MASSACHUSETTS INSTITUTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.001—Structure and Interpretation of Computer Programs
Spring Semester, 1997

Problem Set 5 Solutions

Tutorial Exercise 1: The box an pointer diagram for

```
'(entry (name davis richard) (age 65) (posn (associate professor)))
```

is shown below

Tutorial Exercise 2: SICP 2.26

```
(define x (list 1 2 3))
(define y (list 4 5 6))
(append x y) ==> (1 2 3 4 5 6)
(cons x y) ==> ((1 2 3) 4 5 6)
(list x y) ==> ((1 2 3) (4 5 6))
```

Tutorial Exercise 3: SICP 2.28

The thing to keep in mind here is that `fringe` always returns a list. To compute the fringe of a pair, we recursively apply fringe to the `car` and the `cdr` and append the two resulting lists. If the tree is an atom (not a pair), we return a *list* containing the atom.

```
(define (fringe tree)
  (cond ((null? tree) '())
        ((pair? tree) (append (fringe (car tree)) (fringe (cdr tree))))
        (else (list tree))))
(fringe '((1 2)(3 4))) ==> (1 2 3 4)
```

Tutorial Exercise 4: SICP 2.37

We are given the following procedures, shown in full here.

```
(define (accumulate op init l)
  (if (null? l) init
      (op (car l) (accumulate op init (cdr l)))))

(define (accumulate-n op init seqs)
  (if (null? (car seqs)) '()
      (cons (accumulate op init (map car seqs))
            (accumulate-n op init (map cdr seqs)))))

(define (dot-product v w) (accumulate + 0 (map * v w)))
```

Using these we fill in the blanks to create a matrix-vector arithmetic package as shown below. Multipling a matrix by a vector is just computing the dot-product of the vector with each row of the matrix. We transpose a matrix by cons-ing up the corresponding elements of each row (a column) to form each row of the result. Using these two procedures, we can then multiple two matrices by multiplying each row of the first by the transpose of the second to yield each row of the result.

```
(define (matrix-*-vector m v) (map (lambda (x) (dot-product x v)) m))

(define (transpose mat) (accumulate-n cons '() mat))

(define (matrix-*-matrix m n)
  (let ((cols (transpose n)))
    (map (lambda (row) (matrix-*-vector cols row)) m)))

(define v '(2 4 6))
(define w '(3 5 7))
(dot-product v w) ==> 68
(define m '((1 2 3)(4 5 6)(7 8 9)))
(matrix-*-vector m v) ==> (28 64 100)
(matrix-*-matrix '((1 2 3)(4 5 6)) '((1 2)(2 3)(3 4))) ==> ((14 20) (32 47))
```

Tutorial Exercise 5: SICP 2.54

Two objects are `equal?` if they are both pairs and their `cars` and `cdrs` are both `equal?` or if they are both atoms and are `eq?`.

```
(define (equal? l1 l2)
  (cond ((and (pair? l1) (pair? l2))
         (and (equal? (car l1) (car l2))
              (equal? (cdr l1) (cdr l2))))
        ((eq? l1 l2) #t)
        (else #f)))
```

Lab Exercise 1: The following procedure computes the average age in the database by separately accumulating both the number of entries in the database and the total of all of the ages in the database. I've renamed the accumulation procedure `accumulate-db` to distinguish it from the procedure used above. While it is possible to accumulate both the count and total age in a single pass over the database, it is much simpler to compute them separately.

```
(define (average-age-db db)
  (let ((count (accumulate-db (lambda (x y) (+ 1 y)) 0 db))
        (total-age (accumulate-db (lambda (x y) (+ (age (age-struct x)) y)) 0
db)))
    (/ total-age count)))
```

For example

```
(average-age-db sample-linear-database) ==>
;Value: 61.49
```

We can also use the accumulate procedure to generate a list of all the names in the database

```
(define (all-names-in-db db)
  (accumulate-db (lambda (x y) (cons (names (name-struct x)) y)) '() db))
```

