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

Problem Set 5

Indexing and Databases

Issued: Tuesday, March 4, 1997

Due: In recitation on Wednesday, March 12 for recitations held at 9, 12, 1 and 2;
and on Friday, March 14 for recitations held at 10 and 11.

Tutorial Preparation: For the week of March 10. Reading Assignment: Through the end of Chapter 2 of *SICP.*

REMINDER: Quiz 1 is on Wednesday, March 19.

In this assignment, you will work with a simple personnel data base. The goal of this problem set is to explore issues in symbolic data representation, and to explore the tradeoffs associated with different representations of data, such as computational complexity and ease of programming.

1. Representation of Symbolic Information

In this data set, we are going to explore a simple database, different ways of structuring the information in such a database, and the implications for such choices on the speed and ease of use of such data bases.

In our simple case, each entry of the data base will contain some basic information, both numeric and symbolic. For example, in the personnel database we will examine, we store name, age and position information about each of the people in the data base.

Basic data structures

Each structure in our database will be built with the constructor:

```
(define (make-struct name-st age-st posn-st)
  (list 'entry name-st age-st posn-st))
```

which makes each entry a simple list, marked with a label at the front.

Associated with this constructor are a set of selectors:

```
(define name-struct cadr)
(define age-struct caddr)
(define posn-struct cadddr)
```

Each element of an entry is itself a structure, with associated constructors and selectors:

```
(define (make-name sur given)
  (list 'name sur given))

(define surname cadr)
(define given-name caddr)

(define names cdr)

(define (make-age age)
  (list 'age age))

(define age cadr)

(define (make-posn p)
  (list 'posn p)))

(define position cadr)
```

The database structures

The first organization for our database will simply build on our existing list structure:

```
(define empty-db? null?)
(define empty-db '())
(define next-db car)
(define rest-db cdr)
(define add-struct cons)
```

In other words, we have a base database which is simply the empty list, and we can test for an empty data base by testing for the empty list. We can get the next entry in the database by using the procedure for extracting the next element of a list, and we can add an element to the database by utilizing the constructor for adding an element to a list.

The second organization for our database will also build on the list structure, but will impose the condition that the elements of the list are ordered in some manner (in our case, we will do the ordering by imposing a lexicographical ordering on the name structures of each of the entries, much like in a phone book). Thus the basic constructors and selectors can remain mostly the same.

The third organization for our database will use a binary tree structure, similar to what is discussed in the notes. Here the data abstraction is built around a tree entry:

```
(define tree-entry car)
(define left-branch cadr)
(define right-branch caddr)
(define (make-tree entry l r)
  (list entry l r))
(define empty-tree? null?)
(define empty-tree '())
```

In other words, an node in a tree consists of the entry at that node and a left and right branch below that node. We will again utilize an empty list to correspond to an empty node.

To test the procedures you will create in this problem set, we have built a somewhat random database, whose name is `linear-test-data`. A sample entry from this data base is given below:

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

Since we have chosen to represent our database as a list, note that at least initially, you can use simple list procedures to examine elements of the database.

Because we are using lists to represent our basic databases, we have some generic operations on lists that will be convenient:

```
(define (accumulate op init items)
  (cond ((null? items) init)
        (else (op (car items) (accumulate op init (cdr items))))))

(define (filter predicate items)
  (cond ((null? items) '())
        ((predicate (car items))
         (cons (car items) (filter predicate (cdr items))))
        (else (filter predicate (cdr items)))))

(define (map proc items)
  (if (null? items)
      '()
       (cons (proc (car items))
             (map proc (cdr items)))))
```

Tutorial exercises

You should prepare the following exercises for oral presentation in tutorial. They cover material in sections 2.2 and 2.3 of the text, and they also test your understanding of the material to be used in manipulating our databases, in preparation for doing this week's lab in. You may wish to use the computer to check your answers to these questions, but you should try to do them (at least initially) without the computer.

Tutorial exercise 1: Draw a box and pointer diagram for the sample entry from the database show above.

Tutorial exercise 2: Do exercise 2.26 from the notes.

Tutorial exercise 3: Do exercise 2.28 from the notes.

Tutorial exercise 4: Do exercise 2.37 from the notes.

Tutorial exercise 5: Do exercise 2.54 from the notes.

Manipulating our database

Lab exercise 1: Given the sample database `linear-test-data`, we can gather information about the entire data base, using the generic procedures `map, filter` and/or `accumulate`.

- Write a procedure that determines the average age of the entries of the data base.
- Write a procedure that creates a list of all the names in the database (where a "name" is itself a list containing a surname and a given name).
- Write a procedure that creates a list of all the people over 60 in the database.

Turn in listings of your procedures and examples of their evaluation.

Lab exercise 2: Suppose that we want to find all the associate professors in our database. We can certainly use our `filter` operation to extract the entries, provided we can figure out the right thing to use as a predicate.

To compare symbols, we have provided two procedures:

- `eq?` returns true if the two arguments are equal,
- `alpha<?` returns true if the first argument is lexicographically less than the second.

You should try some examples to determine the details of how these predicates perform.

