# Lisp Tutorial in Examples

Myroslava Dzikovska

CS244/444
Fall 2002

## 1 Working with lists

See Steele "Common Lisp The Language" chapter 15 (sequences) and 16(lists) for all available predicates.

```
(cons 'a 'b)            => (a . b)
(cons 'a '(b))          => (a b)
(cons '(a) 'b)          => ((a) . b)
(cons '(a) '(b))        => ((a) b)
(cons '(a b) '(c d e)) => ((a b) c d e)
(cons 'a nil)           => (a)
(cons '(a b) nil)       => ((a b))
(cons nil 'a)           => (nil . a)
(cons nil '(a b))       => (nil a b)

(list 'a 'b)            => (a b)
(list 'a '(b))          => (a (b))
(list '(a) 'b)          => ((a) b)
(list '(a) '(b))        => ((a) (b))
(list '(a b) '(c d e)) => ((a b) (c d e))
(list 'a nil)           => (a nil)
(list '(a b) nil)       => ((a b) nil)
(list nil 'a)           => (nil  a)
(list nil '(a b))       => (nil (a b))

(append 'a 'b)            => Error
(append 'a '(b))          => Error
(append '(a) 'b)          => (a . b)
(append '(a) '(b))        => (a b)
(append '(a b) '(c d e)) => (a b c d e)
(append 'a nil)           => Error
(append '(a b) nil)       => (a b)
(append nil 'a)           => a
(append nil '(a b))       => (a b)

(setq x '(c d e))    x => (c d e)
(push 'a x)          x => (a b c d e)
(pop x)              x => (b c d e)

(setq x '(a b c))    x => (a b c)
(first x)              => a
(rest x)               => (b c)
(second x)             => b
(nth 0 x)              => a
(nth 1 x)              => b
```

Style note: `(car x)` == `(first x)`, `(cdr x)` == `(rest x)`. It makes for more readable code if you avoid using `car` and `cdr`, and use `first` and `rest` instead.

## 2 Equality Tests

Summary: `eq` is true on identical pointers only (implementation dependent whether same numbers or chars are same pointers). `eql` is true on identical pointers and identical numbers and chars. `equal` is true on objects with isomorphic structure. Normally you want to use `equal` for your comparisons, unless you really care about pointers or dealing with circular lists. `equal` can enter an infinite loop on circular lists.

```
(eq 'a 'a)                        => T
(eql 'a 'a)                       => T
(equal 'a 'a)                     => T

(eq 1 1)                          => implementation dependent
(eql 1 1)                         => T
(equal 1 1)                       => T

(eq #\A #\A)                      => implementation dependent
(eql #\A #\A)                     => T
(equal #\A #\A)                   => T

(eq '(a) '(a))                    => implementation dependent
(eql '(a) '(a))                   => implementation dependent
(equal '(a) '(a))                 => T

(eq (cons 'a 'b) (cons 'a 'b))    => NIL
(eql (cons 'a 'b) (cons 'a 'b))   => NIL
(equal (cons 'a 'b) (cons 'a 'b)) => T

(eq (list 'a 'b) (list 'a 'b))    => NIL
(eql (list 'a 'b) (list 'a 'b))   => NIL
(equal (list 'a 'b) (list 'a 'b)) => T

(setq x (cons 'a 'b))
(eq x x)                          => T
(eql x x)                         => T
(equal x x)                       => T

(setq x (cons 'a 'b))
(setq y x)
(eq y x)                          => T
(eql y x)                         => T
(equal y x)                       => T
```

# 3 Functions and variables

## 3.1 Defining a function

Functions are defined using `defun` form.

```
(defun <name> (<list of parameters>)
 <"optional documentation string">
 <body>
)
```

Reverse the list

```
(defun reverse-recursive (l)
  "Reverse a list using recursion"
  (if (null l)
      nil
    (append (reverse-recursive (rest l)) (list (first l)))))
```

## 3.2 Global and local variables

All undeclared lisp variables are variables are **global**, a very undesirable thing. `let` can be used to declare the variables local

```
(let ((var1 val1)
      (var2 val2)
     ............
      (varn valn))
 (
  <body>
 ))
```

The following function reverses the list using iteration

```
(defun reverse-iterative (l)
  "reverse a list using iteration"
  (let ((result nil))
    (dolist (el l result)
      (push el result))))
```

