PROBLEM SET 2
Due Friday, February 16, 2001

Reading: First-Class Functions Handout (#11); SICP 1.3, 2.2.2--2.2.4

Overview: The purpose of this assignment is to give you experience with first-class functions. These can twist your brain a bit, so leave sufficient time to do the problems.

Reading: First-Class Functions (Handout #1); SICP 1.3; 2.2.2 -- 2.2.4. If you haven't done so already, study the notes on debugging in the MIT-Scheme handout (#7), since this will save you valuable time throughout the rest of the semester.

Submission: Problem 1 is a pencil-and-paper problem that only needs to appear in your hardcopy submission. Problems 2, 3, and 4 involve writing Scheme functions in the files adp.scm, set.scm, and church.scm within the ~/cs251/ps2 directory. For these problems, your hardcopy submission should be your final versions of these files. These files should also be your softcopy submission, which you should copy to the directory ~/cs251/drop/ps1/username, where username is your username.

Please attach to your hardcopy a problem set header sheet (which can be found at the end of this assignment) indicating the time that you (and your partner, if you are working with one) spent on the parts of the assignment. If you work with a partner, you need only submit a single hardcopy and softcopy; please indicate on top of the problem set header sheet where the softcopy can be found.

Problem 1 [15]: 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)

Carefully show all details of the substitution model. If evaluating an expression gives rise to an error, describe the nature of the error. You may use the Scheme interpreter and substitution model interpreter to check your answers, but please do not use these until you have already tried to figure out the answers on your own.
Problem 2 [50]: Aggregate Data Paradigm

Implement the following functions in terms of the higher-order list operations in Appendix A. (These can be found in your local cvs-controlled cs251 directory in util/list-ops.scm) You should not use recursion in any of your definitions, though you may want to define some auxiliary (non-recursive) functions. You will recognize most of these functions from PS1.

Flesh out all of the definitions in the file cs251/ps2/adp.scm. You can test the function from part P by evaluating (test-P), and can test all parts by evaluating (test-adp). (The testing functions are defined in a file adp-test.scm, which is automatically loaded when you load adp.scm.)

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 [5]** \(\text{(sum-multiples-of-3-or-5 \ m \ n)}\)

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.

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

**e [5]** \(\text{(all-contain-multiple? \ n \ intss)}\)

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 [5]** \(\text{(cartesian-product \ lst1 \ lst2)}\)

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 \(\text{lst2}\)).

\[
\begin{align*}
> & (\text{cartesian-product '((1 2) '((a b c))} \\
& ((1 a) (1 b) (1 c) (2 a) (2 b) (2 c)) \\
> & (\text{cartesian-product '2 1) '((c a b))} \\
& ((2 a) (2 b) (1 c) (1 a) (1 b)) \\
> & (\text{cartesian-product '((c a b) '2 1))} \\
& ((c 2) (c 1) (a 2) (a 1) (b 2) (b 1)) \\
> & (\text{cartesian-product '1 1))} \\
& ((1 a)) \\
> & (\text{cartesian-product '() '((a b c))} \\
& ()
\end{align*}
\]
g [5]  (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 \texttt{fast-exp} can be defined recursively as follows:

\[
\text{fast-exp} \quad (\text{base} \quad \text{power}) \\
\begin{align*}
&= \text{if} \quad (\text{even?} \quad \text{power}) \\
&\quad \{ \text{square} \quad (\text{fast-exp} \quad (\text{base} \quad \text{quotient power} \ 2)) \} \\
&\quad \{ \text{base} \quad (\text{fast-exp} \quad \text{base} \quad (\text{quotient power} \ 2)) \} \\
\end{align*}
\]

Unlike the naïve approach to exponentiation, which requires \texttt{power} multiplications, \texttt{fast-exp} requires no more than \(2\times\log_{\text{base}}(\text{power})\) multiplications. For instance, whereas \(3^{1024}\) takes 1024 multiplications by the naïve method, it takes only 10 multiplications via \texttt{fast-exp}! Give a non-recursive definition of \texttt{fast-exp} using the higher-order list operators. Your definition should perform the same number of multiplications as \texttt{fast-exp}. \textit{Hint:} use \texttt{bits} from throws.
i [5]  (repeated fun n)

