Some Lisp Programming

Conditional Evaluation and Predicates

The basic Lisp conditional evaluation construct is `cond`.

```
> (defun my-abs (x)
    (cond ((> x 0) x)
          ((= x 0) 0)
          ((< x 0) (- x))))
MY-ABS
> (my-abs -2)
2
```

Several simplified versions exist, including `if`, `unless` and `when`

```
(defun my-abs (x) (if (>= x 0) x (- x)))
```

Another simplified version of `cond` is `case`:

```
> (defun f (i)
    (case i
      (1 'one)
      (2 'two)
      (t (error "out of range"))))
> (f 1)
ONE
```

Logical expressions can be combined using **`and`**, **`or`**, and **`not`**.

```
> (defun in-range (x) (and (< 3 x) (< x 5)))
IN-RANGE
> (in-range 2)
NIL
> (in-range 4)
T

> (defun not-in-range (x)
    (or (>= 3 x) (>= x 5)))
NOT-IN-RANGE
> (not-in-range 2)
T

> (defun in-range (x) (< 3 x 5))
IN-RANGE
> (in-range 2)
NIL
> (in-range 4)
T
> (defun not-in-range (x) (not (in-range x)))
NOT-IN-RANGE
```

More on Functions

Function as Data

Suppose we want a function `num-deriv` to compute a numerical derivative.

If we define

```
(defun f (x) (+ x (^ x 2)))
```

then we want to get

```
> (num-deriv #'f 1)
3
```

Defining `num-deriv` as

```
(defun num-deriv (fun x)
  (let ((h 0.00001))
    (/ (- (fun (+ x h))
          (fun (- x h)))
       (* 2 h))))
```

will not work – our function is the *value* of `fun`, not its *function definition*:

```
> (num-deriv #'f 1)
error: unbound function - FUN
```

We need a function that calls the value of `fun` with an argument:

```
> (funcall #'+ 1 2)
3
```

A correct definition of `num-deriv` is

```
(defun num-deriv (fun x)
  (let ((h 0.00001))
    (/ (- (funcall fun (+ x h))
          (funcall fun (- x h)))
       (* 2 h))))
```

Another useful function is `apply`:

```
> (apply #'+ '(1 2 3))
6
> (apply #'+ 1 2 '(3 4))
10
> (apply #'+ 1 '(2 3))
6
```

Anonymous Functions

Defining and naming throw-away functions like `f` is awkward.

The same problem exists in mathematics.

Logicians developed the *lambda calculus*:

$$\lambda(x)(x + x^2)$$

is "the function that returns $x + x^2$ for the argument x."

Lisp uses this idea:

```
(lambda (x) (+ x (^ x 2)))
```

is a *lambda expression* for our function.

Lambda expressions are not yet Lisp functions.

To make them into functions, you need to use **`function`** or **`#'`**:

```
#'(lambda (x) (+ x (^ x 2)))
```

To take our derivative:

```
> (num-deriv #'(lambda (x) (+ x (^ x 2)))
             1)
3
```

To plot $2x + x^2$ over the range $[-2, 4]$,

```
(plot-function #'(lambda (x)
                   (+ (* 2 x) (^ x 2)))
               -2
               3)
```

Functions can also use lambda expressions to make new functions and return them as the value of the function.

We will see a few examples of this a bit later.

Local Variables and Environments

Variables and Scoping

A pairing of a variable symbol with a value is called a *binding*

Collections of bindings are called an *environment*.

Bindings can be global or they can be local to a group of expressions.

`let` and `let*` expressions and function definitions set up local bindings.

Consider the lambda expression

```
(lambda (x) (+ x a))
```

The meaning of **`x`** in the body is clear – it is bound to the calling argument.

The meaning of **`a`** is not so clear – it is a *free variable.*

We need a convention for determining the bindings of free variables.

This is the reason we need to use `function` on lambda expressions:

> Free variables in a function are bound to their values in the environment where the function is created

This is called the *lexical* or *static* scoping rule.

Other scoping rules are possible.