Global variables are declared using `defvar` or `defparameter`. Use `defparameter` under normal circumstances. `defvar` is evaluated only once per lisp session, and the value of the variable won't change if you re-load the file, which is undesirable if you're relying on init values of the variables to be re-set every time you reload during debugging. Constants are declared using `defconstant`. The stylistic convention is to surround global variable names with asterisks, and global constant names with pluses.

```
(defparameter *error-number* nil
  "if non-nil, contains the error number for the error that occured ")
(defconstant +lisp-extension+ ".lisp"
  "the extension of files containing lisp code")
```

Note the difference between defvar and defparameter

```
File: testdefvar.lisp
(defvar *x* 'a "a variable")
----------------------------
:ld testdefvar     x=>a
(setq x b)         x=>b
:ld testdefvar     x=>b
```

```
File: testdefparameter.lisp
(defparameter x 'a "a variable")
---------------------------
:ld testdefparameter    x=>a
(setq x b)              x=>b
:ld testdefparameter    x=>a
```

## 3.3 Function Parameters

You probably won't use these, but need to know them to understand the notation in the LISP books.
&*rest* means that the following parameter stands for an arbitrary number or arguments. For example,

```
funcall fn &rest args
```

means that the function `funcall` takes a required argument *fn* and an arbitrary number of arguments following it. You can call it as

```
(funcall #'cons 'a '(b c)) => (a b c)
(funcall #'list 'a 'b 'c)  => (a b c)
```

&*key* is used to denote optional named parameters. For example,

```
position item sequence &key :from-end :test :test-not :start :end :key
```

means that `find` function takes an item, a sequence, and may have 6 different optional arguments. The optional arguments are denoted with *:arg* on the call. For example,

```
(position 'a '(a b c d a) :from-end t)   => 4
```

# 4 Flow Control

Conditionals are best done using `cond` predicate. Below is a definition of a sign function

```
(defun sign (x)
  "Sign function: return -1 if x is negative, 1 if positive, and 0 otherwise"
  (cond
   ((< x 0) -1)
   ((eql x 0) 0)
   ((> x 0) 1)))
```

The conditions are evaluated sequentially. The default condition is "T".

```
(defun test24 (x)
  "Returns 4 if x is divisible by 4, 2 if x is divisible by 2 but not 4, and 1 otherwise"
  (cond
   ((eql (mod x 4) 0) 4)
   ((eql (mod x 2) 0) 2)
   (t 1)
   ))
```

Iteration over a list can be done using `dolist`. Given a list of numbers, this will count all divisible by 4, all divisible by 2, and the rest

```
(defun count24 (x)
  "Return a list with 3 elemenst:
    a number of elements in x divisible by 4,
    a number of elements divisible by 2,
    and a number of elements not divisible by 2"
  (let ((div4 0) (div2 0) (other 0))
    (dolist (el x)
      (cond
       ((eql (mod el 4) 0)
        (setq div4 (+ 1 div4))
        (setq div2 (+ 1 div2)))
       ((eql (mod el 2) 0)
        (setq div2 (+ 1 div2)))
       (t (setq other (+ 1 other))))
      )
    (list div4 div2 other)
    ))
```

Note that a variable `el` defined by `dolist` is automatically considered local.
If there's only 1 condition, use `when`. Count all positive numbers in the list

```
(defun countpositive (x)
  "Return the number of positive elements in x"
  (let ((count 0))
    (dolist (el x)
      (when (>= el 0) (setq count (+ count 1))))
    count))
```

# 5 Transforming lists

For all the examples, we use the following definition, a list of office numbers, with some people sharing offices

```
(setq x '((Mary 221) (John 314) (Jane 221)))
```

## 5.1 Applying a function to all elements

Given the office list, get the list of people who have offices.
Since the person is the first element of every element in the office list, we need to apply the function `first` to each office list element in turn, and return the result.

```
(defun get-names (phonelist)
  "Returns the list of people's names in the given phone list"
  (mapcar #'first phonelist))

(get-names x) => (MARY JOHN JANE)
```

Given the office list, get the list of floors people are on, when floors denoted by the first digit of the office number.
Similarly, we apply a function `getFloor` which we will define to all elements of the office list to get our floor list.

```
(defun get-floor (entry)
 "Given an entry with the name and office number, returns the name
   and the floor where the office is"
  (list (first entry) (floor (second entry) 100))
)

(defun get-floors1 (phonelist)
  "Find out the floors on which people are located"
  (mapcar #'get-floor phonelist))

(get-floors1 x)  => ((Mary 2) (John 314) (Jane 2))
```

