatim}
> (insertion-sort (lexord <) '((2 1) (1 3) (1) () (1 2 3) (2 2) (1 1) (3 2 1) (1 2) (2)))
(() (1) (1 1) (1 2) (1 2 3) (1 3) (2) (2 1) (2 2) (3 2 1))

> (insertion-sort (lexord >) '((2 1) (1 3) (1) () (1 2 3) (2 2) (1 1) (3 2 1) (1 2) (2)))
(() (3 2 1) (2) (2 2) (2 1) (1) (1 3) (1 2) (1 2 3) (1 1))

> (insertion-sort (lexord (lexord <)) '(((1 2) (2 1)) ((2 1) (2 2)) ((1 1) (2 2))
  ((1 2) (1 1)) ((2 1) (1 2))))
  '(((1 1) (2 2)) ((1 2) (1 1)) ((1 2) (2 1)) ((2 1) (1 2)) ((2 1) (2 2)))
\end{verbatim}

You can use the \texttt{test-lexord} procedure in \texttt{lexord.scm} to run the above three test cases.
**Problem 4 [40]: Functional Representation of Sets**

In CS230, you learned the extremely important notion of an abstract data type (ADT). In short, a data type can be defined by an interface of routines that manipulate elements of that type, independent of the details of how those routines are implemented.

ADTs are realizable in almost any programming language. For example, here is the Scheme interface to an ADT for a set of numbers:

\[
\begin{align*}
\text{(set-empty)} & \quad \text{Return an empty set.} \\
\text{(set-singleton } x \text{)} & \quad \text{Return a set whose single element is } x. \\
\text{(list->set } lst \text{)} & \quad \text{Return a set whose elements are the elements of the list } lst. \\
\text{(set-member? } x \text{ } s \text{)} & \quad \text{Return } \#t \text{ if } x \text{ is in set } s \text{ and } \#f \text{ otherwise.} \\
\text{(set-union } s1 \text{ } s2 \text{)} & \quad \text{Return a set whose elements are those that are in either } s1 \text{ or } s2. \\
\text{(set-intersection } s1 \text{ } s2 \text{)} & \quad \text{Return a set whose elements are those that are in both } s1 \text{ and } s2. \\
\text{(set-difference } s1 \text{ } s2 \text{)} & \quad \text{Return a set whose elements are those in } s1 \text{ that are not in } s2.
\end{align*}
\]

As in Java, we can implement this set ADT in Scheme in terms of familiar data structures like arrays, lists, or trees. However, unlike Java, Scheme also allows abstract data types to be implemented as functions. Intuitively, functions are just another kind of data structure. In fact, we shall see that functions are often more flexible data structures than conventional arrays, lists, and trees.

As a concrete example of this approach, we will explore how to implement sets as functions. In particular, we will represent a set as the membership predicate that determines whether a given element is in the set. For instance, the set \( \{2, 3, 5\} \) can be represented as the function

\[
\text{(lambda (x) (or (= x 2) (= x 3) (= x 5)))}
\]

This function returns \#t for the numbers 2, 3, and 5, but returns \#f for all other numbers. The empty set can be represented as the function that returns \#f for all numbers:

\[
\text{(lambda (x) #f)}
\]

This functional representation has numerous advantages over the array/list/tree versions. In particular, it is easy to specify sets that have infinite numbers of elements! For example, the set of all even numbers can be represented by the function

\[
\text{(lambda (x) (= (remainder x 2) 0))}
\]

This predicate is true of even integers, but is false for all other numbers. The set of all real numbers between 5 and 7 (inclusive) can be represented by the function:

\[
\text{(lambda (x) (and (>= x 5) (<= x 7))}
\]
The set of all integers can be represented by the standard Scheme predicate `integer?`, while the set of all numbers can be represented by

```scheme
(lambda (x) #t)
```

(This assumes that the predicates are only being applied to numbers. If we extended the notion of set to include any Scheme value, then the set of all numbers would be represented as the predicate `number?`)

**a.** Representing sets as membership predicates, implement the seven functions in the set ADT presented above. You should test out your implementation on suitable test cases, but only need to turn in your seven definitions.

**b.** Below are some other routines we could add to the interface to the set ADT. For each such routine, indicate whether or not it is possible to implement the routine (1) when sets are represented as lists (2) when sets are represented as membership predicates. Justify your answers.

1. `(set-empty? set)`
   Return #t if the `set` is empty, and false otherwise.

2. `(predicate->set pred)`
   Given a membership predicate `pred`, return a set of the elements for which `pred` is true.

3. `(set->list set)`
   Return a list of all the elements in `set`.

4. `(set-complement set)`
   Return the set of all numbers not in `set`.

5. `(subset? set1 set2)`
   Return #t if all of the elements of `set1` are also elements of `set2`, and #f otherwise.
Problem 5 [20] Church Numerals

The First-Class Functions handout discusses how \( n \)-fold composition functions (so-called Church numerals) can be viewed as the basis of a system for arithmetic. Write Scheme definitions for the functions \texttt{plus}, \texttt{times}, and \texttt{raise} that are described near the end of the First-Class Function handout.