Alternatively we can do this with `map-db`, again renamed to avoid confusion with the built-in `map`.

```
(define (all-names-in-db db)
  (map-db (lambda (x) (names (name-struct x))) db))
```

For the final task of culling the senior-citizens from the database, we filter the database with the appropriate predicate.

```
(define (over-age-60-in-db db)
  (filter-db (lambda (x) (> (age (age-struct x)) 60)) db))
```

The excessive list structure in our entry format is making things a bit awkward here, so we define the following selectors that directly extract useful information from the entry.

```
(define (ename entry)(names (name-struct entry)))
(define (eage entry) (age (age-struct entry)))
(define (eposn entry)(position (posn-struct entry)))
```

Lab Exercise 2 We define the `same?` procedure as specified so that it applies = to numbers, `eq?` to symbols, and recursively applies itself to pairs.

```
(define (same? l1 l2)
  (cond ((number? l1) (and (number? l2) (= l1 l2)))    ; = to numbers
        ((symbol? l1) (and (symbol? l2) (eq? l1 l2)))  ; eq? to symbols
        ((null? l1) (null? l2))                        ; true if both null
        ((pair? l1)                                    ; if pairs
         (and (pair? l2)                               ; recurse
              (and (same? (car l1) (car l2)) (same? (cdr l1) (cdr l2)))))
        (else #f)))                                   ; false if different
```

For example

```
(same? '(1.0 a 2) '(1 a 2)) ==>
;Value: #t
```

To find the associate professors, we filter the list using this procedure to match the list of symbols, `'(associate professor)`. We could have used the built-in `equal?` for this purpose. However, `(equal? 5 5.0)` ==> `#f` while `(same? 5 5.0)` ==> `#t`.

```
(define (all-assoc-profs-db db)
  (filter-db (lambda (x) (same? (eposn x) '(associate professor))) db))
```

Using the result of part 1 above, then, its easy to find the average age of these characters and list their names.

```
(define (avg-age-assoc-profs-db db)
  (average-age-db (all-assoc-profs-db db)))

(define (names-of-assoc-profs-db db)
  (all-names-in-db (all-assoc-profs-db db)))
```

For example

```
1 ]=> (avg-age-assoc-profs-db db)

;Value: 64.46666666666667

(pretty-print (names-of-assoc-profs-db db))
((xiong hal)        (jones jill)        (wells abraham)    (czerwinski martin)
 (chapin joanne)    (tserng berthold)   (tserng steven)    (miller eric)
 (mikkelson hal)    (abelson berthold)  (jones eric)       (dally arthur)
 (dally chris)      (vaneyk gerry)      (vaneyk hal)       (horn joanne)
 (mikkelson steven) (housinger steven)  (miller berthold)  (stein jill)
 (stark lynn)       (czerwinski steven) (miller rebecca)   (schlecht gerry)
 (brown calista)    (tserng ellen)      (grimson peter)    (maritime albert)
 (jacknis andrew)   (smith yale)        (maritime bill)    (meyer jill)
 (stein peter)      (meyer sandy)       (czerwinski sandy) (stein yale)
 (abelson sandy)    (jordan andrew)     (jacknis hal)      (jacknis john)
 (maritime paul)    (maritime sandy)    (abelson abraham)  (beebee andrew)
 (xiong joanne))
```

Lab Exercise 3 Let's create a few more people

```
(define prez
        (make-struct (make-name 'chuck 'vest)
                     (make-age 55)
                     (make-posn '(president))))

(define provost
        (make-struct (make-name 'joel 'moses)
                     (make-age 54)
                     (make-posn '(provost))))
```

We will put these distinguished individuals in their own database

```
(define exec-db (add-struct prez (add-struct provost empty-db)))
```

Now we define a procedure for merging two databases.

```
(define (merge-dbs db1 db2) ; append by any other name...
  (if (empty-db? db1) db2
      (add-struct (next-db db1) (merge-dbs (rest-db db1) db2))))
```

Finally, we redefine our database, `db`, to include the executives along with the rest of the crew.

```
(define db (merge-dbs exec-db db))
```

