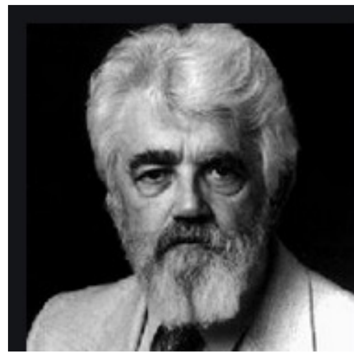
Lisp -- Quick Intro


John McCarthy

- Based on the λ-calculus;
- All data are lists (essentially)

e.g., are e^{ax}, $\mathrm{sqrt}(ax^2+by^2)$ functions??

NO, not until you've specified what values are fixed, and which ones are variable!!

A traditional solution is to invent a function name, e.g.,

$f(x) = e^{ax}$, for all real x [so, e and a are fixed constants]
$g(e,a) = e^{ax}$, for all real $e > 0$, and all real a [x fixed!]
$h(x,y) = ax^2+by^2$ for all real x, y [so a, b are fixed]
$k(a,b,x,y) = ax^2+by^2$ for all real a, b, x, y [nothing fixed]

The λ-calculus identifies variables without function-naming:

$(\lambda\ (x)\ e^{ax})$, $(\lambda\ (e,a)\ e^{ax})$, $(\lambda\ (a,b,x,y)\ (ax^2+by^2))$. ("λ-abstraction")

This makes it easy to "mix together" ordinary expressions (denoting numbers or lists) with expressions denoting functions, as arguments of other functions – very general!

Lisp versions:

```
(lambda (x) (exp (* a x))), (lambda (e a) (exp (* a x))),
(lambda (x y) (+ (* a x x) (*b y y))), ...
```

We can still give functions names in both the λ-calculus and in Lisp. E.g., in the λ-calculus we can define
$f =_{df} (\lambda\,(x)\; e^{ax})$, $h =_{df} (\lambda\,(x,y)\; (ax^2+by^2))$, etc.

Then note the equivalences (for fixed e, a, & any c)
$f(c) = (\lambda\,(x)\; e^{ax})(c) = e^{ac}$ ("λ-conversion")

And in Lisp (where 'exp' is built-in, $= (\lambda\,(x)\; e^x)$),

```
(defun f (x) (exp (* a x))),
(defun h (x y) (+ (* a x x) (* b y y))), etc.
```

Here we likewise have the equivalence
(f c) = (exp (* a x))) = (exp (* a c)) (given a, c).

But it's good to remember that you can also use lambda -defined functions directly in code, and apply them to arguments there.

Moving on to a simple example ("hello", simply or cleverly):

```
sbcl
* 'hello
HELLO
* "hello"
"hello"
* (defparameter *name* nil)
* (defun hello ()
    (format t "Hello, what's your first name?~%")
    (setq *name* (read))
    (format t "Nice to meet you, ~a~%" *name*))
HELLO
* (hello)
Hello, what's your first name?
Len
Nice to meet you, LEN
NIL   ; use (exit) to escape
```

- **Loading a file:** e.g.,
  (load "myfile.lisp")
- **Data types:**
  - atoms (e.g., 3, -2, 2/3, 3.14, T, NIL, ( ), :A, :B3, A, CSC191, \;, JOE, |Joe|, @U.ROCHESTER.EDU, #\a, #\A, #\Space, #\;, ...), "a four-word string"
  - lists: NIL, (), (THIS & (THAT (MAKES 6 "words") ?))
  - arrays: (setq A (make-array '(3 4))
  - hash tables: (setq *KB* (make-hash-table :test 'equal))
  - structures:
    (defstruct coursename time credits room instructor)

- **Functions** – use
  (defun <name> (let (x y z) <body>)) for local variables;
  more examples to come

- **Basic built-in functions**
  (setq x '(A B C))) ; 'setf' is more general
  (car x) ➔ A
  (cdr x) ➔ (B C) ; the rest of the list, w/o 1st element
  (cadr x) ➔ B ; also called (second x)
  (setq x (cons '(D E) x) ➔ ((D E) A B C); "insert 1st item"

- local & global variables
- loops: e.g., (dolist (x mylist) …),
  (dotimes (i 15) …)

Function examples

```
(defun hypotenuse (base height)
      (sqrt (+ (expt base 2) (expt height 2))))
(hypotenuse 3 4) ==> 5.0

(defun atoms-of (lst)
   (cond ((null lst) nil)
            ((atom lst) (list lst))
            (t (append (atoms-of (car lst)) (atoms-of (cdr lst))))))

(atoms-of 'a) ==> (A)
(atoms-of '(a b)) ==> (A B)
(atoms-of '((a b) (c (d)))) ==> (A B C D)
(atoms-of '((a "this thing") (#\; (|That Thing|))))
                        ==> (A "this thing" #\; |That Thing|)
```

Debugging

- I like the following when there's a crash:
  backtrace (:dn in acl) ... unwinds the recent function calls
  list-locals (:loc in acl) ... prints out the local variables

- use the 'trace' function (give names of functions, without quotes, as arguments of 'trace')

- locally rename functions, to be able to trace those occurrences; e.g., suppose you want to see the argument of 'null', and of the first recursive occurrence of 'atoms-of' above; you could write

```
  (defun atoms-of (lst)
    (cond ((null1 lst) nil)
          ((atom lst) (list lst))
          (t (append (atoms-of1 (car lst)) (atoms-of (cdr lst))))))

  (defun null1 (x) (null x))
  (defun atoms-of1 (x) (atoms-of x))
  (trace null1 atoms-of1)
```

```
(atoms-of '((a b) (c (d)))) ==>
 0[2]: (NULL1 ((A B) (C (D))))
 0[2]: returned NIL
 0[2]: (ATOMS-OF1 (A B))
   1[2]: (NULL1 (A B))
   1[2]: returned NIL
   1[2]: (ATOMS-OF1 A)
     2[2]: (NULL1 A)
   etc.
```

One more example – simple 1-level pattern matching

The following will match a list containing constants and variable to a list of constants, returning the latter if the match succeeds; e.g., (match1 ‘(Owns ?x ?y) ‘(Owns Alice Snoopy)) ➔ (OWNS ALICE SNOOPY)

```
(defun match1 (patt wff)
  ; patt: a list of atoms
  ; wff: a list of atoms
  ; RESULT: If patt matches wff, return wff, o/w return NIL.
    (let ((w wff) x y)
         (when (or (atom patt) (atom wff))
            (format t "~%**ERROR: 'match' wants 2 lists of atoms as ~
                 arguments,~%  got ~a and ~a" patt wff)
            (return-from match nil))
         (loop (setq x (pop patt) y (pop w))
               (if (and (not (var x)) (not (eq x y))); mismatch?
                         (return-from match nil))
               (if (and (null patt) (null w)); for a match, return wff
                         (return-from match wff)))
  )); end of match1
```