Day 4: Lisp III

Saturday, February 27, 2021 6:50 PM

- To pass a function to another function, use #'function (aka. 'functionquote')
- Lips equality cheatsheet:
  - = for numbers
    - Properly compares ints and floats
  - eq for symbols and integers
  - eql for strings
  - equal for deep comparisons of lists
- In lisp, data is code
  - (eval '(+ 3 4 5)) takes the data "(+ 3 4 5)" and executes it
  - DEFUN takes data, converts it to a program, and binds a symbol to it
    - It does this by using LAMBDA then SETVAR
- LAMBDA turns data into a program and returns the program: "(lambda (a b c) (+ a (* b c)))"
  - To pass it in elsewhere, still use functionquote: "#'(lambda …)"
  - It can also be passed without the functionquote?: "((lambda …) arg)"
- Temporary variable binding using let: (let ((var1 val1) (var2 val2) (var3 val3)) (expressions using var1, var2, var3))
  - Uses lambda on the inside: ((lambda (var1 var2 var3) (expressions)) val1 val2 val3)
  - NB: val* cannot reference any other var*. If you want to be able to, use let*:
    - (setf x 2) (let ((x 3) (y (* x 2))) (+ y 1)) => 5
    - (setf x 2) (let* ((x 3) (y (* x 2))) (+ y 1)) => 7
    - Uses nested LAMBDAs
- DEFMACRO: macros that are expanded at compile/load time and executed at execution time:
  - (defmacro nil! (var) (list 'setf var 'nil)) then (nil! myvar) does what you'd expect.
    - Note the weird construction of the list so that you can evaluate var but not setf or nil
  - To make that more concise, use backquote (like quote, but "," evaluates an argument):
    - (defmacro nil! (var) `(setf ,var nil))
    - ,@ evaluates the item after it and splices it into the list
  - Ex: to redefine let: (defmacro let2 (vars &rest exprs) `((lambda ,(mapcar #'car vars) , @exprs) ,@(mapcar #'cadr vars)))
  - MACROEXPAND expands a macro but doesn't execute it
- Associated lists (alists): key/value mappings as a lit of a CONS cells: (setvar *ages* '((mary . 23) (john . 22) (tim . 50))
  - (ASSOC KEY ALIST) gets the relevant pair (ex: (ASSOC 'john *AGES*) => (john . 22))
  - Slows down in linear time, as you'd expect, but low overhead
- Hashtables:
  - (defvar *table* (make-hash-table))
    - *NB:* takes various keyword arguments, including ":test" which defaults to "eql". Set it to "equal" if you're using lists as keys.
  - (setf (gethash key *table*) value)
  - (gethash key *table*)
  - He claims that hastables scale in constant time, which seems wrong to me. But better than alists.
  - MAPHASH iterates over every entry in a hashtable
  - REMHASH deletes an item from a hashtable
- Memoiaztion:
  - Ex: (defvar *fibhash* (make-hash-table)) (defun memfib (n)
    (or (gethash n *fibash*) (setf (gethash n *fibhash*) (if (< n 2) 1 (+ (fib (- n 1))) (fib (- n 2))))))
  - There's a general purpose utility version.