Computer Science 320
Concepts of Programming Languages

# Problem Set 3: Abstract Data Types

Out: Sunday, February 5, 2005
Due: ***11:59 PM*** on Friday February 10, 2005

***There are 4 problems in this set, each worth as indicated, for a total of 100 points. The harder problems are marked with a single * (average difficulty) or two ** (higher-than-average difficulty). For the easy points, start with the unmarked problems.***

This assignment will use programming techniques from the preceding two problem sets: You will have to deal with *lists*, with *recursion on list structures*, and with *higher-order procedures.* In addition, it will make precise, through an extended example, the concept of an *abstract data type.* This notion was already presented in lecture, e.g., see Handout HD03.

## An Abstract Data Type for the Algebra of Integer Matrices

An *abstract data type*, or ADT for short, is a fundamental concept in programming: It is a package that organizes data in some hierachical structure, together with an interface of routines (or procedures) that manipulate that data, according to specifications that are independent of the details of how the data is stored and how these routines are implemented.

An ADT can be defined in almost any programming language. For example, the interface of an ADT for manipulating matrices of integers can be specified in as follows, which we call $\mathcal{ADT}$-1 for later reference:

1. `(matrix-zero` $\langle m \rangle$ $\langle n \rangle$`)`
   returns a $m \times n$ matrix with all entries `0`.

2. `(dot-product` $\langle \mathit{vector}_1 \rangle$ $\langle \mathit{vector}_2 \rangle$`)`
   returns the dot-product of two vectors of the same dimension.

3. `(matrix-*-vector` $\langle \mathit{matrix} \rangle$ $\langle \mathit{vector} \rangle$`)`
   returns the product of a $m \times n$ matrix and a $n$-dimensional vector.

4. `(matrix-+-matrix` $\langle \mathit{matrix}_1 \rangle$ $\langle \mathit{matrix}_2 \rangle$`)`
   returns the sum of two $m \times n$ matrices.

5. `(matrix-*-matrix` $\langle \mathit{matrix}_1 \rangle$ $\langle \mathit{matrix}_2 \rangle$`)`
   returns the product of a $m \times n$ matrix $\langle \mathit{matrix}_1 \rangle$ and a $n \times p$ matrix $\langle \mathit{matrix}_2 \rangle$.

6. `(transpose` $\langle \mathit{matrix} \rangle$`)`
   returns the transpose of $\langle \mathit{matrix} \rangle$.

7. `(equal-matrices?` $\langle \mathit{matrix}_1 \rangle$ $\langle \mathit{matrix}_2 \rangle$`)`
   returns `#t` if $\langle \mathit{matrix}_1 \rangle$ and $\langle \mathit{matrix}_2 \rangle$ are equal matrices, and `#f` otherwise.

8. `(zero-matrix?` $\langle \mathit{matrix} \rangle$`)`
   returns `#t` if $\langle \mathit{matrix} \rangle$ is a matrix with all its entries `0`, and `#f` otherwise.

These are not the only meaningful operations on matrices. Programmers may add others that are relevant to whatever application they work on. For example, what is called *Gaussian elimination* to solve independent linear equations is in fact about computing *determinants* and *inverses* of matrices, which require other operations besides the 8 listed above. For the purposes of this assignment, we restrict attention to these 8, but keep in mind that ADT specification in general is open to any adaptation called for by programmers' needs.

∗ $\boxed{\textbf{Problem 1}}$ *(30 points)* Exercise 2.37, page 120, in the [A&S] book. You are asked to provide a particular implementation for some of the functions (functions 3, 5 and 6) in $\mathcal{ADT}$-1 described above.

In Problems 2 and 3, we continue to represent an integer matrix as a list of (equal-length) rows, where each row is a list of integers. Thus, the following $3 \times 4$ matrix:

```
1 2 3 4
4 5 6 6
6 7 8 9
```

is represented by the list of (equal-length) integer lists:

```
((1 2 3 4) (4 5 6 6) (6 7 8 9))
```

Note `(1 2 3 4)` is not the representation of a $1 \times 4$ matrix, but the representation of a 4-dimensional vector. A $1 \times 4$ matrix with the same entries is represented by `((1 2 3 4))`. A $4 \times 1$ matrix is represented by a list of 4 one-element lists, e.g., `((1) (2) (3) (4))`.

∗ $\boxed{\textbf{Problem 2}}$ *(30 points)* We want new implementations of the `transpose` function, different from the one in Problem 1. The transpose of the matrix above is:

```
1 4 6
2 5 7
3 6 8
4 6 9
```

which is represented by the following list of integer lists:

```
((1 4 6) (2 5 7) (3 6 8) (4 6 9))
```

One way to transpose a matrix is by repeatedly computing the columns of the matrix from left to right. The heads of the rows of the input matrix, which make up its leftmost column, is the top row of the output matrix:

```
(define (form-top-row matrix)
  (if (null? matrix)
      '()
      (let ((top-row (car matrix))
            (lower-rows (cdr matrix)))
        (cons (car top-row) (form-top-row lower-rows)))))
```

The tails of the rows form a matrix of the remaining columns:

```
(define (form-remaining-cols matrix)
  (if (null? matrix)
      '()
      (let ((top-row (car matrix))
            (lower-rows (cdr matrix)))
        (cons (cdr top-row) (form-remaining-cols lower-rows)))))
```

For example, calling `form-top-row` and `form-remaining-cols` breaks the matrix in the paragraph preceding Problem 2 in two parts, by returning a 3-dimensional vector and a $3 \times 3$ matrix, respectively:

```
1 4 6          2 3 4
               5 6 6
               7 8 9
```

We can now implement the transpose function as follows:

```
(define (transpose-1 matrix)
   (if (null? (car matrix))
       '()
       (cons (form-top-row matrix)
             (transpose-1 (form-remaining-cols matrix)))))
```

We call it `transpose-1` to distinguish it from the `transpose` in Problem 1.