Notes:

- The file \texttt{ps2/church.scm} in the CS251 download folder contains the code from the function composition section of the First-Class Functions handout, including \texttt{int->church} and \texttt{church->int}.

- For ideas on how to implement these three functions, carefully study the examples involving \texttt{twice} and \texttt{thrice} in the function composition section of the First-Class Functions handout.

- Each of your function definitions should be \textit{extremely} short. In fact, it’s possible to implement each definition as a “one-liner”. (But if you obey Scheme pretty-printing conventions, your definitions will be several lines long.)
Extra Credit Problems

These problems are optional. You should only attempt them after completing the rest of the problems. (Note that extra credit problems need not be turned in by the due date; they can be handed in any time during the semester. However, experience shows that students rarely turn them in after the problem set is due.)

EC1 [20]: Partitioning

Given an equality predicate \(eqpred\) (a so-called equivalence relation) and a list of elements, it is possible to partition the list into sublists (so-called equivalence classes) such that (1) every pair of elements from a given equivalence class are equal according to \(eqpred\); and (2) no two elements from distinct equivalence classes are equal according to \(eqpred\). Define a Scheme function \((\text{partition } \text{eqpred } \text{lst})\) that partitions \(\text{lst}\) into equivalence classes according to \(\text{eqpred}\). The order of elements within an equivalence class does not matter, nor does the order of equivalence classes within a partition. For example:

\[
\begin{align*}
> & (\text{partition} \ (\lambda (a \ b) \\
& \quad (\text{eqpred} \ (\text{remainder} \ a \ 3) \ (\text{remainder} \ b \ 3))) \\
& \quad '(17 \ 42 \ 6 \ 11 \ 16 \ 57 \ 51 \ 1 \ 23 \ 47)) \\
& \quad ((17 \ 11 \ 23 \ 47) \ (16 \ 1) \ (42 \ 6 \ 57 \ 51)) \\
\end{align*}
\]

\[
\begin{align*}
> & (\text{partition} \ (\lambda (a \ b) \\
& \quad (\text{eqpred} \ (\text{quotient} \ a \ 10) \ (\text{quotient} \ b \ 10))) \\
& \quad '(17 \ 42 \ 6 \ 11 \ 16 \ 57 \ 51 \ 1 \ 23 \ 47)) \\
& \quad ((17 \ 57 \ 47) \ (23) \ (11 \ 51 \ 1) \ (6 \ 16) \ (42) ) \\
\end{align*}
\]

\[
\begin{align*}
> & (\text{partition} \ (\lambda (\text{lst1} \ \text{lst2}) \\
& \quad (\text{eqpred} \ (\text{length} \ \text{lst1}) \ (\text{length} \ \text{lst2}))) \\
& \quad '((1 \ 2) \ () \ (2 \ 2) \ (1 \ 2 \ 3) \ (1 \ 3) \ (6) \ (2 \ 1) \ (1 \ 2 \ 1) \ (3 \ 1 \ 2) \ (5 \ 1))) \\
& \quad ((5 \ 1) \ (2 \ 1) \ (1 \ 3) \ (2 \ 2) \ (1 \ 2)) \ ((5)) \ ((3 \ 1 \ 2) \ (1 \ 2 \ 1) \ (1 \ 2 \ 3)) \ ((6)) ) \\
\end{align*}
\]

\[
\begin{align*}
> & (\text{partition} \ (\lambda (\text{lst1} \ \text{lst2}) \\
& \quad (\text{eqpred} \ (\text{sum} \ \text{lst1}) \ (\text{sum} \ \text{lst2}))) \\
& \quad '((1 \ 2) \ () \ (2 \ 2) \ (1 \ 2 \ 3) \ (1 \ 3) \ (6) \ (2 \ 1) \ (1 \ 2 \ 1) \ (3 \ 1 \ 2) \ (5 \ 1))) \\
& \quad ((2 \ 1) \ (1 \ 2)) \ ((5) \ ((1 \ 2 \ 1) \ (1 \ 3) \ (2 \ 2)) \ ((5 \ 1) \ (3 \ 1 \ 2) \ (6) \ (1 \ 2 \ 3)) ) \\
\end{align*}
\]

\[
\begin{align*}
> & (\text{partition} \ (\lambda (\text{lst1} \ \text{lst2}) \\
& \quad (\text{eqpred} \ ((\text{equal?} \ (\text{insertion-sort} < \ \text{lst1}) \\
& \quad \quad \quad (\text{insertion-sort} < \ \text{lst2})))) \\
& \quad '((1 \ 2) \ () \ (2 \ 2) \ (1 \ 2 \ 3) \ (1 \ 3) \ (6) \ (2 \ 1) \ (1 \ 2 \ 1) \ (3 \ 1 \ 2) \ (5 \ 1))) \\
& \quad ((2 \ 1) \ (1 \ 2)) \ ((5) \ ((3 \ 1 \ 2) \ (1 \ 2 \ 3)) \ ((1 \ 3)) \ ((6)) \ ((1 \ 2 \ 1)) \ ((5 \ 1)) ) \\
\end{align*}
\]