An example: making a derivative function:

```
> (defun make-num-deriv (fun)
    (let ((h 0.00001))
      #'(lambda (x)
          (/ (- (funcall fun (+ x h))
                (funcall fun (- x h)))
             (* 2 h)))))
MAKE-NUM-DERIV
> (setf f (make-num-deriv
             #'(lambda (x) (+ x (^ x 2)))))
#<Closure: #120cbe80>
> (funcall f 1)
3
> (funcall f 3)
7
```

Another example: making a normal log likelihood:

The log likelihood of a sample from a normal distribution is

$$-\frac{n}{2}\left[\log \sigma^2 + \frac{(\overline{x}-\mu)^2}{\sigma^2} + \frac{s^2}{\sigma^2}\right]$$

A function for evaluating this expression as a function of μ and σ^2 is returned by

```
(defun make-norm-log-lik (x)
  (let ((n (length x))
        (x-bar (mean x))
        (s-2 (^ (standard-deviation x) 2)))
    #'(lambda (mu sigma-2)
      (* -0.5
         n
         (+ (log sigma-2)
            (/ (^ (- x-bar mu) 2) sigma-2)
            (/ s-2 sigma-2))))))
```

The result returned by this function can be maximized, or it can be plotted with `spin-function` or `contour-function`.

Local Functions

It is also possible to set up local functions using `flet`:

```
> (defun f (x)
    (flet ((add-1 (x) (+ x 1)))
      (add-1 x)))
F
> (f 2)
3
> (add-1 2)
error: unbound function - ADD-1
```

`flet` sets up bindings in parallel, like `let`

`flet` cannot be used to define local recursive functions.

`labels` is like `flet` but allows mutually recursive function definitions.

Optional, Keyword and Rest Arguments

A number of functions used so far take optional arguments, keyword arguments, or variable numbers of arguments.

A function taking an optional argument is defined one of three ways:

```
(defun f (x &optional y) ...)
(defun f (x &optional (y 1)) ...)
(defun f (x &optional (y 1 z)) ...)
```

In the second and third forms, the default value is 1; in the first form it is `nil`

In the third form, `z` is `t` if the optional argument is supplied; otherwise `z` is `nil`.

We can add an optional argument for the step size to `num-deriv`:

```
(defun num-deriv (fun x &optional (h 0.00001))
  (/ (- (funcall fun (+ x h))
        (funcall fun (- x h)))
     (* 2 h)))
```

Keyword arguments are defined similarly to optional arguments:

```
(defun f (x &key y) ...)
(defun f (x &key (y 1)) ...)
(defun f (x &key (y 1 z)) ...)
```

The function is then called as

```
(f 1 :y 2)
```

Using a keyword argument in **num-deriv**:

```
(defun num-deriv (fun x &key (h 0.00001))
  (/ (- (funcall fun (+ x h))
        (funcall fun (- x h)))
     (* 2 h)))
```

With a keyword argument, **num-deriv** is called as

```
(num-deriv #'f 1 :h 0.001)
```

A function with a variable number of arguments is defined as

```
(defun f (x &rest y) ...)
```

All arguments beyond the first are made into a list and bound to `y`.

For example:

```
> (defun my-plus (&rest x) (apply #'+ x))
MY-PLUS
> (my-plus 1 2 3)
6
```

If more than one of these modifications is used, they must appear in the order `&optional`, `&rest`, `&key`.

There is an upper limit on the number of arguments a function can receive.

Mapping

Mapping is the process of applying a function elementwise to a list.

The primary mapping function is **mapcar**:

```
> (setf x (normal-rand '(2 3 2)))
((0.27397 3.5358) (-0.11065 1.2178 1.050)
 (0.78268 0.95955))
> (mapcar #'mean x)
(1.904895 0.71913 0.8711149)
```

Mapcar can take several lists as arguments:

```
> (mapcar #'+ '(1 2 3) '(4 5 6))
(5 7 9)
```

Using **mapcar**, we can define a simple numerical integrator for functions on $[0, 1]$:

```
> (defun integrate (f &optional (n 100))
    (let* ((x (rseq 0 1 n))
           (fv (mapcar f x)))
      (mean fv)))
INTEGRATE
> (integrate #'(lambda (x) (^ x 2)))
0.335017
```

More on Compound Data

Lists

Lists are the most important compound data type.

