

LISP Tutorial for CSC244/444

Hao Zhang
Fall 2004

1. LISP = LISt Processing

1.1 lists, evaluations

LISP is normally used as an interpreter. It interprets *s-expressions*, including *lists* and *atoms*.

→ (1+ 1) ;1+ is the “increment by 1” function

2

Lists are defined recursively as sequences of lists or atoms (*symbols* or *numbers*).

→ (1+ (1+ (1+ 1)))

4

Consequently, lists are interpreted recursively in a top-down fashion:

(*function-name arg_1 arg_2 ... arg_n*)

At the bottom, numbers evaluate to themselves; symbols evaluate to the last value assigned to them.

Notice that symbols serve the dual roles of function identifiers and variable specifiers in LISP.

→ (setq 1+ 17) ;assigns 17 to the variable value of 1+, but the function value of 1+ is unchanged.

17

→ (1+ 1+) ;depending on the position of the symbol in the list, either the function value or the variable value will be retrieved.

18

1.2 setq and quote

LISP has the tendency to evaluate everything. *setq* is named for *set quote*. It is special in that it actually quotes the first argument and takes it literally as a symbol. Using *setq*, a symbol, a number, a list, all can be assigned to a variable.

We have the apostrophe character reserved for *quote* function.

→ (setq a '(1+ 1+))

```

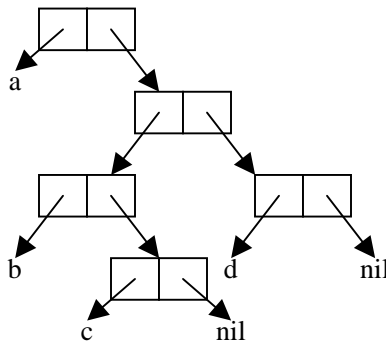
(1+ 1+)
→ (setq a '1+)
1+
→ a
1+
→ (setq a '( I saw (the man) (with (a telescope))))
( I SAW (THE MAN) (WITH (A TELESCOPE)))
→ '()
NIL

```

1.3 car, cdr, cadr, etc. cons, list, append

Now, we can quote lists. The next step is to manipulate the lists. Before doing so, we'd better have the structural representations of lists in our mind.

Lists are represented in LISP using binary trees: the left subtree of a node points to the first element of a list, and the right subtree points to the rest of the list.



The binary tree representation of list **(a (b c) d)**

A node of a LISP binary tree is often called a *cons* cell. The notation for a cons cell is called a *dotted pair*. The dotted pair representation of the list **(a (b c) d)** is

(a . ((b . (c . nil)) . (d . nil)))

Dotted pair representations are also *s-expressions*. Actually, they are more general than lists. **(a . b)** is an example of neither list nor atom.

```

→ '(a . ((b . (c . nil)) . (d . nil)))
(A (B C) D)
→ '(a . b)
(A . B)

```

Now, we summarize the list manipulation functions in terms of operations on dotted pairs.

car returns the left component of a dotted pair.

cdr returns the right component of a dotted pair.

e.g. (cdr '(A B)) is the same as (cdr '(A . (B . NIL))) which yields (B . NIL), i.e., (B)

cadr returns the left component of the right component of the given dotted pair.

e.g. (cadr '(a (b c) d)) yields (b c)

more: caaar, cadar...

The following are functions that create rather than decompose lists.

cons returns a new cons cell, left of which points to the first actual argument, right of which points to the second actual argument.

e.g. (cons 'a 'b) => (a . b)
 (cons 'a '(b)) => (a b)
 (cons '(a) 'b) => ((a) . b)
 (cons nil 'b) => (nil . b)
 (cons 'a nil) => (a)

list returns a sequence of new cons cells linked together by their right links and left links of which point to the arguments in order.

e.g. (list 'a 'b) => (a b)
 (list 'a '(b)) => (a (b))
 (list '(a) 'b) => ((a) b)
 (list nil 'b) => (nil b)
 (list 'a nil) => (a nil)
 (list nil nil nil) => (nil nil nil)

append can be thought of changing the rightmost link of each non-rightmost argument from nil to the next argument so as to link all the arguments into a longer list. The only difference is that actually implementation will clone the cons to be modified instead.

e.g. (append 'a 'b) =>Error, the first argument is not a list
 (append '(a) 'b) =>((a) . b)
 (append '(a) nil '(b)) =>(a b)

1.4 functions, predicates, conditionals, iterations

We already have a special function *setq* to save variable values. Another special function is *defun*, needed to save function definitions.

A call to *defun* looks like ths:

**(defun fname (v1 v2 ... vn)
 (...body of code 1...)**

(...body of code 2...) ...)

Functions can be nameless. Instead of using `defun` with three parts of arguments, we can use *lambda* operator with only two parts of arguments: the formal parameter list and the function body.

**(lambda (v1 v2 ... vn)
 (...body of code 1...)
 (...body of code 2...) ...)**

A symbol is *bound* in a function if it is a formal parameter of that function. Otherwise, it is *free* within that function. Modifying a variable of free symbol will cause global effects, which should be avoided.

let is the answer if you need to define local variables.

**(let ((par1 val1) (par2 val2) ... (parn valn))
 exp1
 exp2
 ...
 expm)**

let is based on *lambda*. It is equivalent to:

((lambda (par1 par2 ... parn) exp1 exp2 ... expm) (val1 val2 ... valn))

In essence, the local variables created by *let* are also formal parameters.

Global variables are declared and initialized using *defvar* or *defparameter*.

Constants are declared using *defconstant*.

The stylistic convention is to surround global variable names with asterisks, and global names with pluses.

e.g.

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

Predicates are special functions that return true or false. In LISP, false is indicated by `nil`, and true by any value other than `nil`. As a convention, LISP often returns the atom `t` to mean true. Many LISP predicate names end in `p`, for "predicate".

Predicates are usually used together with conditionals, iterations to control the execution of a program. In LISP, conditionals and iterations are also functions.

The general form of *cond* looks like:

```

(cond (exp11 exp12 exp13 ...)
      (exp21 exp22 exp23 ...)
      (exp31 exp32 exp33 ...)
      .
      .
      .
      (expn1 expn2 expn3 ...))

```

Other conditionals include *if*, *when*, *unless*, and *case*

In LISP, logical operators **and** and **or** stop evaluation when they have determined a result; hence they are useful for flow of control.

e.g.

```

(defun even-50-100 (x)
;~~~~~
; to check if an s-expression evaluates to an even number between 50 and 100
;
  (and (numberp x) (evenp x) (>x 49) (<x 101)))

```

v.s.

```

(defun even-50-100 (x)
;~~~~~
; to check if an s-expression evaluates to an even number between 50 and 100
; implemented using cond
  (cond ((numberp x)
         (cond ((evenp x)
                (cond ((>x 49) (<x 101)))))))

```

There are two types of iterations in LISP, the “do” series which are “structured” and the “unstructured” “prog” series.

e.g.

```

→ (defun do-rev(l)
    (do ((x 1 (cdr x)) (res nil (cons (car x) res)))
        ((null x) res)))

```

DO-REV

```

→ (do-rev '(a b c))
(C B A)

```

e.g.

```
(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))
```