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

\[
\begin{align*}
> & \ ((\text{repeated} \ (\lambda (x) (+ x 1)) \ 5) \ 0) \\
& \ 5 \\
> & \ ((\text{repeated} \ (\lambda (x) (* 2 x)) \ 3) \ 1) \\
& \ 8
\end{align*}
\]

j [5]  (inner-product nums1 nums2)

Assume that \( \text{nums1} \) is the list of numbers \((a1 \ a2 \ldots \ an)\) and \( \text{nums2} \) is the list of numbers \((b1 \ b2 \ldots \ bn)\). (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 \((a1*b1) + (a2*b2) + \ldots + (an*bn)\).

\[
\begin{align*}
> & \ (\text{inner-product} \ '(1 \ 2 \ 3) \ '(4 \ 5 \ 6)) \\
& \ 32 \ ; \ 4 + 10 + 18 \\
> & \ (\text{inner-product} \ '() \ '()) \\
& \ 0
\end{align*}
\]
Problem 3 [20]: Functional Representation of Sets of Numbers

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 integers:

\[
\begin{align*}
\text{(set-empty)} & \quad \text{Return an empty set.} \\
\text{(set-singleton \(x\))} & \quad \text{Return a set whose single element is \(x\).} \\
\text{(list->set \(lst\))} & \quad \text{Return a set whose elements are the elements of the list \(lst\).} \\
\text{(set-member? \(x\) \(s\))} & \quad \text{Return #t if \(x\) is in set \(s\) and #f otherwise.} \\
\text{(set-union \(s1\) \(s2\))} & \quad \text{Return a set whose elements are those that are in either \(s1\) or \(s2\).} \\
\text{(set-intersection \(s1\) \(s2\))} & \quad \text{Return a set whose elements are those that are in both \(s1\) and \(s2\).} \\
\text{(set-difference \(s1\) \(s2\))} & \quad \text{Return a set whose elements are those in \(s1\) 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 integers 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 \((x)\)}
\begin{align*}
\text{(or (= \(x\) 2) (= \(x\) 3) (= \(x\) 5))}
\end{align*}
\]

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 integers can be represented by the function

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

This predicate is true of even integers, but is false for all other integers. The set of integers between 251 and 6001 (inclusive) can be represented by the function:
The set of all numbers can be represented by

\[(\lambda (x) (\text{and} (\geq x 251) (\leq x 6001)))\]

This assumes that the predicates are only being applied to integers. If we extended the notion of set to include other Scheme values, then the set of all integers would be represented as the predicate `integer?`.

**a.** Representing sets as membership predicates, implement the seven functions in the set ADT presented above. You should do this by fleshing out the skeleton definitions in the file `~/cs251/ps2/set.scm`. You can test each of these functions \(F\) by evaluating \((\text{test-}F)\), and can test all your set functions by evaluating \((\text{test-set})\). Your implementation by loading the file `cs251/ps2/set-test.scm` and evaluating the invocation \((\text{test-set})\).

(The testing functions are defined in a file `set-test.scm`, which is automatically loaded when you load `set.scm`.)

**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. \((\text{set-empty?} \ \text{set})\)
   Return \#t if the \text{set} is empty, and false otherwise.

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

3. \((\text{set->list} \ \text{set})\)
   Return a list of all the elements in \text{set}.

4. \((\text{set-complement} \ \text{set})\)
   Return the set of all numbers not in \text{set}.

5. \((\text{subset?} \ \text{set1} \ \text{set2})\)
   Return \#t if all of the elements of \text{set1} are also elements of \text{set2}, and \#f otherwise.
Problem 4 [15] Church Numerals

The First-Class Functions handout (#11) 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 plus, times, and raise that are described near the end of the First-Class Function handout. Flesh out these definitions in the file cs251/ps2/church.scm, which also contains code from the function composition section of the First-Class functions handout. You may test your definitions by loading cs251/ps2/church-test.scm and evaluating (test-church).