Lab Exercise 4 To change the organization of the database, we define new constructor and selector procedures. As an example of *defensive programming* we tag our new database entries with a version-specific tag, `'entry-v1`, and write our selectors so they will work with either version of the database. After the new database has been in operation for some time, it would probably be safe to replace the defensive but slow selectors with more direct versions, but in most applications the small improvement in performance would not be worth the danger of an incorrect result. With the safe approach, we can use a set of databases while they are being converted. There is no need to synchronize the switch of selectors and constructors with the conversion of the database.

```
; constructor and selectors for new entry, old entry, and any entry
(define (make-struct name-st age-st posn-st)
  (list 'entry-v1 name-st posn-st age-st))

(define tag car)
(define (age-struct entry)
  (define age-old-struct caddr)
  (define age-v1-struct cadddr)
  (cond ((eq? (tag entry) 'entry-v1) (age-v1-struct entry))
        ((eq? (tag entry) 'entry) (age-old-struct entry))))
(define (posn-struct entry)
  (define posn-old-struct cadddr)
  (define posn-v1-struct caddr)
  (cond ((eq? (tag entry) 'entry-v1) (posn-v1struct entry))
        ((eq? (tag entry) 'entry) (posn-old-struct entry))))
```

We can now write an *idempotent* routine that converts an entry to the new format. The entire database is converted by mapping this function over the database. Note that we don't use `map` here as that would be an abstraction violation since it uses `cons` rather than `add-struct` to glue together its results.

```
(define (convert-entry entry)
  (make-struct (name-struct entry) (age-struct entry) (posn-struct entry)))
(define (convert-db db)
  (if (empty-db? db) empty-db
      (add-struct (convert-entry (next-db db)) (convert-db (rest-db db)))))
```

For an example, consider the following five-entry database.

```
(define new-db
'((entry (name horn andrew) (age 74) (posn (associate professor)))
  (entry (name abelson eric) (age 96) (posn (associate professor)))
  (entry (name dally eric) (age 35) (posn (assistant professor)))
  (entry (name schlecht paul) (age 62) (posn (professor)))
  (entry (name stein peter) (age 92) (posn (professor)))))

(pp (convert-db new-db)) ==>
((entry-v1 (name horn andrew) (posn (associate professor)) (age 74))
 (entry-v1 (name abelson eric) (posn (associate professor)) (age 96))
 (entry-v1 (name dally eric) (posn (assistant professor)) (age 35))
 (entry-v1 (name schlecht paul) (posn (professor)) (age 62))
 (entry-v1 (name stein peter) (posn (professor)) (age 92)))
```

Lab Exercise 5 To search the database we define a generic `member1?` procedure. I have renamed it here to avoid conflict with `member?` which is used in the utilities that build the random database. This procedure walks the database and returns the first entry, if any, for which `item` matches applying the procedure `proc` to the entry.

```
; returns first entry for which item matches result of applying proc to
; entry - renamed to avoid conflict.
(define (member1? item proc db)
  (cond ((empty-db? db) #f)
        ((same? item (proc (next-db db))) (next-db db))
        (else (member1? item proc (rest-db db)))))
```

Using this procedure, we can now search the database for some curious characters.

```
(member1? '(grimson eric) ename db)
==> (entry (name grimson eric) (age 31) (posn (dean)))

(member1? '(grimson eric) eage db)
==> ()
(member1? '(Sussman Ellen) ename db)
==> (entry (name sussman ellen) (age 70) (posn (administrative assistant)))
(member1? '(sussman ellen) ename db)
==> (entry (name sussman ellen) (age 70) (posn (administrative assistant)))
```

Note that since we have used symbols, rather than strings, to represent names, they are not case sensitive. It would have been a better design decision to represent names with strings and check for equality using `string=?`.

This procedure will be of order n, where n is the number of entries in the database. To see this, note that a random name will appear anywhere in the list, hence on average we'd have to search half the list, $n/2$ entries to find someone. In the worst-case, we will have to search the entire list, n entries. If we double the size of the list, the amount of work to be done doubles, to n on average and $2n$ in the worst-case.

