Computer Science 320
Concepts of Programming Languages
# Problem Set 4: Mutable Data Objects

Out: Friday, February 10, 2006
Due: ***4:59 PM*** on Friday, February 17, 2005

***There are 5 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.***

In lecture this past week, we talked about complications resulting from the use of assignments and their side-effects: loss of *referential transparency*, unexpected results from (careless) *aliasing*, difficulty in understanding their meaning (which call for complicated semantic models, such as the *environment model*). We also talked a little about their benefits, but an extra measure of care is necessary if we want to avoid their pitfalls. We continue this discussion here, by considering several situations for which programming with side-effects is beneficial.

The software for this assignment is in a single directory called `Scheme-code-for-PSet04`. We made a "tar" file of this directory,[1] called `Scheme-code-for-PSet04.tar` which can be downloaded from the course webpage. To restore the directory `Scheme-code-for-PSet04` and its contents on your local machine, issue the following Unix command:

```
tar -xvf Scheme-code-for-PSet04.tar
```

and hit RETURN. You will need the provided software for Problems 3, 4 and 5.

## 1 Side-Effects Can Be Useful

**Problem 1** *(20 points)* Write a Scheme procedure, called "`remember`", which takes a single integer as input and then returns the sum of the current input and its previous input. To do this, `remember` must store its last input. Here is a possible session:

```
=> (remember 1)
;Value: ''First time''
=> (remember 2)
;Value: 3
=> (remember 3)
;Value: 5
```

**Problem 2** *(20 points)* Write a Scheme procedure, called "`delete-odd!`", which deletes all the odd numbers in a list of numbers. The procedure must alter, i.e., mutate, the input list, *not* return a new list. (And this is why we call it "`delete-odd!`" rather than "`delete-odd`".) For example, executing the following code:

[1] For the curious, file `Scheme-code-for-PSet04.tar` was generated by issuing the following command at the Unix prompt: "`tar -cvf Scheme-code-for-PSet04.tar Scheme-code-for-PSet04`".

```
(define x (list 1 2 3 4))
(set! x (delete-odd! x))
x
```

should return the list `(2 4)`.

## 2 Tables As Mutable Data Structures

Although it is possible to represent and manipulate tables in a pure functional style – see, for example, Section 2.4.3 in the [A&S] book, pp 179-187 – tables are generally meant to store bulky data records, which are more efficiently represented by mutable list structures. An implementation of a (one-dimensional) table is in file `tables.scm` in the directory `Scheme-code-for-PSet04`.

* **Problem 3** *(20 points)* The implementation of the procedure `assoc` in the file `tables.scm` uses the predefined `equal?` to test whether two keys are equal. This is not always an appropriate test. For instance, we may have a table with numeric keys in which we do not need an exact match to the number we are looking up, but only a number within an approximate bound from it (below or above).

1. Write the code for a new procedure `approx-assoc`, which is a variation of procedure `assoc` in that it takes a third argument `same-key?` to test whether two keys are within a given bound from each other.

2. Adjust the code for `lookup` and `insert!` accordingly. Call the new procedures `approx-lookup` and `approx-insert!`. In your adjustment of `insert!` you have to decide whether or not to insert a new record whose key is within the given bound from a key already in the table: State your decision clearly and write `approx-insert!` accordingly.

## 3 An Application of Tables: Memoization

*Memoization* is a programming technique to avoid recomputation. Many applications call for the repeated generation of different members of the same collection of objects. To do this without careful consideration of the required resources may not be computationally feasible.

This problem is encountered with the Fibonacci function (see, for example, Handout 10). In the case of Fibonacci, the obvious implementation is not practically feasible and cannot be used on input values much larger than 20 or 25. A more clever (but less transparent) implementation will terminate and return an output on an input reaching 5000 or more. In lecture, our implementation of `fast-fib` used memoization; it was a somewhat *ad-hoc* approach to memoization, which worked only for Fibonacci, based on our understanding of the function.

In the problem below, a systematic approach to memoization is developed, which can be applied to any one-argument function, including Fibonacci.

** **Problem 4** *(20 points)* Do Exercise 3.27 in the [A&S] text, on page 272. At the end of the exercise, on page 273, there are three parts: Ignore the first part ("Draw an environment diagram $\cdots$"), and only do the two last parts:

1. Explain why `memo-fib` computes the $n$th Fibonacci number in a number of steps proportional to $n$.

2. Would the scheme still workif we had simply defined `memo-fib` to be `(memoize fib)`?

The code for this problem is provided in the three files `tables.scm`, `memoize.scm` and `memo-fib.scm`, all in the directory `Scheme-code-for-PSet04`.

## 4 Message-Passing Style

In file `bank.scm` in the directory `Scheme-code-for-PSet04`, you will find the code of the procedure `make-account` to "create banking accounts" on page 223 of the [A&S] book. (The same code is also reproduced in Handout 12.) In the next problem you need to modify the code of `make-account`.

* **Problem 5** *(20 points)*

1. Exercise 3.3, page 225, in [A&S].
2. Exercise 3.4, page 225, in [A&S].