

# Lisp Concepts for Homeworks #1 and #2

## CMSC671 – Fall 2003 – Prof. desJardins

### 1 Basic concepts

<http://argus.irb.hr/cd/b/lisp/common-lisp-tutorial.html> covers the fundamentals of Lisp. If you're having trouble with the basics, you should first work your way through the examples in this tutorial. There are also a couple of other tutorials posted on the website (on the syllabus page).

### 2 Key functions for HW1

Here are the functions that I used for my own solutions to the Lisp programming part of Homework #1. This does not mean that you must use exactly these functions; there are many other correct ways to implement these functions.

```
>
car
cdr
cond
defun
error
if
lambda
list
list
listp
loop
lower-case-p
mapcan
nconc
not
null
numberp
progn
remove-if
return-from
setf
stringp
upper-case-p
```

### 3 Map constructs and lambda functions

`mapcar` and related constructs are powerful built-in functions for operating on lists. When combined with the notion of a `lambda` (anonymous) function, you can do amazingly complicated things. (A “lambda function” looks just like a regular function, but instead of using `defun`

`function-name (args) body`) the syntax `#'(lambda (args) body)` represents a function with that definition.) For example, if I have a list of numbers and want to compute the square of every element of the list:

```
> (mapcar #'(lambda(e1) (* e1 e1)) '(1 2 3))
(1 4 9)
```

I could also do this by defining a `square` function and then using that in the `mapcar`:

```
> (defun square(n) (* n n))
SQUARE
> (mapcar #'square '(1 2 3))
(1 4 9)
```

Notice that the syntax `#'` is used to quote the name of the function to be used in the `mapcar`. This is called *read-macro* syntax. `#'function-name` is just shorthand for `(function function-name)`, which retrieves the function definition for `function-name`.

`mapcan` uses `nconc` (which is the destructive “append” operation) to append together all of the results. I can use this to generate a list that contains every number in the original list, its square, and its cube:

```
> (mapcan #'(lambda(e1) (list e1 (* e1 e1) (* e1 e1 e1))) '(1 2 3))
(1 1 1 2 4 8 3 9 27)
```

A useful function that is often used with `mapcar` is `remove-if`, which you can use to remove elements of a list that satisfy a given property. For example, this set-difference function returns the elements of the first list that don’t appear in the second list:

```
(defun set-diff (l1 l2)
  (remove-if #'null (mapcar #'(lambda (e1) (if (member e1 l2) nil e1))
                    l1)))
```

(Note that this function doesn’t do any error checking, and doesn’t remove duplicates.)

Here’s another version of `set-diff` that’s a bit more elegant, skipping `mapcar` altogether and just using `remove-if` directly on the first list.

```
(defun set-diff (l1 l2)
  (remove-if #'(lambda (e1) (member e1 l2)) l1))
```

## 4 Dotted pairs

A dotted pair is simply a list in which the `car` and `cdr` are both atoms:

```
> (cons 'a 'b)
(A . B)
```

Association lists (see below) often contain dotted pairs.

## 5 Association lists

An association list is just a list of lists. Usually (but not always), each sublist is a dotted pair. The built-in function `assoc` looks for a sublist whose `car` matches a specified element, and returns that sublist. If the element isn’t found, `assoc` returns `nil`. For example:

```
> (assoc 'a '((a 1) (b 2) (c 3)))
(A 1)
> (assoc 'x '((a 1) (b 2) (c 3)))
NIL
```

If the association list contains dotted pairs, then `(cdr (assoc KEY ASSOC-LIST))` will return the `cdr` of the retrieved sublist (i.e., the element associated with the given key). Try it!

## 6 Defining variables and constants

Lisp programmers typically use asterisks to indicate a global variable. `defvar` is used at the top level (i.e., not inside a function) to declare a global variable. For example,

```
(defvar *global-variable* 27
  "This is a global variable definition")
```

Within a function, if you are planning to use a global variable, you should declare it to be special:

```
(defun foo (bar)
  (declare (special *global-variable*))
  ...)
```

You can actually create a dynamically scoped variable *anywhere* by using a special declaration.

Local (lexically scoped) variables can be defined using `&aux` in a function specification (and optionally specifying a value for the variable as in the example below), or using the `let` construct anywhere within a function. See below.

You can, in fact, just create a (global) variable by starting to use it, but that's not considered to be good programming practice.

Constants are defined using `defconstant`, are often denoted by names in all-capitals:

```
(defconstant INFINITY 100000000000
  "Not really infinity, but a very very very big number")
```

Please remember that it's good programming style to use meaningful variable names (i.e., ones that allow the casual reader of your code to immediately understand the purpose of a variable).