EC2 [20]: Improved Partitioning

Assume that \(eqpred\) from EC1 has unit cost. It can be shown that the best-case running time of the \(\text{partition}\) function from EC1 has a worst-case quadratic asymptotic running time. This running time can be improved to \(n \ (\log n)\) if the partitioning function is supplied with a less-than-or-equal-to predicate \(\text{leqpred}\) (also assumed to have unit cost), and the equality predicate for partitioning is derived from \(\text{leqpred}\). Define a Scheme partitioning function \((\text{partition-leq } \text{leqpred } \text{lst})\) that partitions \(\text{lst}\) according to \(\text{leqpred}\) and runs in \(n \ (\log n)\) worst-case asymptotic time.

\textbf{Note:} you need not solve EC1 in order to solve EC2.
EC3 [20]: Predecessor

The predecessor function \texttt{pred} on Church numerals has the following behavior:

- if \( c \) is a Church numeral representing a non-zero integer \( n \), then \((\texttt{pred } c)\) is a Church numeral representing the integer \( n - 1 \).
- if \( c \) is the Church numeral representing 0, then \((\texttt{pred } c)\) is the Church numeral 0.

For example:

\[
(\text{church->int } (\text{pred } (\text{int->church } 5)))
\]

\[
4
\]

\[
(\text{church->int } (\text{pred } (\text{int->church } 1)))
\]

\[
0
\]

\[
(\text{church->int } (\text{pred } (\text{int->church } 0)))
\]

\[
0
\]

Write the \texttt{pred} function in Scheme. \textit{Hint}: iterate over a pair of Church numerals.
Appendix A: Higher-Order List Operations

(define generate
  (lambda (seed next done?)
    (if (done? seed)
      ()
      (cons seed (generate (next seed) next done?)))))

(define map
  (lambda (f lst)
    (if (null? lst)
      ()
      (cons (f (car lst))
            (map f (cdr lst))))))

(define filter
  (lambda (pred lst)
    (if (null? lst)
      ()
      (if (pred (car lst))
        (cons (car lst) (filter pred (cdr lst)))
        (filter pred (cdr lst))))))

(define foldr
  (lambda (op init lst)
    (if (null? lst)
      init
      (op (car lst) (foldr op init (cdr lst))))))

(define foldl
  (lambda (op init lst)
    (if (null? lst)
      init
      (foldl op (op (car lst) init) (cdr lst))))))

(define forall?
  (lambda (pred lst)
    (if (null? lst)
      #t
      (and (pred (car lst))
           (forall pred (cdr lst))))))

(define exists?
  (lambda (pred lst)
    (if (null? lst)
      #f
      (or (pred (car lst))
          (exists? pred (cdr lst))))))

(define some
  (lambda (pred lst)
    (if (null? lst)
      #f
      (if (pred (car lst))
        (car lst)
        (some pred (cdr lst))))))
Name:

Date & Time Submitted (only if late):

Collaborators (anyone you collaborated with in the process of doing the problem set):

In the Time column, please estimate the time you spent on the parts of this problem set. Please try to be as accurate as possible; this information will help me to design future problem sets. I will fill out the Score column when grading your problem set.

<table>
<thead>
<tr>
<th>Part</th>
<th>Time</th>
<th>Score</th>
<th>Part</th>
<th>Time</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>General Reading</td>
<td></td>
<td></td>
<td>Problem 2j</td>
<td></td>
<td>[7]</td>
</tr>
<tr>
<td>Problem 1</td>
<td>[20]</td>
<td></td>
<td>Problem 2k</td>
<td>[10]</td>
<td></td>
</tr>
<tr>
<td>Problem 2a</td>
<td>[5]</td>
<td></td>
<td>Problem 2l</td>
<td>[10]</td>
<td></td>
</tr>
<tr>
<td>Problem 2b</td>
<td>[5]</td>
<td></td>
<td>Problem 2m</td>
<td>[10]</td>
<td></td>
</tr>
<tr>
<td>Problem 2c</td>
<td>[5]</td>
<td></td>
<td>Problem 3</td>
<td>[20]</td>
<td></td>
</tr>
<tr>
<td>Problem 2d</td>
<td>[6]</td>
<td></td>
<td>Problem 4</td>
<td>[40]</td>
<td></td>
</tr>
<tr>
<td>Problem 2f</td>
<td>[8]</td>
<td></td>
<td>Problem EC1</td>
<td>[20]</td>
<td></td>
</tr>
<tr>
<td>Problem 2g</td>
<td>[10]</td>
<td></td>
<td>Problem EC2</td>
<td>[20]</td>
<td></td>
</tr>
<tr>
<td>Problem 2h</td>
<td>[10]</td>
<td></td>
<td>Problem EC3</td>
<td>[20]</td>
<td></td>
</tr>
<tr>
<td>Problem 2i</td>
<td>[7]</td>
<td></td>
<td>Subtotal</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Subtotal</td>
<td></td>
<td></td>
<td>Total</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>