MASSACHUSETTS INSTITUTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.001—Structure and Interpretation of Computer Programs
Fall Semester, 1989

Problem set 2

Issued:	Tuesday 19 September
Due:	recitation, Wednesday 27 September
Reading:	From text, Chapter 1, sections 1.2 and 1.3. (This completes chapter 1.)

Problem sets should always be handed in at recitation. Late work will not be accepted.

Homework exercises

Write up and turn in the following exercises from the text:

- Exercise 1.11: Iterative exponentiation
- Exercise 1.24: Iteration
- Exercise 1.25: Products
- Exercise 1.28: Perversity

Also do the following exercise:

For each of the following expressions, what must `f` be in order for the evaluation of the expression to not cause an error? For each expression, give a definition of `f` such that evaluating the expression will not cause an error, and say what the expression's value will be, given your definition.

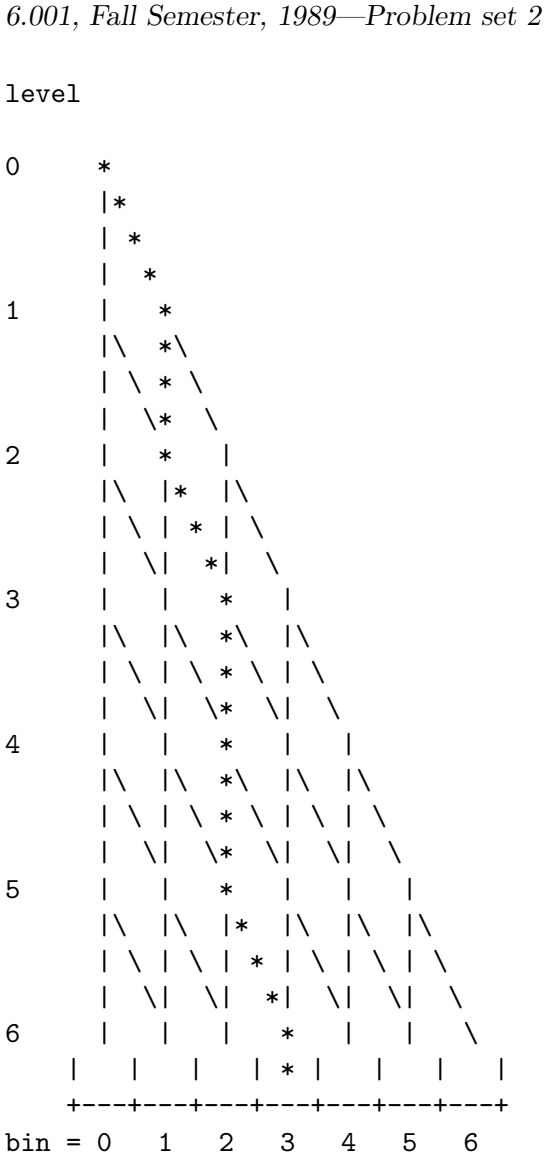
```
f
(f)
(f 3)
((f))
(((f)) 3)
```

Programming Assignment

Hand in your work on all labelled problems.

Pascal's Triangle

Imagine that we have a triangular maze, with paths connecting an entrance at the top to a set of bins at the bottom as in the diagram below. Eventually the path ends up in one of the bins at the bottom of the lattice. (Perhaps you have seen a demonstration of such a maze in an exhibit on probability.) We illustrate one typical path on the diagram by labelling it with "*".



There are two ways to get to level 1. There are four ways to get to level 2 (one way to get to level 2 bin 0, two ways to get to level 2 bin 1, and one way to get to level 2 bin 2).

Problem 1

In general, how many ways are there to get to the nth level of the maze? (This answer is easy. Work it out before even thinking about going to lab.)

It is a more complicated problem to determine how many different paths there are from the vertex to a particular bin at the bottom of the maze.

We observe that if we are at level zero we can get into the maze only at bin zero. If we are not at level zero there are only two ways to get to bin k at level n — we must come from either bin k of level $n-1$ or bin $k-1$ at level $n-1$. Thus the number of paths to bin k at level n must be the sum of the number of paths to the contributing bins. We can formalize this description as a Scheme program as follows:

```
(define (n-paths level bin)
  (if (= level 0)
      (if (= bin 0) 1 0)
      (+ (n-paths (- level 1) (- bin 1))
   (n-paths (- level 1) bin))))
```

Compute, by hand, the table of values of `(n-paths n k)` for $n, k = 0, 1, 2, 3, 4, 5$. This takes very little work (fewer than five minutes!) if you organize it carefully. Start with small values and you will soon see the pattern. Check your answer on a Chipmunk if you like.

You have probably noticed that the numbers you computed in the previous exercise are the binomial coefficients (Pascal's triangle). These numbers represent the number of combinations of n objects taken k at a time, ignoring order.

$$\texttt{(n-paths n k)} = C(n, k)$$

Our algorithm for computing these numbers is one of the worst that a sensible person might devise.