## 7 Functions and local variables

As you know, `defun` is the special form for defining a new function. In addition to the standard argument list, Lisp also permits `optional`, `rest`, `keyword`, and `local` arguments. Here's an example of a function definition that uses them all:

```
(defun new (x y &optional (z 0)
  &rest rest
  &key flag1 flag2
  &aux (a 'a) b c)
  ...)
```

`x` and `y` are standard arguments, and must be the first two values passed in. The next argument, `z`, is optional. If a third argument is given, `z` will be bound to that argument; otherwise, `z` will be bound to its default value, 0. (If no default value is specified in the declaration, and no value is passed in, the optional argument will take the value `nil`.) If there are four or more arguments, `rest` will be bound to a list containing each of the remaining arguments. `&aux` specifies local variables, optionally with initial values (as with `a` here).

Since keyword arguments were specified, these remaining arguments must also be keyword-value pairs. A *keyword* in Lisp is simply a symbol that starts with a colon (e.g., `:key`). Therefore, the following invocation is legal:

```
(new 1 2 3 :flag1 t :flag2 t)
```

and results in the following bindings:

```
x    = 1
y    = 2
z    = 3
```

```

rest = '(:flag1 t :flag2 t)
flag1 = T
flag2 = T
a = 'a
b = NIL
c = NIL

```

Lisp's argument types make it easy to invoke functions in flexible ways, and make it easy to extend programs by adding new optional or keyword arguments (without having to rewrite all of the existing function calls). In general, I don't recommend mixing this many argument types in one function specification (at least not until you're more familiar with the syntax), but you may find some of the types to be useful for upcoming homeworks.

See Graham Section 6.3 and Steele Chapter 5 for more information (the various types of argument options for functions are described in Steele Section 5.2.2).

Local variables are bound using the `let` and `let*` forms.

```

(let ((var1 value1) (var2 value2) var3)
  expr1
  expr2
  ...)

```

The variables `var1` and `var2` are set to `value1` and `value2` respectively, and `var3` is created and bound to `nil`. Then the expressions `expr1`, `expr2`, etc. are evaluated. When the `let` ends, the local variables no longer exist.

With `let*`, all variable bindings are performed in parallel. If you wish to create a series of variables whose bindings may depend on the previous local variable's binding, use `let`, which binds variables sequentially.

## 8 File I/O

`read` is the main input routine in Lisp. `read` parses a Lisp expression (but doesn't evaluate it).

```
(read)
```

by itself just reads one expression from the input stream. `read` also takes several optional arguments: the stream from which to read, an `eof-error-p` flag that tells `read` whether to signal an error at the end of a file, and an `eof-value` argument that tells `read` what to return at the end of the file if `eof-error-p` is `nil`.

The stream should be an open input stream. You can open a file as an input stream by using `open`, but it's better to use `with-open-file`:

```

(with-open-file (str "filename" :direction :input)
  (let ((temp (read str nil 'eof)))
    (if (eq temp 'eof)
        (format t "Read end of file~%")
        (format t "INPUT: ~s~%")))))

```

opens a file named `filename` and creates an input stream named `str` to read from this file. The invocation of `read` parses a Lisp expression from the stream (i.e., the first expression that appears in `filename`) and places it in the variable `temp`. If the end of file is reached, `read` returns the symbol `'eof`.

## 9 Loops

Common Lisp's `loop` facility is a powerful iteration tool that you can use to create just about any kind of iterative construct you can imagine.

In its simplest form, with no keywords, `loop` just loops forever over the body:

```
(loop (format t "hello~%"))
```

just prints “hello” until you kill it or your computer crashes.

In its more general form, `loop` has three parts: the *prologue*, which contains initialization forms and possibly a termination test, the *body*, which contains expressions to be executed each time through the loop, and the *epilogue*, which contains code that is executed after the loop ends.

A loop consists of a series of clauses, containing keywords that are parsed by Lisp to construct the loop. In the most general case, you can have lots of different clauses, but in practice, most loops have either one initialization/termination clause (which I’ll call “init”) or an init clause plus a termination (“end”) clause, and one execution (“body”) clause. Here’s an example of a loop with a `for` init and a `do` body:

```
(loop for i from 1 to 10
      do (format t "~s squared is ~s%" i (* i i))
      )
```

`for`, `from`, `to`, and `do` are all loop keywords. `i` is a local variable created by the `for` init clause. 1 and 10 could be replaced by any expressions that evaluate to numbers. The `format` expression could be replaced by any other expression to be executed, including a `progn` that would allow you to specify a series of expressions to be evaluated.