Notes:

- Your definitions should not use any of the following: int->church, church->int, repeated, n-fold, or recursion.

- Your definitions may use any other functions. In particular, the following functions are useful for some: succ, compose, identity, zero, and one, where zero and one are defined as:

```scheme
(define zero (lambda (f) (lambda (x) x))) ; same as (n-fold 0)
(define one (lambda (f) (lambda (x) (f x)))) ; same as (n-fold 1)
```

Note that the above functions are not required for your solutions. Indeed, there are solutions for all three definitions that use none of the above functions.

- For ideas on how to implement these three functions, carefully study the examples involving twice and thrice in the function composition section of the First-Class Functions handout. You can implement all three functions by generalizing patterns you see in invocations of twice and thrice.

- One way to think of addition is as repeated incrementing. For example, we can get the Church numeral for five by using three applications of the successor function starting with two: (succ (succ (succ two))). Can you express this using three, succ, and two? If so, you know how to do addition on Church numerals! Then note that multiplication is repeated addition and exponentiation is repeated multiplication.

- Each of your function definitions should be 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 \( \text{eqpred} \) (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 \( \text{eqpred} \); and (2) no two elements from distinct equivalence classes are equal according to \( \text{eqpred} \). Define a Scheme function \((\text{partition eqpred lst})\) 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:

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

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

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

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

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

EC2 [20]: Improved Partitioning

Assume that \( \text{eqpred} \) 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 \((\text{partition-leq leqpred lst})\) that partitions \( \text{lst} \) according to \( \text{leqpred} \) and runs in \( n \ (\log n) \) worst-case asymptotic time.

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

The predecessor function \( \text{pred} \) on Church numerals has the following behavior:

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

For example:

\[
\begin{align*}
(\text{church} \rightarrow \text{int} \ (\text{pred} \ (\text{int} \rightarrow \text{church} \ 5))) & \quad 4 \\
(\text{church} \rightarrow \text{int} \ (\text{pred} \ (\text{int} \rightarrow \text{church} \ 1))) & \quad 0 \\
(\text{church} \rightarrow \text{int} \ (\text{pred} \ (\text{int} \rightarrow \text{church} \ 0))) & \quad 0
\end{align*}
\]

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

(define identity
  (lambda (x) x))

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

(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 map2
  (lambda (f lst1 lst2)
    (map (lambda (duple)
          (f (first duple) (second duple)))
          (zip lst1 lst2))))

(define zip
  (lambda (lst1 lst2)
    (if (or (null? lst1) (null? lst2))
      ()
      (cons (list (car lst1) (car lst2))
        (zip (cdr lst1) (cdr lst2))))))

(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)
        none
        (if (pred (car lst))
            (car lst)
            (some pred (cdr lst))))))

(define none '(*none*))
(define none? (lambda (x) (eq? x none)))

(define some
  (lambda (pred lst)
    (if (null? lst)
        none
        (if (pred (car lst))
            (car lst)
            (some pred (cdr lst))))))

(define none '(*none*))
(define none? (lambda (x) (eq? x none)))
CS251 Problem Set 2  
Due Friday, February 16, 2001

Names of Team Members:

Date & Time Submitted:

Soft Copy Directory:

Collaborators (any teams collaborated with in the process of doing the problem set):

In the Time column, please estimate the total time each team member 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 2h</td>
<td>[5]</td>
<td></td>
</tr>
<tr>
<td>Problem 2c</td>
<td>[5]</td>
<td></td>
<td>Problem 4</td>
<td>[20]</td>
<td></td>
</tr>
<tr>
<td>Problem 2g</td>
<td>[5]</td>
<td></td>
<td><strong>Subtotal</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Subtotal</strong></td>
<td></td>
<td></td>
<td><strong>Total</strong></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>