The problem with which we are left is that entries in the position field (or the name field for that matter) are presented as lists of strings.

- Create a procedure, called `same-list?`, that tests the equality of two lists of symbols, where equality means that the lists must be of the same length and each item in order must match the corresponding item of the other list.

- Using that procedure, create a procedure that will return a list of all the associate professors in the database.
- Create a procedure to find out the average age of the associate professors.
- Create a procedure to create a list of the names of all the associate professors.

Turn in a listing of your procedures and examples of their evaluation.

Lab exercise 3: Create some new entries, and insert them into the database. Turn in a listing demonstrating how you did this.

Lab exercise 4: Suppose we decide to change the order of the structures in an entry to be *name, position* and then *age*. Write a procedure that will convert the current database to this form. What other changes must you make to the database system? Turn in a listing of your procedures and other changes.

Lab exercise 5: We are ready to create a procedure to let us test membership in the database, which is a common procedure that we will want to use in many other procedures. Of course, we have to decide what we want to use to determine membership, and for simplicity, we will assume that all the names of people in the database are unique. Using the procedure you created in Lab Exercise 2 for testing equality of lists of symbols, write a procedure called `member?` with the property that evaluating a form such as

```
(member? '(grimson eric) linear-test-data)
```

will return false if there is no entry in the database with that name, and will return the full entry if there is one. Turn in a listing of your procedure, as well as examples of its evaluation on:

```
(member? '(grimson eric) linear-test-data)
(member? '(rodriguez gerry) linear-test-data)
(member? '(Sussman Ellen) linear-test-data)
(member? '(sussman ellen) linear-test-data)
```

What is the order of growth associated with using `member?`?

Once you have created `member?` you can use it to build other useful database procedures. For example, suppose we have two databases that need to be merged. This means that we want to create a single database with the property that no entry is duplicated. Write a procedure that achieves this, and turn in a listing. What is the order of growth associated with your procedure? Create a small database and merge it with our test database, and turn in a listing showing that your procedure works correctly. (Note that the TAs don't really want to see an copy of the original database! Choose your test so that it clearly indicates that your procedure works correctly, without bogging the reader down in lots of irrelevant details.)

Lab exercise 6: Now we want to consider converting our simple database into one with a bit more structure. In particular, we want our new database to again have a list as a top level structure, but now we want our entries to be ordered lexicographically by name.

To do this, we will need a new membership predicate, `member-ordered?` which, as with `member?`, either returns false if the element is not a member of the database, or returns the actual entry associated with that name in the database. You should be able to take advantage of the ordering of the database to make a more efficient predicate. Write `member-ordered?`, remembering that `eq?` can be used to check equality of symbols, and `alpha<?` can be used to check ordering of symbols. Turn in a listing of your procedure.

Similar to `member-ordered?`, we can create a procedure for inserting an entry into an ordered database. Create such a procedure `insert-ordered`, with calling pattern `(insert-ordered elt ordered-db)` where `elt` is a structure made with `make-struct` and `ordered-db` is an ordered database. `Insert-ordered` should create a new version of the database with the element inserted in the appropriate position. Turn in a listing of your procedure.

Finally, create a procedure `convert-to-ordered-db` which takes a standard unordered database as input, and returns an ordered database. Turn in a listing of your procedure. Test by examining the first few entries of your database.

Suppose we wanted to use the operation of merging in this new ordered data base. What changes do we need to make? What is the order of growth of the new version of `merge`? Is this new ordered database an improvement over the original form?

Lab exercise 7: The third version of our database will be built using a binary tree structure. Each element of a tree is itself a tree (hence the nice recursion) and is either an empty tree (which we represent as an `empty-db`) or consist of three elements: an entry, a left branch, and a right branch. The entry is a standard entry as in the previous cases, while the left and right branches are trees. The tree has a nice ordering property, which is that all the elements of the left branch of the tree are "less than" the entry of the tree, which is "less than" all of the elements of the right branch (where we are again using lexicographical ordering on the name to determine "less than").

Similar to Lab Exercise 6, create procedures `member-tree?, insert-tree,` and `convert-to-tree` for testing membership in a tree structured database, inserting an element into a tree structured database and converting a list structured database into a tree-structured database.

Turn in listings of your procedures.

What is the order of growth associated with `member-tree?` and with `insert-tree`?

Lab exercise 8: Use your `convert-to-tree` procedure to generate a tree-structured version of the database. Test out the new database by writing a procedure that linearizes the tree back to a list, i.e. that collects the entries of the left branch of a node, plus the actual entry of a node, plus the entries of the right branch of a node into a list. Apply this procedure to your tree database, then use that to generate a list of the names of the linearized version. Turn in a listing of your procedure, plus a listing of the first few elements of the result, indicating that the names are in lexicographic order.

How much time did you spend on this homework assignment? Report separately time spent before going to lab and time spent in the 6.001 lab.

If you cooperated with other students working this problem set please indicate their names on your solutions. As you should know, as long as the guidelines described in the *6.001 Policy on Cooperation* handout are followed, such cooperation is allowed and encouraged.