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 In particular, `(op init (car lst))` in the original PS2 has now been replaced by `(op (car lst) init);`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 *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 [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.

```
> (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 [7] `(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 [8] `(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 [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)
```

h [10] `(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)))
```

Give a non-recursive definition of `fast-expt` using the higher-order list operators. *Hint:* use `bits` from above.

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

> (fast-expt 2 1234)
2958112246080986290600446957161035907863396871353729922395562070506573507962389242610538
7248378050186443647759070955993120820899330381760937027212482840944941362110665443775183
4957268119292038611820152183238920773559833931912089288676526559936024879031137085494026
6862452110061179427034023276609931709804888749380902312739825386061877261903500988327294
1129544640111837184

> (fast-expt 2 12345)
1641710106882582163560207416639065014101272355307358812721161030879250941713901442801590
3453643945773487041912714040166719551033108565718533272108923640119304449345711629976884
4344303479235489462436380672117015123283299131391904179287678259173308536738761981139958
6548808522349084483388172890141667741698692513393798285997484929187754378647390322177780
5133388299007411624628126936493372489234213450470249104001663755742981089378076519741858
9477584716543480995722533317862352141459217781316266211186486157019262080414077670264642
7360184269981135234457326808561443298769722733007033925849977292071979710839457003454940
9240014718699730701206945406848958903567697944816984806083692494582419770649330610825851
1936030341393221586423523264452449403781993352421885094664052270795527632721896121424813
1735224746743958861550922034040367307484747817107157454461354680981398318240832596479191
7527350368156117268462428338443850477650300043224160455045437411632082222719191132212348
4085063926350606342197146407841178028071147192533942517270553513988142925976090769695456
2211596990525835330113316520793470930981730869754835392744640233574564844654829274795694
3732036859222276027817030607673343880109837079767571127467105497071144215893056168434313
5774118741594506702833147396758825015850042983343690345185995956235143825771620543546030
6645626478546564313026445741198738202155957186186244852324220065755500070688837342414546
8636885673449626538590880940397249468513774112286689671967805393728581840975167032014050
1843039224040735870096889596273419106389103662095318937990625980136711988237421962315266
6868560895059814384408506380675893211417594990170238395968584555481920001400851422941669
8706349902479268133484315979093632135191985975866956920054150761209978090970519890217602
6219872201715422096090343686272984351441594569506778041062663266799342793856313801540959
8158457885847590332488282485615864502711727772409717956560820018481158152609305216631674
8017388606401911857277828151673515777955588816778706443255859541084398744649788166628842
3233170060413025924629950477303342180149398926073618582715358742250388958231281694757980
5237912636994507329523257276642099477860639825617753276385045169185701013193916984123886
0760374248441474826838966912911802687896973578228684111684265641057464760752441890072032
8045377993386279808768990376289424757351052369393977137871998119168898493037938756635621
5576231384044592665988377842293257998387820260604814968655617570318390022570918028769492
4839274417566911224208843988324833631059700125738598077696152935119887774719353105495688
1808332177946751404038228718567911769630971553915410012677600002457982207465176670752102
1170027739805480896965309724764396945998812818129732172658538847279065354797458540853388
5110514458548199415620649743674589994487773253141254127901430032459489062394114550985694
0982863769834430048120562966797907114102689879364945689860493474954538422367719507882513
1660510073529940683192514506666766483682005643293829987588757604142596540049772613099882
6731980635485605178455399093661063473337598415902872237861498445025538631558563199450335
0002142910493190254825610707400589976364985748467955131077971641882672895854571236368282
8113362207691747847201133312690847465242041242634750541128416309335861661950361156964696
8607560048042056355756761683563325262232717281100214639275444505118216980528463025970354
2633955126179520113059629914229833688535925729676778028406897316106101038469119090984567
1525919623654150396463945915038307976263392469860570777586114136649141687453752667862981
4117149657394161438774412584368567706361978291875982310602105403775785776158747224083504
0580447360544029064930412569943169729238102162312218687930203068055400275795180972382856
6966552794082123448320
```

i [7] `(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 [7] `(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
```

k [10] `(splits lst)`

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

```
> (splits '(1 2 3))
((() (1 2 3)) ((1) (2 3)) ((1 2) (3)) ((1 2 3) ()))

> (splits '(2 3))
((() (2 3)) ((2) (3)) ((2 3) ()))

> (splits '(2))
((() (2)) ((2) ()))

> (splits '())
((() ()))
```

l [10] `(insert-all elt lst)`

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

```
> (insert-all 1 '(2 3 4))
((1 2 3 4) (2 1 3 4) (2 3 1 4) (2 3 4 1))

> (insert-all 1 '(3 4))
((1 3 4) (3 1 4) (3 4 1))

> (insert-all 1 '(4))
((1 4) (4 1))

> (insert-all 1 '())
((1))
```

Hint: use `splits` from above.

m [10] `(permutations lst)`

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

```
> (permutations ‘())
(())

> (permutations ‘(1))
((1))

> (permutations ‘(1 2))
((1 2) (2 1)) ; Order doesn’t matter

> (permutations ‘(1 2 3))
((1 2 3) (1 3 2) (2 1 3) (2 3 1) (3 1 2) (3 2 1)) ; Order doesn’t matter
```

Hint: use `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`):

```
(define insertion-sort
  (lambda (less-than? elts)
    (if (null? elts)
        '()
       (insert less-than?
               (car elts)
               (insertion-sort less-than? (cdr elts))))))

(define insert
  (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 `insertion-sort` procedure takes a binary `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 less-than? predicate. For example:

```
> (insertion-sort < '(6 1 8 10 3 5))
(1 3 5 6 8 10)

> (insertion-sort > '(6 1 8 10 3 5))
(10 8 6 5 3 1)

> (insertion-sort
    ;; Sort elements by their distance from 5
    (lambda (a b)
      (< (abs (- a 5))
         (abs (- b 5))))
      '(6 1 8 10 3 5))
(5 6 3 8 1 10)
```

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

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

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

2. *m* < *n* and *ai* and *bi* are equal for all *i* in the range [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 `lexord` that returns the lexical ordering predicate *P′* when given the predicate *P*. To test two elements for equality, you should **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 `lexord` in action:

```
> (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)))
```

You can use the `test-lexord` procedure in `lexord.scm` to run the above three test cases.

Problem 4 [40]: Functional Representation of Sets

In CS230, you learned the 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:

`(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 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 numbers 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 numbers. The set of all real numbers between 5 and 7 (inclusive) can be represented by the function:

```
(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

```
(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.

> **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 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 `plus`, `times`, and `raise` that are described near the end of the First-Class Function handout.

Notes:

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

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

CS251 Problem Set 2

Due Friday, February 25, 2000

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.*

Part	Time	Score	Part	Time	Score
General Reading			Problem 2j [7]		
Problem 1 [20]			Problem 2k [10]		
Problem 2a [5]			Problem 2l [10]		
Problem 2b [5]			Problem 2m [10]		
Problem 2c [5]			Problem 3 [20]		
Problem 2d [6]			Problem 4 [40]		
Problem 2e [7]			Problem 5 [20]		
Problem 2f [8]			Problem EC1 [20]		
Problem 2g [10]			Problem EC2 [20]		
Problem 2h [10]			Problem EC3 [20]		
Problem 2i [7]			**Subtotal**		
Subtotal			**Total**		