1. What does `transpose-1` return if the rows of the "matrix" do not have the same length? Some cases cause an error, and some cases do not. Classify these cases.
2. What happens if the test `(null? (car matrix))` in the function `transpose-1` is replaced by the test `(not (null? (filter null? matrix)))`? Explain in a couple of sentences.
3. Write an alternative transpose function, call it `transpose-2` which, instead of turning columns into rows, turns rows into columns. (To get credit for this problem, you have to write `transpose-2` in the style of `transpose-1`.)

**Problem 3** *(15 points)* In terms of the higher-order procedure `map`, the method used by `transpose-1` to carry out the transpose of a matrix can be implemented more concisely and directly as follows:

```
(define (transpose-3 matrix)
   (if (null? (car matrix))
       '()
       (cons (map car matrix) (transpose-3 (map cdr matrix)))))
```

In the same style, i.e., using `map` (and other procedues), write an implementation of the method of `transpose-2` in terms of the higher-order procedure `map`. Call the resulting function `transpose-4`.

## A More Generic Abstract Data Type for Matrix Algebra

Our specification of $\mathcal{ADT}$-1 for the algebra of integer matrices is not "abstract" (or "generic") enough, in the sense that there are other kinds of matrices — boolean matrices, real matrices, and other kinds still — that have many properties in common with integer matrices and their operations. To abstract the common properties of all these kinds of matrices and their algebras, we adjust $\mathcal{ADT}$-1 so that a new parameter, called *matrix type*, is taken into account. We call the entries in the matrices *basic values*, which can be integers, real numbers, or booleans in this assignment. The new adjusted ADT is called $\mathcal{ADT}$-2. We would like a matrix type to specify several things, 4 of them for this assignment:

1. A unique basic value which is a "zero" relative to a binary additive operation and a binary multiplicative operation on basic values. We call this value *basic zero*. For example, for the set of booleans as basic values, we can take `#f` as the basic zero relative to "`or`" (as an additive operation) and "`and`" (as a multiplicative operation). Note the analogy between `0` and `#f`:

$$\begin{array}{llll} x+0=0+x=x & \text{for every integer } x & x \vee \text{false} = \text{false} \vee x = x & \text{for every boolean } x \\ x \times 0 = 0 \times x = 0 & \text{for every integer } x & x \wedge \text{false} = \text{false} \wedge x = \text{false} & \text{for every boolean } x \end{array}$$

2. An equality predicate on basic values, which we call *basic equality.*

3. A binary additive operation on basic values, which we call *basic sum.* For example, for the set of booleans as basic values, we can take "`or`" as basic sum.

4. A binary multiplicative operation on basic values, which we call *basic product.* For example, for the set of booleans as basic values, we can take "`and`" as basic product.

Representing a matrix type as a list of the 4 parts above, here are the Scheme declarations we want you to use:

```
(define int-matrix-type
  (lambda (m)
    (cond ((eq? m 'zero) 0)
          ((eq? m 'eq)   (lambda (x y) (= x y)))
          ((eq? m 'add)  (lambda (x y) (+ x y)))
          ((eq? m 'mult) (lambda (x y) (* x y))))))
(define bool-matrix-type
  (lambda (m)
    (cond ((eq? m 'zero) ())
          ((eq? m 'eq)   (lambda (x y) (eq? x y)))
          ((eq? m 'add)  (lambda (x y) (or  x y)))
          ((eq? m 'mult) (lambda (x y) (and x y))))))
(define real-matrix-type
  (lambda (m)
    (cond ((eq? m 'zero) 0)
          ((eq? m 'eq)   (lambda (x y) (= x y)))
          ((eq? m 'add)  (lambda (x y) (+ x y)))
          ((eq? m 'mult) (lambda (x y) (* x y))))))
(define new-int-matrix-type
  (lambda (m)
    (cond ((eq? m 'zero) 999999)
          ((eq? m 'eq)   (lambda (x y) (=   x y)))
          ((eq? m 'add)  (lambda (x y) (min x y)))
          ((eq? m 'mult) (lambda (x y) (+   x y))))))
```

We define `int-matrix-type` to be as shown above in order to enforce the requirement that the basic equality, basic sum and basic product, are *binary* operations. (In Scheme, =, + and * can each consume any finite number of arguments, not just 2.) The same comment applies to the definition of `bool-matrix-type`, `real-matrix-type` and `new-int-matrix-type`.

** $\boxed{\textbf{Problem 4}}$ *(25 points)* $\mathcal{ADT}$-2 is obtained from $\mathcal{ADT}$-1 by taking into the account the information provided by a matrix type.

1. Propose a design of $\mathcal{ADT}$-2 that will accomplish this goal. (There is more than one possible design. You will get credit for proposing "something appropriate".) Your design should be reflected in the way you write the interface for $\mathcal{ADT}$-2, which is what you have to turn in to get credit. *Hint*: Inspire yourself by the presentation for $\mathcal{ADT}$-1 preceding Problem 1.

2. Based on your design of $\mathcal{ADT}$-2, write an implementation of the function `dot-product`. *Hint*: Adjust the code for `dot-product` in Problem 1 (given at the top of page 121 in [A&S]).