But we don't need two separate functions for such a simple task. Instead of defining a separate getFloor function, we can "inline" it by using a generic lambda function.

```
(defun get-floors2 (phonelist)
  "Find out the floors on which people are located"
  (mapcar (lambda (entry)
            (list (first entry) (floor (second entry) 100)))
          phonelist))

(get-floors2 x) => ((Mary 2) (John 3) (Jane 2))
```

Given the office list, write a generic function that produces phone numbers, assuming that the number is formed by adding a specified extension to the office number. Again, we can inline the function, and it can access the variable `extension` from the enclosing block.

```
(defun make-phone-list (officelist extension)
  "Given an office list, creates a phone list by adding a specified
  extension to the office numbers"
  (mapcar (lambda (entry)
            (list (first entry) (+ (second entry) extension)))
          officelist))

(make-phone-list x 275000) => ((Mary 275221) (John 275314) (Jane 275221))
```

Given the office list, write a function that finds all floors on which people reside. We have a pattern for it in our `get-floors2` function, but it also returns people's names. We need to get the second elements of the list returned by `get-floors2` to get the floor numbers only.

```
(defun get-floors-a (phonelist)
  (mapcar #'second (get-floors2 phonelist)))
```

```
(get-floors-a x) => (2 3 2)
```

That's almost right, but there are duplicate numbers in the list. Function `remove-duplicates` removes them

```
(defun get-floors (phonelist)
  (remove-duplicates
   (mapcar #'second (get-floors2 phonelist))))
```

```
(get-floors x) => (3 2)
```

Note that if 2 elements are equal, the earlier one in the sequence will be discarded by `remove-duplicates`. If you want the last one to be discarded, add *:from-end t* as the optional argument to the call.

## 5.2 Searching and selecting elements and sublists

Given the office list, find the name of some person who is in 221

```
(find 221 x :key #'second) => (Mary 221)
```

-OR-

```
(find-if (lambda (entry) (eql (second entry) 221)) x) => (Mary 221)
```

Find all people who are in 221. To do that, we want to remove the information about all people who are NOT in 221

```
(remove-if-not (lambda (entry) (eql (second entry) 221)) x)
```

-OR-

```
(remove-if (lambda (entry) (not (eql (second entry) 221)) x))
```

This is not quite enough. Here's what we get:

```
(remove-if-not (lambda (entry) (eql (second entry) 221)) x)
 => ((Mary 221) (Jane 221))
```

If we want to get the list of people names in 221 only, we have to get the list of first elements of phone list entries we found, as we did in the first example of "applying a function ... " section.

```
(mapcar #'first
  (remove-if-not (lambda (entry) (eql (second entry) 221)) x))
 => (Mary Jane)
```

Test functions are important in searches!!! The default test is `eq`, causing a common pitfall

```
(position '(1 2) '((3 4) (1 2) (2 5)))
 => nil
```

The problem is that find function compares elements using `eq`, and lists have to be compared using `equal`

```
 (position '(1 2) '((3 4) (1 2) (2 5)) :test #'equal)
 => 1
```

Note that `position` counts from 0, hence the result is 1 for the second element of the list

# 6 Examples of Useful data structures

## 6.1 Association lists (a-lists)

Lists composed of key-value pairs. Easy to search and transform.

```
(setf  age-list '((Mary . 22) (John . 19) (Jane . 30)))
(acons 'Jill 25 age-list)
    => ((Jill . 25) (Mary . 22) (John . 19) (Jane . 30))
(assoc 'John age-list)
    => (John . 19)
(setf birth-year-list
     (mapcar (lambda (x) (cons (car x) (- 1999 (cdr x))))))
    => ((Jill . 1974) (Mary . 1977) (John . 1980) (Jane . 1969))
```

## 6.2 Structures

```
(defstruct person name age sex)
(setf p1 (make-person :name Mary :age 22 :sex f))
(person-name p1) => Mary
(setf p2 (make-person))
(setf (person-name p2) John)
(setf (person-age p2) 19)
(setf (person-ses p2) f)
```

Note that you must use `setf` and not `setq` as the assignment operator to work with structure fields.

## 6.3 Hash Tables

```
(setf phone-book (make-hash-table))
(setf (gethash 'John phone-book) "999-999")
(setf (gethash 'Jane phone-book) "123-456")
(gethash 'Mary phone-book) => nil
(gethash 'John phone-book) => "999-999"
```