Problem 2

In general, how many calls to `n-paths` are required to compute the value of `(n-paths n k)` using the algorithm indicated above? The answer is a simple function of n (independent of k).

An alternative (and far better way) to compute binomial coefficients is to use the formula:

$$C(n, k) = \frac{n!}{k!(n-k)!}$$

Unfortunately, factorials become very large very fast, and arithmetic with large numbers is expensive. Also, computing $C(n, k)$ in this way requires a number of redundant multiplications (because some of the multiplications in the computation of n! are just to be undone by the division by either $k!$ or $(n - k)!$).

Problem 3

Write a short, elegant program for `(n-paths level bin)`, that avoids the redundant multiplications indicated above. Hand in this program, along with examples of its use, showing that it works for n up to 6. Give an estimate of the order of growth of time and space used by your program.

Smoothing

When random errors occur in measurements it is common to average data to reduce their effect. The following exercises are concerned with a form of averaging called smoothing. Consider measurements of the height of a ball thrown upwards with velocity V at time -T, with T chosen such that the ball reaches its maximum height (0) at time $t = 0$. The true position of the ball is $y(t) = -\frac{1}{2}gt^2$, the measured position $m(t) = y(t) + e(t)$, where $e(t)$ is the measurement error.

Problem 4

Construct a procedure `true-pos` with arguments V, g and t that computes the true height of the ball at time t.

To reduce the effect of measurement errors, we might attempt to smooth the data by averaging adjacent values: $S[m(t)] = \frac{1}{2}(m(t + h) + m(t - h))$, where h is a small time increment.

We can express the idea of smoothing as a procedure (analogous to the `deriv` procedure discussed on p. 68 of the text) as follows:

```
(define (smooth f h)
  (lambda (x)
    (average (f (+ x h))
     (f (- x h)))))
(define (average x y)
  (/ (+ x y) 2))
```

Problem 5

Even in the absence of random error ($e(t) = 0$) $S[m(t)]$ differs from the true position, $y(t)$. We call this difference the systematic error introduced by smoothing. Determine how the systematic error depends on h. You are to develop an expression for the systematic error in the measurement of $y(t)$ as a function of h.

Problem 6

Verify, by numerical experiment, that the systematic error introduced by applying `smooth` to `true-pos` has the dependence on h that you determined above.

When the random error term is non-zero, $S[m(t)]$ differs from $y(t)$ by the sum of the systematic error determined above and a term equal to $\frac{1}{2}(e(t-h)+e(t+h))$. In many cases this is smaller than $e(t)$. For example, when $e(t-h)$, $e(t)$, and $e(t+h)$ are roughly of the same size, $\frac{1}{2}(e(t-h)+e(t+h))$ is of the same size when $e(t-h)$ and $e(t+h)$ are of the same sign, but smaller when they differ in sign. In most practical situations, h is chosen by compromise since reducing the size of h decreases the systematic error, but also reduces the likelihood that the contributions of $e(t-h)$ and $e(t+h)$ to $S[m(t)]$ will cancel out.

When the random error of measurement is large it is often advantageous to repeat the smoothing operation a number of times, though this will increase the systematic error. So, for example, `(smooth (smooth m h) h)` may be less noisy than `(smooth m h)` which may be less noisy than m.

One useful abstraction for repeatedly applying a procedure is the `repeated` procedure, defined below:

```
(define (repeated f n)
  (if (= n 0)
      identity
      (compose f (repeated f (- n 1)))))
(define (identity x) x)
(define (compose f g)
  (lambda (x) (f (g x))))
```

For example, using this procedure, one can write:

```
==> ((repeated square 3) 5)
390625
```

Problem 7

Define and test a procedure, `nth-smooth`, of three arguments, `n`, the number of times to apply the smoothing procedure; `f`, the function to smooth; and `h`, the smoothing interval. `nth-smooth` must return the smoothed procedure. For example, `(nth-smooth 2 m h)` should return a procedure equivalent to `(smooth (smooth m h) h)`. You should be sure to work this out carefully before testing it on a Chipmunk. The required procedure is not long, but some thought is required to get it right.

`((smooth f h) t)` is equivalent to the algebraic expression $\frac{1}{2}(f(t-h)+f(t+h))$. Write algebraic expressions equivalent to `((smooth (smooth f h) h) t)` and to `((smooth (smooth (smooth f h) h) h) t)`.

You should see that the coefficients of `f` in these expressions are closely related to the numbers in Pascal's triangle, discussed above.

Problem 8

How many times must `f`, the procedure that implements the function being smoothed, be applied when evaluating `(nth-smooth n f h)`?

Problem 9

Write a different implementation of `nth-smooth` that applies the procedure that implements the function being smoothed only $n + 1$ times when evaluating `(nth-smooth n f h)`? Make use of your work from Problem 3.