

For the exercises below, you should turn in a transcript, with your name and login, demonstrating that your functions work, i.e. test them out on some inputs. Since the functional features of LISP make recursive techniques very natural, you should generally try to use recursion in your solutions. Avoid the use of `setq` where possible. Also, even though you will interact with these routines directly, it is not usually necessary to test a function's input parameters for errors. As always, try to use good programming style. Choose appropriate names for your procedures and variables and use comments where appropriate.

1. Write a recursive function `two-to-the` which takes a single non-negative integer parameter x and returns 2^x . Your procedure should take logarithmic time in x and should use `*` and *not* `expt` or `exp`.
2. Write a procedure (`numerical-derivative f`) which is given a function $f(x)$ and returns a function computing the numerical derivative $f'(x)$:

The *derivative* of a function $f(x)$ is nearly given by

$$f'(x) = \frac{f(x + \delta) - f(x)}{\delta}$$

where δ is very small. Write a procedure, `numerical-derivative` which takes as an argument a procedure computing a function f , returns a procedure computing the derivative of x . You should choose δ to be a small number (like .001). To compute the derivative of x^2 at $x = 5$, you should be able to type:

```
(defun square (x) (* x x))  
(funcall (numerical-derivative #'square) 5)
```

or even,

```
(setf (symbol-function 'square-prime) (numerical-derivative #'square))  
(square-prime 5)
```

Hint: You'll need `funcall`, `lambda`, and `function` or `#'`.

3. Write the following functions: `(last 1)` (returns the last element of l); `all-but-last 1`; `middle 1` (returns the middle element of list l , assuming an odd number of elements; do not use any numbers!)
Procedures `last` and `all-but-last` should take linear time. A definition of `middle` using `cdr` and `all-but-last` uses quadratic time and space; can you write a version that uses linear time and constant additional space?
4. A standard mathematical set function is *powerset* which computes the set of all subsets of a set. Write a `powererset` function. It may require other subsidiary functions.
5. (Symbolic Differentiation) One important AI application area is symbolic mathematics, particularly calculus. For this problem, you will construct a function `deriv` which differentiates simple, single-variable mathematical expressions. The `deriv` function takes two arguments. The first is a mathematical expression in standard Lisp syntax, containing numbers, atoms (representing constants and variables) and the functions `+`, `-`, `*`, `/`, `exp`, `expt`. The second is the name of the variable over which to differentiate. Other symbols in the expression are treated as constants. The rules of differentiation are as follows (where u and v are arbitrary expressions):

- $u = \text{constant} \Rightarrow du/dx = 0; dx/dx = 1$
- $d(u + v)/dx = du/dx + dv/dx; d(u - v)/dx = du/dx - dv/dx$
- $d(uv)/dx = u dv/dx + v du/dx$
- $d(u/v)/dx = (v du/dx - u dv/dx)/v^2$
- $v = \text{constant} \Rightarrow d(u^v)/dx = (du/dx)vu^{(v-1)}$
- $d(e^u)/dx = (du/dx)e^u$

Test your function on some interesting inputs.

6. (a) Write defstructs for points and lines (actually line segments) in two dimensions.
(b) Write a function (`distance p1 p2`) that returns the distance between two points.
(c) Write a function (`midpoint 1`) that returns the midpoint of a line segment.
(d) Write a function (`intersectp l1 l2`) that decides if two lines intersect. [Hint: the easiest way to do this is as follows. Suppose the position vectors of the endpoints of $l1$ and $l2$ are \mathbf{a} , \mathbf{b} and \mathbf{c} , \mathbf{d} respectively. A general point on $l1$ is $\alpha\mathbf{a} + (1 - \alpha)\mathbf{b}$; similarly, a general point on $l2$ is $\beta\mathbf{c} + (1 - \beta)\mathbf{d}$. The lines cross where

$$\alpha\mathbf{a} + (1 - \alpha)\mathbf{b} = \beta\mathbf{c} + (1 - \beta)\mathbf{d}$$

This gives you two equations (equating both x and y coefficients) for α and β . Solve these by hand, and use your program to calculate the values of α and β . The line segments actually intersect iff $0 \leq \alpha, \beta \leq 1$, unless they're parallel.

A sketch of a solution is shown below. You'll need to add the support procedures `p-cross`, `between` after you've worked out the algebra (about 7-10 lines of code total):

```
(defun dx (p q) (- (point-x p) (point-x q)))
(defun dy (p q) (- (point-y p) (point-y q)))
(defun intersectp (l1 l2)
  ;; l1 is line ab; l2 is line cd
  ;; assume the lines cross at alpha a + (1-alpha) b,
  ;;      also known as beta c + (1-beta) d
  ;; line segments intersect if 0<=alpha,beta<=1 unless they're parallel (q=0)
  (let* ((a (line-endpoint1 l1))
         (b (line-endpoint2 l1))
         (c (line-endpoint1 l2))
         (d (line-endpoint2 l2))
         (q (p-cross a b c d)))
    (if (not (zerop q))
        (let ((alpha (/ (p-cross d b c d) q))
              (beta  (/ (p-cross d b a b) q)))
          (and (>= alpha 0) (>= beta 0)
               (<= alpha 1) (<= beta 1)))
        (or (between c a b)
            (between d a b)))))
```

- (e) Write a `defstruct` for polygons (essentially a list of points) and scenes (essentially lists of polygons).
- (f) Write a function (`visiblep p1 p2 scene`) that checks if one point is visible from another in a scene (that is, there is a straight line from one to the other not intersecting any polygon).

7. A memoized version of a function, f , is one which first checks to see if f has ever been called with the same arguments before. If so, just return the same values as computed last time. Otherwise call f .

Write a lisp macro `defmfun` which behaves just like `defun` but instead uses memoization. For example, if you define

```
(defmfun f (x)
  (cond ((= x 0) 0)
        ((= x 1) 1)
        (t (+ (f (- x 1)) (f (- x 2))))))
```

then the call `(f 25)` should return 75025 in a fraction of a second.

Hint: You'll need to learn about macros, hash-tables and the following symbols: `#'` and `'` and `,` and `,@`. For *good* macros, also look up `multiple-value-bind`, `&body` and `gensym`.

8. Run the AIMA `test` routines to make sure it works and you know how to run it. To load the AIMA code, type
- ```
(load "/Net/solen/home/w/o/wolfe/public/385/aima/aima.lisp")
```

You'll want to look at the Overview of the AIMA code link in the course homepage to see how to test it. If you're not using the department machines, you can install a copy of your own; again, check out the course page.