CS61A – Homework 2.1	**Kurt Meinz**
University of California, Berkeley	Summer 2003

Topic: Recursion and iteration

Lectures: Monday June 30, Tuesday July 1

Reading: Abelson & Sussman, Section 1.2 through 1.2.4 (Pages 31–47)

In this assignment you'll practice writing procedures that evolve iterative processes. The homework is due at **8 PM on Sunday, July 6**. Please put your solutions into a file called `hw2-1.scm` and submit electronically by typing `submit hw2-1` in the appropriate directory. Include test cases and make sure that your `.scm` files loads without errors.

Question 1. You've seen the `keep` higher-order function in lecture. It takes two arguments: a predicate and a sentence. It returns a new sentence of only those elements that satisfy the predicate (i.e. those for which the predicate returns a true value):

```
STk> (keep odd? '(1 2 3 4 5 6 7))
(1 3 5 7)
STk> (keep (lambda (x) (equal? x 'foo)) '(follow the white rabbit))
()
```

Write `keep` so it generates an iterative process.

Question 2. The `fast-expt` procedure presented on Page 45 performs exponentiation in a logarithmic number of steps using successive squaring. Its order of growth is approximately $\Theta(log_2(n))$, which is pretty damn good. However, the book's version evolves a recursive process: each time n is even a call to `square` is left to be done before the function returns. Re-write `fast-expt` so it evolves an iterative process (and still uses a logarithmic number of steps, of course). The idea behind successive squaring is:

$$b^n = (b^{\frac{n}{2}})^2 = (b^2)^{\frac{n}{2}}$$

To adapt this to an iterative algorithm, you'll need to maintain an extra iteration variable, call it a for "answer," that is taken to be 1 initially; the final value of a will be the result of `fast-expt`. The value of ab^n should not change from one iteration to the next. In other words, ab^n should remain *invariant* throughout the computation. The individual values of a, b and n may change from iteration to iteration.

```
STk> (fast-expt 3 6)
729
STk> (fast-expt 2 32)
4294967296
```

The adventure continues on the next page.

Question 3. Read and complete Exercise 1.37 from SICP. Don't get intimidated by the math. This question has *nothing* to do with ϕ, the special number 1.6180, except that its inverse can be approximated with the continued fraction:

$$\cfrac{1}{1+\cfrac{1}{1+\cfrac{1}{1+\cdots}}}$$

You don't need to understand the mathematical significance of ϕ. However, your `cont-frac` function should give a good approximation to $\frac{1}{\phi}$:

```
STk> (cont-frac (lambda (i) 1.0) (lambda (x) 1.0) 100)
0.618033988749895
```

But before you start approximating $\frac{1}{\phi}$, test your function with a small k-term finite continued fraction like:

$$\cfrac{1}{1+\cfrac{2}{2+\cfrac{3}{3}}}$$

There are just three terms in this fraction, making it easy to compute by hand:

```
STk> (/ 1 (+ 1 (/ 2 (+ 2 (/ 3 3)))))
0.6
```

Using `cont-frac` should give matching results:

```
STk> (cont-frac (lambda (x) x) (lambda (x) x) 3)
0.6
```

Hint: You will find it easier to count up from one to k in the recursive version, and to count down from k to zero in the iterative version.

Question 4. A *perfect number* is defined as a number equal to the sum of all its factors less than itself. For example, the first perfect number is 6, because $1+2+3=6$. The second perfect number is 28, because $1+2+4+7+14=28$. What is the third perfect number? Write a procedure `next-perfect` that takes a single number n and tests numbers starting with n until a perfect number is found:

```
STk> (next-perfect 4)
6
STk> (next-perfect 6)
6
STk> (next-perfect 7)
28
```

To find the third perfect number evaluate `(next-perf 29)`. To do this problem, you'll need a `sum-of-factors` subprocedure. If you run this program when the system is heavily loaded, it may take a while to compute the answer! Make sure your program can find 6 and 28 first.

Does `next-perfect` evolve an iterative or recursive process?