CS61A Summer 2010

George Wang, Jonathan Kotker, Seshadri Mahalingam, Steven Tang, Eric Tzeng

Homework 5

Due: Monday, July 26 2010, at 7AM

*N*ote: The number of stars (★) besides a question represents the *estimated* relative difficulty of the problem: the more the number of stars, the harder the question.

1 How to Start and Submit

First, download `http://inst.eecs.berkeley.edu/~cs61a/su10/hw/hw5.scm` and use it as a template while filling out your procedures. See `http://www-inst.eecs.berkeley.edu/~cs61a/su10/hw-faq.pdf` for submission instructions.

2 Mutable Data

1. (★★) Draw a box and pointer diagram for the following sequence of expressions.

```
> (define a (list 1 2 3))
> (define b (list 1 2 3))
> (define c (cons a b))
> (define d (list c (cdr a) (cadr b)))
> (define e (list 0 1 2))
> (set-cdr! (cdr e) (cdadr d))
```

2. (★) Write the procedure `append!` that, given two lists, appends the second list onto the first using ONLY list mutation: *do not create any new pairs.*

   Examples:

```
> (define a (list 1 2 3))
> (define b (list 4 5 6))
> (append! a b)
> a
(1 2 3 4 5 6)
> b
(4 5 6)
```

3. (★★) Write the procedure `merge!` that, given two lists (both are lists of numbers only and both are in increasing order) combines the list such that the resulting list is in increasing numerical order as well. Use ONLY list mutation, *d*o not create any new pairs.

   Examples:

```
> (merge! (list 2 3 5 9) (list 1 6 7 11))
(1 2 3 5 6 7 9 11)
```

4. (★★) SICP Exercises: `count-pairs`
   (a) 3.16
   (b) 3.17

5. (★★) Queue Implementation: SICP Exercise 3.21.

   We strongly recommend reading the section on queues in SICP before attempting the question.

6. (★★★) (Optional) Memoization: SICP Exercise 3.27

3 Vectors

*N*ote: For the following questions, please do not use `list->vector`, `vector->list` or `apply`. Use vector procedures like `make-vector` and `vector-set!`.

1. (★) Write a procedure called `vsearch` that searches an unsorted vector for a given element and returns the location index of that item (remember that vectors, like lists, are numbered starting at 0). If the element is not found in the vector, return -1. In general, does this seem like this would take more time, less time, or the same amount of time doing such a search on a list?

   Examples:

```
> (define sample-vec (make-vector 5 12)
#(12 12 12 12 12)
> (vector-set! sample-vec 3 6)
okay
> sample-vec
#(12 12 12 6 12)
> (vsearch sample-vec 6)
3
> (vsearch 'q (vector 'h 'i 'j 'k 'l))
-1
```

2. (★★) Write a procedure called `vector-append` for vectors that appends two vectors, like the 'append' procedure does for lists.

   Examples:

```
> (vector-append (make-vector 3 'a) (make-vector 2 'q))
#(a a a q q)
> (vector-append (vector 'a 'b 'c) (vector 1 2 3))
#(a b c 1 2 3)
```

3. (★) Write a procedure `vector-reverse` that takes a vector and returns a new vector with the entries of the given vector reversed.

   Examples:

```
> (vector-reverse (vector 'a 'b 'c))
#(c b a)
> (vector-reverse (vector 1))
#(1)
```

4. (★★★) Write a procedure `vector-filter` that takes a predicate function and a vector as arguments and returns a **new** vector containing only those elements of the argument vector for which the predicate was true. The new vector must exactly big enough for these elements that are true in the predicate.

   How does the runtime of your procedure compare with this:

```
(define (vector-filter pred vec)
    (list->vector (filter pred (vector->list vec))))
```

5. (★★) Remember `vector-reverse`? Great! Write the procedure `vector-reverse!` that reverses a given vector "in place". In other words, do **not** create a new vector; use the given one only.

Examples:

```
> (define sample-vec (vector 1 2 3 4 5))
> (vector-reverse! sample-vec)
#(5 4 3 2 1)
> sample-vec
#(5 4 3 2 1)
```

4 Meta-Circular Evaluator

1. The following *SICP* exercises are designed to familiarize yourself with the structure of the metacircular evaluator by incrementally adding features. We recommend that you do the following exercises in the given order. Some are less directed and more open-ended than others and require you to make and/or justify design decisions.
   - (★) 4.2(a) : `cond` clauses
   - (★★) 4.4 : `and` & `or`
   - (★★) 4.5 : `(<test> => <recipient>) cond clauses`
   - (★★) 4.6 : `let`
   - (★★) (Optional) 4.7 : `let*`
   - (★) 4.13 : `make-unbound!`
   - (★★★) 4.14 : Why doesn't primitive `map` work?
2. (★★★) (Optional) Modify the metacircular evaluator to allow *type-checking* of arguments to procedures. Here's how the feature should work:

   When a new procedure is defined, a formal parameter can be either a symbol (as usual) **or** a list of two elements. In the second case: The first value will be a predicate procedure of one argument that returns `#t` if the argument is of the desired type. The second value will be the name of the formal parameter.

   Example:

```
MCE> (define (foo (integer? num) ((lambda (x) (not (null? x))) list))
(list-ref list num))
foo
MCE> (foo 3 '(a b c d e))
d
MCE> (foo 3.5 '(a b c d e))
Error: wrong argument type -- 3.5
MCE> (foo 2 '())
Error: wrong argument type -- ()
```

5 Extra for Experts (Optional)

SICP Exercises 4.16 - 4.21. These exercises explore the subtle differences in how variables are defined and scoped, using recursion, mutual recursion and environment diagrams.

6 Feedback

Now that you are done, please leave me some feedback at the following link regarding how the course is going. This is not worth points, but will give us valuable feedback that in turn improves your course experience. Thanks! `https://spreadsheets.google.com/viewform?formkey=dDlZX1I2OGc1dk5GOEN3Y0VqZi1zUnc6MQ`