Lists can be empty:

```
> (list)
NIL
> '()
NIL
> ()
NIL
```

They can be used to represent sets:

```
> (union '(1 2 3) '(3 4 5))
(5 4 1 2 3)
> (intersection '(1 2 3) '(3 4 5))
(3)
> (set-difference '(1 2 3) '(3 4 5))
(2 1)
```

In addition to using `select`, you can get pieces of a list with

```
> (first '(1 2 3))
1
> (second '(1 2 3))
2
> (rest '(1 2 3))
(2 3)
```

Two other useful functions are `remove-duplicates`

```
> (remove-duplicates '(1 1 2 3 3))
(1 2 3)
```

and `count`:

```
> (count 2 '(1 2 3 4) :test #'=)
1
> (count 2 '(1 2 3 4) :test #'<=)
3
```

`remove-duplicates` also accepts a `:test` argument.

Vectors

Vectors are a second form of compound data.

A vector is constructed with the `vector` function

```
> (vector 1 2 3)
#(1 2 3)
```

or by typing its printed representation:

```
> (setf x '#(1 2 3))
#(1 2 3)
> x
#(1 2 3)
```

Elements of vectors can be extracted and changed with `select`:

```
> (select x 1)
2
> (setf (select x 1) 5)
5
> x
#(1 5 3)
```

Vectors can be copied with `copy-vector`.

Vectors are usually stored more efficiently than lists, and their elements can be accessed more rapidly.

But there are fewer functions for operating on vectors than on lists:

```
> (first x)
error: bad argument type - #(1 5 3)
> (rest x)
error: bad argument type - #(1 5 3)
```

Sequences

Lists, vectors, and strings are sequences.

Several functions operate on any sequence:

```
> (length '(1 2 3))
3
> (length '#(1 2 3))
3
> (length "abc")
3
> (select '(1 2 3) 0)
1
> (select '#(1 2 3) 0)
1
> (select "abc" 0)
#\a
```

Sequences can be coerced to different types with `coerce`:

```
> (coerce '(1 2 3) 'vector)
#(1 2 3)
> (coerce "abc" 'list)
(#\a #\b #\c)
```

Arrays

The `matrix` function constructs a two-dimensional array:

```
> (matrix '(2 3) '(1 2 3 4 5 6))
#2A((1 2 3) (4 5 6))
```

Again you can type the printed representation

```
> (setf m '#2A((1 2 3) (4 5 6)))
#2A((1 2 3) (4 5 6))
```

and `select` extracts and modifies elements:

```
> (select m 1 1)
5
> (select m 1 '(0 1))
#2A((4 5))
> (select m '(0 1) '(0 1))
#2A((1 2) (4 5))
> (setf (select m 1 1) 'a)
A
> m
#2A((1 2 3) (4 A 6))
```

Format

`format` is a very flexible output function.

It prints to *output streams* or to strings.

The default output stream is `*standard-output*`; it can be abbreviated to `t`:

```
> (format *standard-output* "Hello~%")
Hello
NIL
> (format t "Hello~%")
Hello
NIL
> (format nil "Hello")
"Hello"
```

`~%` is the *format directive* for a new line.

Other useful format directives are `~a` and `~s`:

```
> (format t "Examples: ~a ~s~%" '(1 2) '(3 4))
Examples: (1 2) (3 4)
NIL
> (format t "Examples: ~a ~s~%" "ab" "cd")
Examples: ab "cd"
```

These two directives differ in their handling of *escape characters*.

There are many other format directives.

Some Statistical Functions

Some Basic Functions

```
> (difference '(1 3 6 10))
(2 3 4)
> (pmax '(1 2 3) 2)
(2 2 3)
> (split-list '(1 2 3 4 5 6) 3)
((1 2 3) (4 5 6))
> (cumsum '(1 2 3 4))
(1 3 6 10)
> (accumulate #'* '(1 2 3 4))
(1 2 6 24)
```

Sorting Functions

```
> (sort-data '(14 10 12 11))
(10 11 12 14)
> (rank '(14 10 12 11))
(3 0 2 1)
> (order '(14 10 12 11))
(1 3 2 0)
```