To merge two databases without replication, we test each member of one for inclusion in the other before inserting it.

```
; merges two databases so that no two entries have the same name
(define (disjoint-merge-db db1 db2)
  (if (empty-db? db1) db2
      (let ((entry (next-db db1))
            (rest1 (rest-db db1)))
        (if (member1? (ename entry) ename db2)
            (disjoint-merge-db rest1 db2)
            (add-struct entry (disjoint-merge-db rest1 db2))))))
```

To test this we will make a few small databases with known replicated entries and merge them.

```
(define dbx (merge-dbs (create-random-database 3) exec-db))
(define dby (merge-dbs exec-db (create-random-database 2)))
(define dbz (disjoint-merge-db dbx dby))

(pp dbx) ==>
((entry (name rodriguez steven) (age 85) (posn (administrative assistant)))
 (entry (name sussman yale) (age 56) (posn (associate professor)))
 (entry (name tserng bill) (age 25) (posn (assistant professor)))
 (entry (name chuck vest) (age 55) (posn (president)))
 (entry (name joel moses) (age 54) (posn (provost))))

(pp dby) ==>
((entry (name chuck vest) (age 55) (posn (president)))
 (entry (name joel moses) (age 54) (posn (provost)))
 (entry (name rodriguez steven) (age 69) (posn (associate professor)))
 (entry (name meyer eric) (age 77) (posn (administrative assistant))))

(pp dbz) ==>
((entry (name sussman yale) (age 56) (posn (associate professor)))
 (entry (name tserng bill) (age 25) (posn (assistant professor)))
 (entry (name chuck vest) (age 55) (posn (president)))
 (entry (name joel moses) (age 54) (posn (provost)))
 (entry (name rodriguez steven) (age 69) (posn (associate professor)))
 (entry (name meyer eric) (age 77) (posn (administrative assistant))))
```

As required, the executives show up only once in the database. Also notice that only one entry for `(rodriguez steven)` is included even though the two entries have different ages and positions.

The order of growth of the merge procedure is n^2. One way to see this is to note that each of the items in `db1` must be compared against every item in `db2` in order to see if it is already present in `db2`. If there are n items in each list, then every one of the n items must be compared against n items, giving a total of n^2 operations.

Lab Exercise 6 In an ordered database, we need only search down the list until we find a name that is lexicographically larger than the name we are looking for. Here we define `member-ordered?` to take just a name and a database and scan down the database until it either finds the name or encounters a *larger* name.

```
; returns first entry with name matching argument, assumes database
; is ordered by name
(define (member-ordered? name db)
  (if (empty-db? db) #f
      (let* ((entry (next-db db))
             (rest (rest-db db))
             (aname (ename entry)))
        (cond ((same? name aname) entry)
              ((list<? aname name) (member-ordered? name rest))
              (else #f)))))
```

To insert an entry in the right spot we the following procedure that walks down the list until it finds a name larger than the name we are inserting and inserts the new entry just before that name.

```
; inserts an entry into an ordered database
(define (insert-ordered entry odb)
  (if (empty-db? odb) (add-struct entry odb)
      (let* ((aname (ename entry))
             (dentry (next-db odb))
             (dname (ename dentry))
             (rest (rest-db odb)))
        (cond ((same? aname dname) odb) ; drop duplicate entries
              ((list<? dname aname)
               (add-struct dentry (insert-ordered entry rest))) ; move right
              (else (add-struct entry odb)))))) ; insert here
```

Note that both of these procedures need to compare full names which are lists, not symbols and thus need the utility procedure, `list<?` shown below.

```
; applies alpha<? to a list
(define (list<? l1 l2)
  (cond ((and (symbol? l1) (symbol? l2)) (alpha<? l1 l2))
        ((null? l1) (not (null? l2)))   ; #t if l1 is shorter than l2
        ((and (pair? l1) (pair? l2))
         (if (same? (car l1) (car l2))  ; only compare cdrs if cars same
             (list<? (cdr l1) (cdr l2))
             (list<? (car l1) (car l2))))
        (else #f)))
```