The `do` version of the loop returns `nil`, unless you use `(return EXPR)` within the loop to return a result (which ends the iteration, even if the termination condition hasn’t been reached).

Some other init clauses include:

```
for VAR in LIST      ;; LIST is any expression that evaluates
                    ;; to a list
for VAR across ARRAY ;; iterates through values of an array

for VAR being each hash-key of HASH-TABLE
for VAR being each hash-value of HASH-TABLE
    ;; iterate through keys or values of a hash table
    ;; very useful for HW #2!
repeat N            ;; iterate N times

for VAR = EXPR1 then EXPR2 ;; VAR is initialized to
                        ;; EXPR1, then to EXPR2 on subsequent
                        ;; iterations
```

The last of these init clauses will loop forever, unless you use `return` or specify a termination clause, such as:

```
while EXPR      ;; keep going as long as EXPR is true
until EXPR      ;; keep going until EXPR is false
```

These termination clauses can appear anywhere in the loop (i.e., before *or* after the body), and are evaluated at that point in the iteration.

The `do` body just executes expressions; it doesn’t return any values. There are many body clauses that tell the `loop` macro to accumulate and return values, including:

```
collect EXPR ;; collect the values returned by EXPR in a list
nconc EXPR   ;; nconc the lists returned by EXPR
append EXPR  ;; append the lists returned by EXPR
count EXPR   ;; count the number of times EXPR is evaluated
sum EXPR     ;; return the sum of each invocation of EXPR
maximize EXPR ;; return the maximum value of EXPR
minimize EXPR ;; return the minimum value of EXPR
```

To give one more example,

```
(loop for x in '(a 1) (b 2) (c 3) (d 4))
  sum (second x))
```

returns the value 10.

Graham Section 14.5 goes into more detail on loops. You can also refer to Chapter 26 of the Common Lisp reference manual for more than you ever wanted to know about loops, including step increments, looping downwards, executing multiple bodies and combining the results with *finally*, and much, much more.

## 10 Formatting Lisp code in emacs

If you don't normally use emacs, you may want to start. Emacs has a built-in Lisp formatting mode that is very useful in getting your parentheses lined up, which in turn is helpful in writing readable and bug-free code. (Vi also has some Lisp formatting help, but not as much as emacs.)

Emacs is available on the department Unix and Linux machines, and you can download xemacs for Windows from <http://www.xemacs.org/> .

Normally, if you edit a file ending in ".lisp," emacs will automatically enter Lisp mode. You can tell it's done this because on the status line at the bottom of the screen, it will say (Lisp). If it doesn't put you in Lisp mode automatically, you can type **Meta-x lisp-mode** to enter Lisp mode.

Fill mode is a "minor mode" that can be combined with Lisp mode to automatically indent each new line as you type it. You can get into fill mode by typing **Meta-x auto-fill-mode**.

As you type expressions in Lisp mode, each time you type a closing parenthesis, emacs will show you the matching opening parenthesis.

Useful formatting commands in emacs include:

- **TAB** - indent the current line correctly
- **Ctrl-SPACE** - mark one end of a region
- **C-M-**  
- indent a region correctly (from the marked line to the cursor position)
- **C-M-q** - indent the current Lisp expression correctly (i.e., from the parenthesis pointed to by the cursor to the matching
- **C-M-f** - move forward one expression
- **C-M-b** - move backward one expression

## 11 A few simple functions

Here's a recursive function that duplicates the functionality of the built-in function **reverse**:

```
(defun my-rev (l)
  ;; If the list is empty, our recursion is done
  (cond ((null l) nil)
        ;; If it's not a list, something's gone wrong
        ((not (list l)) (error "~s is not a list" l))
        ;; Else recursively reverse the cdr of the list, and
        ;; then append the car of the list. Notice that nconc
        ;; takes 2 lists, so we have to put the car into its
        ;; own list first. Try writing this function with
        ;; "(car l)" as the 2nd argument to nconc instead
        ;; of "(list (car l))" and see what happens.
```

```
;; Thought question: Why is it safe to use nconc?  
;; Answer: we don't care about the partial list  
;; constructed by the recursive call, so it's OK to  
;; destroy it.  
(t (nconc (my-rev (cdr l)) (list (car l))))
```

Here's another approach to implementing the same function, this time iterative, using the loop construct:

```
(defun my-rev (l)  
  ;; Error checking!  
  (if (not (listp l)) (error "~s is not a list~" l))  
  ;; Loop backwards through the indices of the elements of  
  ;; the list, from (n-1) (since the index used by "nth" is  
  ;; zero-based) to 0, collecting the nth element.  
  (loop for i from (- (length l) 1) downto 0  
        collect (nth i l)))
```