

File tree traversal Example

```
(defun dir (root filefn dirfn)
  (let ((files (directory root)))
    (loop for f in (reverse files) do
      (if (pathname-name f)
          (funcall filefn f)
          (funcall dirfn f))))))

(defun listdirs (root)
  (labels ((dirfn (root)
            (print root)
            (dir root
              #'(lambda (f)
                  (declare (ignore f))
                  nil)
              #'dirfn)))
    (dirfn root)))

(defun listall (root)
  (labels ((dirfn (root)
            (print root)
            (dir root
              #'(lambda (f) (print f))
              #'dirfn)))
    (dirfn root)))

(defun recurse (root filefn dirfn)
  (labels ((dirfn (root)
            (funcall dirfn root)
            (dir root
              filefn
              #'dirfn)))
    (dirfn root)))
```

Output

```
* (listall #p"../")

#p"../"
#p"/home/fac/swm/PLC/Lecture/tmp/perl/"
#p"/home/fac/swm/PLC/Lecture/tmp/perl/x.pl~"
#p"/home/fac/swm/PLC/Lecture/tmp/perl/x.pl"
#p"/home/fac/swm/PLC/Lecture/tmp/perl/filter.pl"
#p"/home/fac/swm/PLC/Lecture/tmp/lisp/"
#p"/home/fac/swm/PLC/Lecture/tmp/flex/"
#p"/home/fac/swm/PLC/Lecture/tmp/flex/pascal.l~"
#p"/home/fac/swm/PLC/Lecture/tmp/flex/lex.yy.c"
#p"/home/fac/swm/PLC/Lecture/tmp/flex/a.out"
#p"/home/fac/swm/PLC/Lecture/tmp/bison/"
#p"/home/fac/swm/PLC/Lecture/tmp/bison/tests/"
#p"/home/fac/swm/PLC/Lecture/tmp/bison/calc/a.out"
NIL

* (listdirs #p"../")

#p"../"
#p"/home/fac/swm/PLC/Lecture/tmp/perl/"
#p"/home/fac/swm/PLC/Lecture/tmp/lisp/"
#p"/home/fac/swm/PLC/Lecture/tmp/flex/"
#p"/home/fac/swm/PLC/Lecture/tmp/bison/"
#p"/home/fac/swm/PLC/Lecture/tmp/bison/tests/"
#p"/home/fac/swm/PLC/Lecture/tmp/bison/calc/"
NIL
```

Functions

```
(defun square (x)
  (* x x))

(square 6)

(square (square 3))

(defun app (x y)
  (cond ((null x)
        y)
        (t (cons (car x)
                  (append (cdr x) y)))))
```

```
* (app '(a b) '(c d))

(A B C D)
* (app '(a b) 'c)

(A B . C)
* (app 'x 'y)
```

Type-error in KERNEL::OBJECT-NOT-LIST-ERROR-HANDLER: X is not of type LIST

Restarts:
 0: [ABORT] Return to Top-Level.

Debug (type H for help)

```
(CAR 1 X)[:EXTERNAL]
0] :q
```

Copying

```
(defun shallow-copy (lst)
  (cond ((null lst)
        nil)
        (t (cons (car lst)
                  (shallow-copy (cdr lst))))))
```

```
* (shallow-copy '(x y z))
```

```
(X Y Z)
```

```
(let ((x '(a b c)))
  (let ((y (shallow-copy x)))
    (eq x y)))
```

```
NIL
```

```
(defun deep-copy (lst)
  (cond ((atom lst)
        lst)
        (t (cons (deep-copy (car lst))
                  (deep-copy (cdr lst))))))
```

```
(let ((x '((a) b c)))
  (let ((y (shallow-copy x)))
    (eq (car x) (car y))))
```

```
T
```

```
(let ((x '((a) b c)))
  (let ((y (deep-copy x)))
    (eq (car x) (car y))))
```

```
NIL
```

Reversing a list

```
(defun rev (lst)
  (rev1 lst nil))

(defun rev1 (lst1 lst2)
  (cond ((null lst1)
        lst2)
        (t (rev1 (cdr lst1)
                  (cons (car lst1) lst2)))))
```

```
* (rev '(a b c d))
```

```
(D C B A)
```


Generating partitions

```
(defun partition (lst)
  (cond ((null lst)
        (list nil))
        (t (let* ((head (car lst))
                  (tail (cdr lst))
                  (tailpart (partition tail)))
              (append (mapcar #'(lambda (part)
                                (cons (list head)
                                      part))
                        tailpart)
                      (mapcan
                       #'(lambda (part)
                           (maplist
                            #'(lambda (piece1)
                                (maplist
                                 #'(lambda
                                     (piece2)
                                     (if (eq
                                         piece1
                                         piece2)
                                         (cons head
                                             (car
                                              (car
                                               piece2)))
                                         (car
                                         part))
                                     part))
                            tailpart))))))))))
```

Output

```
* (partition '(a))
((A))
* (partition '(a b))
((A) (B))
(A B))
* (partition '(a b c))
((A) (B) (C))
(A) (B C))
(A B) (C))
(B) (A C))
(A B C))
* (partition '(a b c d))
((A) (B) (C) (D))
(A) (B) (C D))
(A) (B C) (D))
(A) (C) (B D))
(A) (B C D))
(A B) (C) (D))
(B) (A C) (D))
(B) (C) (A D))
(A B) (C D))
(B) (A C D))
(A B C) (D))
(B C) (A D))
(A C) (B D))
(C) (A B D))
(A B C D))
```