CS251 Programming Languages **Handout #12**
Prof. Lyn Turbak **February 9, 2001**
Wellesley College

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 problem, your hardcopy submission should be your final versions of these files. These file 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 *i*th element is a list of the form $(a_i\ b_i)$. Return a list of the form `(lst1 lst2)` where `lst1` and `lst2` are length *len* lists whose *i*th elements are a_i and b_i, respectively.

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

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

> (unzip '())
(() ())
```

d [5] `(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.

```
> (sum-multiples-of-3-or-5 0 10)
33 ; 3 + 5 + 6 + 9 + 10

> (sum-multiples-of-3-or-5 –9 12)
22

> (sum-multiples-of-3-or-5 18 18)
18

> (sum-multiples-of-3-or-5 10 0)
0 ; The range “10 up to 0” is empty.
```

e [5] `(all-contain-multiple? n intss)`

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

```
> (all-contain-multiple? 5 '((17 10 12) (25) (3 7 5)))
#t

> (all-contain-multiple? 3 '((17 10 12) (25) (3 7 5)))
#f

> (all-contain-multiple? 3 '())
#t
```

f [5] `(cartesian-product lst1 lst2)`

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

```
> (cartesian-product '(1 2) '(a b c))
((1 a) (1 b) (1 c) (2 a) (2 b) (2 c))

> (cartesian-product '(2 1) '(c a b))
((2 c) (2 a) (2 b) (1 c) (1 a) (1 b))

> (cartesian-product '(c a b) ‘(2 1))
((c 2) (c 1) (a 2) (a 1) (b 2) (b 1))

> (cartesian-product '(1) '(a))
((1 a))

> (cartesian-product '() '(a b c))
()
```

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)
```

h [5] `(fast-expt base power)`

The fast exponentiation procedure `fast-expt` can be defined recursively as follows:

```
;; 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))))))))

(define even? (lambda (n) (= 0 (remainder n 2))))
(define square (lambda (x) (* x x)))
```

Unlike the naïve approach to exponentiation, which requires `power` multiplications, `fast-expt` requires no more than $2*\log_2(\texttt{power})$ multiplications. For instance, whereas 3^{1024} takes 1024 multiplications by the naïve method, it takes only 10 multiplications via `fast-expt`! Give a non-recursive definition of `fast-expt` using the higher-order list operators. Your definition should perform the same number of multiplications as `fast-expt`. *Hint:* use `bits` from above.

```
> (fast-expt 2 123)
10633823966279326983230456482242756608

> (fast-expt 2 1234)
29581122460809862906004469571610359078633968713537299223955620705065735079623892426105383
72483780501864436477590709559931208208993303817609370272124828409449413621106654437751834
95726811929203861182015218323892077355983393191208928867652655993602487903113708549402668
62452110061179427034023276609931709804888749380902312739825386061877261903500988327294112
9544640111837184

> (fast-expt 2 12345)
16417101068825821635602074166390650141012723553073588127211610308792509417139014428015903
45364394577348704191271404016671955103310856571853327210892364011930444934571162997688443
44303479235489462436380672117015123283299131391904179287678259173308536738761981139958654
88085223490844833881728901416677416986925133937982859748492918775437864739032217778051333
38829900741162462812693649337248923421345047024910400166375574298108937807651974185894775
84716543480995722533317862352141459217781316266211186486157019262080414077670264642736018
42699811352344573268085614432987697227330070339258499772920719797108394570034549409240014
71869973070120694540684895890356769794481698480608369249458241977064933061082585119360303
41393221586423523264452449403781993352421885094664052270795527632721896121424813173522474
67439588615509220340403673074847478171071574544613546809813983182408325964791917527350368
15611726846242833844385047765030004322416045504543741163208222271919113221234840850639263
50606342197146407841178028071147192533942517270553513988142925976090769695456221159699052
58353301133165207934709309817308697548353927446402335745648446548292747956943732036859222
27602781703060767334388010983707976757112746710549707114421589305616843431357741187415945
06702833147396758825015850042983343690345185995956235143825771620543546030664562647854656
43130264457411987382021559571861862448523242200657555000706888373424145468636885673449626
53859088094039724946851377411228668967196780539372858184097516703201405018430392240407358
70096889596273419106389103662095318937990625980136711988237421962315266686856089505981438
44085063806758932114175949901702383959685845554819200014008514229416698706349902479268133
48431597909363213519198597586695692005415076120997809097051989021760262198722017154220960
90343686272984351441594569506778041062663266799342793856313801540959815845788584759033248
82824856158645027117277724097179565608200184811581526093052166316748017388606401911857277
82815167351577795558881677870644325585954108439874464978816662884232331700604130259246299
50477303342180149398926073618582715358742250388958231281694757980523791263699450732952325
72766420994778606398256177532763850451691857010131939169841238860760374248441474826838966
91291180268789697357822868411168426564105746476075244189007203280453779933862798087689903
76289424757351052369393977137871998119168898493037938756635621557623138404459266598837784
22932579983878202606048149686556175703183900225709180287694924839274417566911224208843988
32483363105970012573859807769152935119887774719353105495688180833217794675140403822871857
67911769630971553915410012677600002457982207465176670752102117002773980548089696530972476
43969459988128181297321726585388472790653547974585408533885110514458548199415620649743674
58999448777325314125412790143003245948906239411455098569409828637698344300481205629667979
07114102689879364945689860493474954538422367719507882513166051007352994068319251450666676
64836820056432938299875887576041425965400497726130998826731980635485605178455399093661063
47333759841590287223786149844502553863155856319945033500021429104931902548256107074005899
76364985748467955131077971641882672895854571236368282811336220769174784720113331269084746
52420412426347505411284163093358616195036115696469686075600480420563557567616835632252622
23271728110021463927544450511821698052846302597035426339551261795201130596299142298336885
35925729676778028406897316106101038469119090984567152591962365415039646394591503830797626
33924698605707775861141366491416874537526678629814117149657394161438774412584368567706361
97829187598231060210540377578577615874722408350405804473605440290649304125699431697292381
02162312218687930203068055400275795180972382856696655279408212344832
```

