

Topic 2: Recursion, recursion, recursion

Guy McCusker¹

¹University of Bath

Recursive functions on lists

This lecture will contain several examples of simple list-functions.

They will all be recursive functions, and they will all follow a very similar pattern:

- ▶ Ask whether the input list is empty. If it is, return an answer.
- ▶ If the input list is not empty, split it into its `car` and `cdr`.
- ▶ Compute the answer using a recursive call on the `cdr`, together with some work using the `car` if necessary.

A reminder: the `len` function

```
(defun len (mylist) (  
  if (null mylist)  
      0  
      (+ 1 (len (cdr mylist)))  
  )  
)
```

- ▶ Declare the function with `defun`
- ▶ Test whether the input list is empty
- ▶ If it is, return an answer
- ▶ If it is not, compute the answer using a recursive call on the `cdr`.

Is this list a list of atoms?

Our next example will test whether a list contains nothing but atoms.

We'll return `t` if it does and `nil` — also known as the empty list, `()` — if it does not.

We'll follow the common pattern.

When is a list a list of atoms?

- ▶ Is the empty list a list of atoms?
 - ▶ Yes. It doesn't contain anything that is not an atom.
- ▶ Is a non-empty list a list of atoms?
 - ▶ Maybe.
 - ▶ If its `car` is an atom, then it might be.
 - ▶ If its `cdr` is also a list of atoms, then it is.

Base cases

```
(defun lat (mylist)
  (if (null mylist)
      t
      (...))
)
```

- ▶ Declare the function.
- ▶ Test whether the input is empty.
- ▶ If it is, return an answer.
- ▶ If it is not, ...

Recursive steps

The recursive step is going to check whether a non-empty list is a list of atoms.

There are two questions to ask.

- ▶ Is the car of the list an atom?
(atom (car mylist))
- ▶ Is the cdr of the list a list of atoms?
(lat (cdr mylist))

Note that the second of these is a recursive call.

So what's the answer?

The answer will be `t` when *both* of these questions are answered with `t`. That is:

```
(and (atom (car mylist))
      (lat (cdr mylist)) )
```

So our final function is:

```
(defun lat (mylist)
  (if (null mylist)
      t
      (and (atom (car mylist))
            (lat (cdr mylist)) )
  )
)
```

Let's try it out

```
user> (defun lat (mylist)
        (if (null mylist)
            t
            (and (atom (car mylist))
                  (lat (cdr mylist)) )
        )
      )
lat
user> (lat ())
#t
user> (lat '(a b 123 foo bar))
#t
user> (lat '(a b (123 foo) bar))
()
```

Is this atom in this list?

Let's write a function to find out whether a given atom is in a given list
i.e. whether it is a *member* of the list.

We follow the familiar pattern:

```
(defun mem (a mylist)
  (if (null mylist)
      ...
      ...
  )
)
```

When is an atom a member of a list?

- ▶ If the list is empty, the atom is not a member.
- ▶ If the list is non-empty, there are two ways the atom might be a member:
 - ▶ it might be the car of the list
 - ▶ it might be a member of the cdr of the list.

When is an atom a member of a non-empty list?

The atom is a member of a non-empty list if:

- ▶ either the atom is the same as the car
(eq a (car mylist))
- ▶ or it is a member of the cdr
(mem a (cdr mylist))

We can perform this either-or test using or:

```
(or (eq a (car mylist))
    (mem a (cdr mylist)))
```

Putting it all together

```
(defun mem (a mylist)
  (if (null mylist)
      nil
      (or (eq a (car mylist))
          (mem a (cdr mylist))))
  )
)
mem
user> (mem 1 '(3 2 1 2 3))
#t
user> (mem 'foo '(3 2 1 2 3))
()
```

Building new lists from old

In the next couple of examples, we'll write functions which take lists as input and return new lists as output.

We'll build these new lists using `cons`.

The first example takes a list `mylist` and an atom `a`, and gives back a new list which is the same as `mylist` one, but with the first occurrence of `a` (if there is one) removed.

Since this function removes a *member* from a list, we'll call it `rember`.

How does `rember` work?

The pattern is the same as always:

```
(defun rember (a mylist)
  (if (null mylist)
      ...
      ...
  )
)
```

If `mylist` is empty, then removing a member should return the empty list, so:

```
(if (null mylist)
    '()
    ...
)
```

What if `mylist` is not empty?