Interpolation and Smoothing

```
(spline x y :xvals xv)
(lowess x y)
(kernel-smooth x y :width w)
(kernel-dens x)
```

Linear Algebra Functions

```
(identity-matrix 4)
(diagonal '(1 2 3))
(diagonal '#2a((1 2)(3 4)))
(transpose '#2a((1 2)(3 4)))
(transpose '((1 2)(3 4)))
(matmult a b)
(make-rotation '(1 0 0) '(0 1 0) 0.05)

(lu-decomp a)
(inverse a)
(determinant a)
(chol-decomp a)
(qr-decomp a)
(sv-decomp a)
```

Odds and Ends

Errors

The **error** functions signals an error:

```
> (error "bad value")
error: bad value
> (error "bad value: ~s" "A")
error: bad value: "A"
```

Debugging

Several debugging functions are available:

debug/nodebug – toggle debug mode; in debug mode, an error puts you into a break loop.

break – called within a function to enter a break loop

baktrace – prints traceback in a beak loop.

step – single steps through an evaluation.

Example

Estimating a Survival Function

Suppose the variable **times** contains survival times and **status** contains status values, with 1 representing death and 0 censoring.

To compute a Kaplan-Meier or Fleming-Harrington estimator, we first need the death times and the unique death times:

```
(setf dt-list
      (coerce (select times
                      (which (= 1 status)))
              'list))
(setf udt
      (sort-data
       (remove-duplicates dt-list :test #'=)))
```

Next, we need the number of deaths and the number at risk at each death time:

```
(setf d (mapcar #'(lambda (x)
                    (count x dt-list :test #'=))
                udt))
(setf r (mapcar #'(lambda (x)
                    (count x times :test #'<=))
                udt))
```

Using these values, we can compute the Kaplan-Meier estimator at the death times as

```
(setf km (accumulate #'* (/ (- r d) r)))
```

The Fleming-Harrington estimator is

```
(setf fh (exp (- (cumsum (/ d r)))))
```

Greenwood's formula for the variance is

```
(* (^ km 2) (cumsum (/ d r (pmax (- r d) 1))))
```

The **pmax** expression prevents a division by zero.

Tsiatis' formula leads to

```
(* (^ km 2) (cumsum (/ d (^ r 2))))
```

To construct a plot we need a function that builds the consecutive corners of a step function:

```
(defun make-steps (x y)
  (let* ((n (length x))
         (i (iseq (+ (* 2 n) 1))))
    (list (append '(0) (repeat x (repeat 2 n)))
          (select (repeat (append '(1) y)
                          (repeat 2 (+ n 1)))
                  i))))
```

Then

```
(plot-lines (make-steps udt km))
```

produces a plot of the Kaplan-Meier estimator.

Weibull Regression

Suppose `times` are survival times, `status` contains death/censoring indicators, and `x` contains a matrix of covariates, including a column of ones.

A Weibull model for these data has a log likelihood of the form

$$\sum s_i \log \alpha + \sum (s_i \log \mu_i - \mu_i)$$

where

$$\begin{aligned} \log \mu_i &= \alpha \log t_i + \eta_i \\ \eta_i &= x_i \beta \end{aligned}$$

and α is the Weibull exponent, β is a vector of parameters.

A function to compute this log likelihood is

```
(defun llw (x y s log-a b)
  (let* ((a (exp log-a))
         (eta (matmult x b))
         (log-mu (+ (* a (log y)) eta)))
    (+ (* (sum s) log-a)
       (sum (- (* s log-mu) (exp log-mu))))))
```

Reasonable initial estimates for the parameters might be $\alpha = 1$,

$$\beta_0 = \frac{\Sigma s_i}{\Sigma t_i}$$

for the constant term, and $\beta_i = 0$ for all other i.

For a single covariate:

```
> (newtonmax
   #'(lambda (theta)
     (llw x
          times
          status
          (first theta)
          (rest theta)))
   (list 0 (/ (sum status) (sum times)) 0))
maximizing...
Iteration 0.
Criterion value = -510.668
...
Iteration 8.
Criterion value = -47.0641
Reason for termination: gradient size ...
(0.311709 -4.80158 1.73087)
```