i [5] `(repeated` *`fun n`*`)`

Return the *n* -fold composition of the function *fun*.

```
> ((repeated (lambda (x) (+ x 1)) 5) 0)
5

> ((repeated (lambda (x) (* 2 x)) 3) 1)
8
```

j [5] `(inner-product` *`nums1 nums2`*`)`

Assume that *nums1* is the list of numbers (*a1 a2 … an*) and *nums2* is the list of numbers (*b1 b2 … 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*) + … + (*an*bn*).

```
> (inner-product '(1 2 3) '(4 5 6))
32 ; 4 + 10 + 18

> (inner-product '() '())
0
```

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:

`(set-empty)`
Return an empty set.

`(set-singleton` *`x`*`)`
Return a set whose single element is *`x`*.

`(list->set` *`lst`*`)`
Return a set whose elements are the elements of the list *`lst`*.

`(set-member?` *`x s`*`)`
Return #t if x is in set s and #f otherwise.

`(set-union` *`s1 s2`*`)`
Return a set whose elements are those that are in either *`s1`* or *`s2`*.

`(set-intersection` *`s1 s2`*`)`
Return a set whose elements are those that are in both *`s1`* and *`s2`*.

`(set-difference` *`s1 s2`*`)`
Return a set whose elements are those in *`s1`* that are not in *`s2`*.

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

```
(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:

```
(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

```
(lambda (x) (= (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:

```
(lambda (x) (and (>= x 251) (<= x 6001))
```

The set of all numbers can be represented by

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

(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 `(test-`*`F`*`)`, and can test all your set functions by evaluating `(test-set)`. your implementation by loading the file `cs251/ps2/set-test.scm` and evaluating the invocation `(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.

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

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

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

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

v. `(subset?` *`set1 set2`*`)`
Return `#t` if all of the elements of *`set1`* are also elements of *`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:

```
(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 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 `(partition` *`eqpred lst`*`)` that partitions `lst` into equivalence classes according to *`eqpred`*. The order of elements within an equivalence class does not matter, nor does the order of equivalence classes within a partition. For example:

```
> (partition (lambda (a b)
               (= (remainder a 3) (remainder b 3)))
   '(17 42 6 11 16 57 51 1 23 47))
( (17 11 23 47) (16 1) (42 6 57 51) )

> (partition (lambda (a b)
               (= (quotient a 10) (quotient b 10)))
   '(17 42 6 11 16 57 51 1 23 47))
( (17 57 47) (23) (11 51 1) (6 16) (42) )

> (partition (lambda (lst1 lst2)
               (= (length lst1) (length lst2)))
    '((1 2) () (2 2) (1 2 3) (1 3) (6) (2 1) (1 2 1) (3 1 2) (5 1)))
( ((5 1) (2 1) (1 3) (2 2) (1 2)) (()) ((3 1 2) (1 2 1) (1 2 3)) ((6)) )

> (partition (lambda (lst1 lst2)
               (= (sum lst1) (sum lst2)))
    '((1 2) () (2 2) (1 2 3) (1 3) (6) (2 1) (1 2 1) (3 1 2) (5 1)))
( ((2 1) (1 2)) (()) ((1 2 1) (1 3) (2 2)) ((5 1) (3 1 2) (6) (1 2 3)) )

> (partition (lambda (lst1 lst2)
               (equal? (insertion-sort < lst1)
                       (insertion-sort < lst2)))
    '((1 2) () (2 2) (1 2 3) (1 3) (6) (2 1) (1 2 1) (3 1 2) (5 1)))
( ((2 1) (1 2)) (()) ((3 1 2) (1 2 3)) ((1 3)) ((6)) ((1 2 1)) ((5 1)) )
```

EC2 [20]: Improved Partitioning

Assume that *`eqpred`* from EC1 has unit cost. It can be shown that the best-case running time of the `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 *`leqpred`* (also assumed to have unit cost), and the equality predicate for partitioning is derived from *`leqpred`*. Define a Scheme partitioning function `(partition-leq` *`leqpred lst`*`)` that paritions *`lst`* according to *`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 `pred` on Church numerals has the following behavior:

- if *c* is a Church numeral representing a non-zero integer *n*, then `(pred c)` is a Church numeral representing the integer *n - 1*.
- if c is the Church numeral representing 0, then `(pred c)` is the Church numeral 0.

For example:

```
(church->int (pred (int->church 5))
4

(church->int (pred (int->church 1))
0

(church->int (pred (int->church 0))
0
```

Write the `pred` function in Scheme. *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)))
```

Problem Set Header Page
Please make this the first page of your hardcopy submission.

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.*

Part	Time	Score	Part	Time	Score
General Reading			Problem 2h [5]		
Problem 1 [15]			Problem 2i [5]		
Problem 2a [5]			Problem 2j [5]		
Problem 2b [5]			Problem 3 [10]		
Problem 2c [5]			Problem 4 [20]		
Problem 2d [5]			Problem EC1 [20]		
Problem 2e [5]			Problem EC2 [20]		
Problem 2f [5]			Problem EC3 [20]		
Problem 2g [5]			**Subtotal**		
Subtotal			**Total**		