Now we can convert to an ordered db with a little recursion

```
(define (convert-to-ordered-db db)
  (if (empty-db? db) empty-db
      (insert-ordered (next-db db)
                      (convert-to-ordered-db (rest-db db)))))
```

For example

```
(pp dbp) ==>
((entry (name sussman yale) (age 56) (posn (associate professor)))
 (entry (name tserng bill) (age 25) (posn (assistant professor)))
 (entry (name joel moses) (age 54) (posn (provost))))

(pp (convert-to-ordered dbp)) ==>
((entry (name joel moses) (age 54) (posn (provost)))
 (entry (name sussman yale) (age 56) (posn (associate professor)))
 (entry (name tserng bill) (age 25) (posn (assistant professor))))
```

Well, Joel is now in the right place, even if I did get his name backwards.... (This also illustrates a standard problem with databases: no matter how good the code that runs them, they are no better than the data you put in. Yet another instance of GIGO.)

We could recode the `disjoint-merge` above by replacing `member1?` with `member-ordered?` and get some performance improvement, however we would still have poor performance, $\Theta(n^2)$. Consider in particular the case where every entry in the first list is larger than every entry in the second list.

To get linear order of growth for our merge, we use the algorithm shown below which steps down the two ordered lists repeatedly selecting the smallest element from either list to add to the new database being constructed.

```
; merge two ordered databases - result in order - duplicates discarded
; linear in the sum of the sizes of the two databases
(define (ordered-merge db1 db2)
  (cond ((empty-db? db1) db2)
        ((empty-db? db2) db1)
        (else (let* ((next1 (next-db db1)) (next2 (next-db db2))
                     (rest1 (rest-db db1)) (rest2 (rest-db db2))
                     (name1 (ename next1))
                     (name2 (ename next2)))
                (cond ((same? name1 name2) (ordered-merge rest1 db2))
                      ((list<? name1 name2)
                       (add-struct next1 (ordered-merge rest1 db2)))
                      (else
                       (add-struct next2 (ordered-merge db1 rest2)))))))))
```

Lets give it a whirl

```
(ordered-merge (convert-to-ordered-db dbx) (convert-to-ordered-db dby))) ==>
((entry (name chuck vest) (age 55) (posn (president)))
 (entry (name joel moses) (age 54) (posn (provost)))
 (entry (name meyer eric) (age 77) (posn (administrative assistant)))
 (entry (name rodriguez steven) (age 69) (posn (associate professor)))
 (entry (name sussman yale) (age 56) (posn (associate professor)))
 (entry (name tserng bill) (age 25) (posn (assistant professor))))
```

Now consider the order of growth of the above procedures. `member-ordered?` still has both average and worst-case complexity of $O(n)$, since on average we will have to search half-way down the list to either find the entry or find an entry larger than it. For cases where the entry is not found, the search time is cut in half. With the unordered database we had to scan to the end of the list to determine that the entry was not present. However, this *constant factor* improvement is of little importance – it does not change the order of growth.

The insertion procedure is also $O(n)$ as we will on average scan half way down the list to find the point at which we should insert an entry. This is much worse than the $O(1)$ `add-struct` procedure used in the unordered database.

The advantage of the ordered database comes with the merge procedure. Here we have reduced the complexity from n^2 with the unordered database to n with the ordered database. To see that this merge procedure has order of growth, n, consider that it visits each element of each list exactly once. Note that if we had naively converted the `disjoint-merge` procedure above to use the new data structure, we would still have $O(n^2)$ growth for the merge. The algorithm and data structure have to be changed together to realize the advantage.

