

# Map and Apply

Map and Apply are two very powerful Scheme tools that are frequently misunderstood by students.

Map in general can take a function of n-arguments and n lists, but it is easier to think of it if we have a function of one argument and a single list of values. The result of f

`(map f lat)`

is a new list, whose first element is `(f (car lat))`, whose second element is `(f (cadr lat))` and so forth. The ith element of the returned list is the result of applying f to the ith element of lat.

For example,

```
(map (lambda (x) (+ x 2)) '(1 2 3 4 5))
```

returns

```
(3 4 5 6 7)
```

The second argument to map does not need to be a flat list; map takes as an argument each element at the top level of the list.

For example,

```
(map car '( (1 2) (3 (4 5)) (6)))
```

returns

```
(1 3 6)
```

Map in general can take a function of  $n$  arguments and  $n$  argument-lists, all of the same length. The result of

`(map f arg1 arg2 ... argn )`

is a new list whose  $i$ th element is the result of applying  $f$  to the  $i$ th element of each of the argument lists

For example

`(map (lambda (x y) (+ x y)) '(1 2 3) '(4 5 6))`

returns

`(5 7 9)`

Map has all kinds of useful applications. For example, suppose we have a binding list in a let expression:

```
([x 3] [y 45] [z 123])
```

We can get the list of symbols being bound, (x y z), from

```
(map car '([x 3] [y 45] [z 123]))
```

and the list of values being bound from

```
(map cadr '([x 3] [y 45] [z 123]))
```

If you write the factorial function

```
(define fact  
  (lambda (x)  
    (if (= x 1) 1 (* x (fact (- x 1))))))
```

and what to check it out quickly, you can do so with

```
(map fact '(1 2 3 4 5 6 7))
```

and get

```
(1 2 6 24 120 720 5040)
```

Apply has a simpler definition, but I find that students have a harder time thinking about it. If  $f$  is a function of  $n$  arguments and  $L$  is a list of  $n$  elements,

$(\text{apply } f \ L)$

is the result of calling  $f$  with the elements of  $L$  as its arguments.

For example,  $(+ \ '(2 \ 3))$  makes no sense but  $(\text{apply } + \ '(2 \ 3))$  does make sense and has the value 5, as you would expect.

We can define a procedure that finds the distance of a 2D point from the origin:

```
(define dist  
  (lambda (x y)  
    (sqrt (+ (* x x) (* y y)))))
```

(dist 3 4) correctly returns 5.

However, if we have a point *p* defined as a pair: (x y) we can't use *dist* to find its distance from the origin because *dist* wants 2 separate arguments. However we can do this with *apply*:

```
(apply dist p)
```



Max is a pre-defined Scheme function that takes any number of numerical arguments and returns the largest of its arguments.

For example,

```
(max 2 5 6 3 9 5 6)
```

returns 9.

We might want to find the maximum value of a list; we can get this with

```
(apply max lat)
```

Map and apply are often used together to recurse on a structured list.

For example, here is a function that finds the largest number in a structured list of numbers, such as (2 (4 5 (6)) 3 (4 (5))):

```
(define largest
  (lambda (L)
    (cond
      [(null? L) -1]
      [(number? L) L]
      [else (apply max (map largest L))])))
```

Here is a function that counts the number of atoms in an S-expression. Remember that an S-expression can be null, an atom, or a list:

```
(define count
  (lambda (L)
    (cond
      [(null? L) 0]
      [(not (pair? L)) 1] ; this means L is an atom
      [else (apply + (map count L))])))
```