Lab Exercise 7: By keeping the tree ordered as described in the problem set we can search it by following only a single path from the root to a leaf with no backtracking. Either we find what we are looking for along this path or it is not in the tree. The procedure below carries out this tree search. It compares the name we are searching for to the current node. If it matches we are done; if its less than this node we search the left subtree; if its greater than this node we search the right subtree. If we ever encounter an `empty-tree` we know this name isn't present.

```
; test for membership in a tree
; growth is linear in depth of tree, logarithmic with number of nodes
; assuming tree is balanced
(define (member-tree? name tree)
  (if (empty-tree? tree) #f
      (let* ((tentry (tree-entry tree))
             (tname (ename tentry)))
        (cond ((same? name tname) tentry)
              ((list<? name tname) (member-tree? name (left-branch tree)))
              (else (member-tree? name (right-branch tree)))))))
```

This search has an order of growth equal to the depth of the tree. In the average case the tree is balanced and the depth of the tree (and hence the order of growth) is $log_2(n)$ where n is the number of nodes in the tree. In this case each time we select a branch we cut the size of the remaining search roughly in half by discarding the left or right subtree of a node. Hence doubling the problem size would add only one level to the tree and one step to our search. In the worst-case, however the tree could degenerate into a list structure, with all left branches empty for example. This occurs, for example, if we insert entries into the tree in order, not that unlikely a situation. In this worst case the order of growth is linear in the number of nodes.

Insertion is similar in spirit. Note that this builds new list structure when inserting in the tree. It does not insert *in place* as you might be tempted to do.

```
; inserts an entry in a tree
; growth is linear in depth of tree, logarithmic with number of nodes
; assuming tree is balanced
(define (insert-tree entry tree)
  (if (empty-tree? tree) (make-tree entry empty-tree empty-tree)
      (let ((tentry (tree-entry tree))
            (left (left-branch tree))
            (right (right-branch tree)))
        (if (list<? (ename entry) (ename tentry))
            ; insert into left branch
            (make-tree tentry (insert-tree entry left) right)
            ; insert into right branch
            (make-tree tentry left (insert-tree entry right))))))
```

Now we want to convert from the linear model to the tree:

```
; converts a database to a tree
(define (convert-to-tree db)
  (if (empty-db? db) empty-tree
      (insert-tree (next-db db)
                   (convert-to-tree (rest-db db)))))
```

For example

```
(define tdba (convert-to-tree dba))
tdba  ==>
((entry (name maritime lynn) (age 49) (posn (administrative assistant)))
 ((entry (name jordan martin) (age 39) (posn (dean))) () ())
 ((entry (name meyer peter) (age 88) (posn (dean)))
  ()
  ((entry (name schlecht michael) (age 61) (posn (professor)))
   ((entry (name mikkelson bill) (age 71) (posn (associate professor)))
    ()
    ((entry (name rodriguez steven) (age 34) (posn (associate professor)))
     ((entry (name rodriguez jill) (age 29) (posn (dean)))
      ((entry (name rodriguez andrew) (age 57) (posn (dean))) () ())
      ())
     ()))
   ((entry (name tait chris) (age 35) (posn (assistant professor)))
    ((entry (name stein chris) (age 36) (posn (dean))) () ())
    ())))))

(member-tree? '(tait chris) tdba) ==>
(entry (name tait chris) (age 35) (posn (assistant professor)))

(member-tree? '(no name) tdba) ==>
()
```

Lab Exercise 8 To linearize the tree we visit each node, linearize the left and right subtrees, and then glue together the three pieces, left subtree, node entry, and right subtree, with `merge-dbs` and `add-struct`.

```
(define (linearize-tree tree)
  (if (empty-tree? tree) empty-db
      (merge-dbs (linearize-tree (left-branch tree))
                 (merge-dbs (add-struct (tree-entry tree) empty-db)
                            (linearize-tree (right-branch tree))))))
```

For example:

```
(define sample-tree-database (convert-to-tree sample-linear-database))
(define sample-linearized-database (linearize-tree sample-tree-database))

(pp(first-few sample-linearized-database 5)) ==>
((entry (name abelson abraham) (age 30) (posn (associate professor)))
 (entry (name abelson berthold) (age 25) (posn (associate professor)))
 (entry (name abelson berthold) (age 53) (posn (assistant professor)))
 (entry (name abelson calista) (age 74) (posn (assistant professor)))
 (entry (name abelson lynn) (age 83) (posn (assistant professor))))
```