To remove `a` from a non-empty list, we have to find the first occurrence of `a`, if there is one.

Maybe the `car` of `mylist` is `a`.

If it is, what should we return?

In this case, we should return `(cdr mylist)`.

What if the `car` is not `a`?

If the `car` of `mylist` is not `a`, what should we do?

We need to return a list. Its first element should be the same as the first element of `mylist`. What should the rest be?

The rest should be the result of removing the first occurrence of `a` from `(cdr mylist)`.

That is, the rest should be `(rember a (cdr mylist))`.

So we should return

```
(cons (car mylist) (rember a (cdr mylist)))
```

Let's write the function

```
(defun rember (a mylist)
  (if (null mylist)
      '()
      (if (eq a (car mylist))
          (cdr mylist)
          (cons (car mylist) (rember a (cdr mylist))))
  )
)
```

Trying it out

```
user> (rember 'ham '())
()

user> (rember 'ham '(ham tomato mozzarella))
(tomato mozzarella)

user> (rember 'ham '(tomato ham mozzarella))
(tomato mozzarella)

user> (rember 'ham '(bacon lettuce tomato))
(bacon lettuce tomato)

user> (rember 'ham '(ham tomato ham mozzarella))
(tomato ham mozzarella)
```

Remove them all!

There's a simple change we can make to the `rember` function which will remove *all* occurrences of the given atom rather than just the first.

Can you see what it is?

Lists of lists

Since Lisp is so list-based, we tend to use lists to structure most of our data.

A big advantage is the ability to put lists inside other lists. This lets us create lists of pairs, things that look like trees, and all kinds of other interesting data structures.

For instance, we might store a list of students on a course as a list of lists, each sublist containing a student's *given name* and *family name* as the two elements. Like this:

```
((Andy Budd) (Richard Rutter) (Jeremy Keith))
```

Getting the first names

Let's write a function *firsts* which returns a list containing just the first names (given names) from a list like this.

As usual it will be a recursive function, and it will work as follows:

- ▶ Check if the input list is empty, and if so respond appropriately.
- ▶ If the input list is not empty, build the answer list by using a recursive call on the `cdr` plus information from the `car`.
- ▶ Build the answer list using `cons`.

Thinking it through

- ▶ If the input list is empty, the answer list should be empty.
- ▶ If the input list is not empty:
 - ▶ the first element of the answer list should be the first name from the `car`
 - ▶ the rest of the answer list should be the rest of the first-names.

I hope the recursive call is obvious from the above description.

Empty input list

```
(defun firsts (mylist)
  (if (null mylist)
      '()
      ...
  )
)
```

Non-empty input list

If `mylist` is not empty, the first first-name we have to return is the first-name from `(car mylist)`.

That is to say, the first first-name is `(car (car mylist))`.

The rest of the first-names will be returned by a recursive call: `(firsts (cdr mylist))`.

We build the answer list using `cons` to attach the first element to the rest, like this:

```
(cons (car (car mylist)) (firsts (cdr mylist)))
```

The complete function

```
(defun firsts (mylist)
  (if (null mylist)
      '()
      (cons (car (car mylist)) (firsts (cdr mylist)))
  )
)
```

Trying it out

```
user> (defun firsts (mylist)
      (if (null mylist)
          '()
          (cons (car (car mylist)) (firsts (cdr mylist)))
      )
    )
firsts
user> (firsts '((Andy Budd) (Jeremy Keith) (Richard Rutter)))
(Andy Jeremy Richard)
```

An abbreviation

When working with lists of lists and so on, it is common to want to take the `car` of the `car`, or the `cdr` of the `car`, or the `car` of the `cdr`, or ...

Most Lisp implementations provide some abbreviations for these things:

```
car
cdr
caar
cadr
cdar
cddr
caaar
caadr
```

and so on.

Sanity check: what does `cadr` do?

In principle, `cadr` could be one of two things: the `car` of the `cdr` or the `cdr` of the `car`.

Let's test it out to find out which it is.

```
user> (cadr '((Andy Budd) (Richard Rutter)))
(Richard Rutter)
```

So which is it?

Using the abbreviation in `firsts`

Using these abbreviations lets us shorten

```
(defun firsts (mylist)
  (if (null mylist)
      '()
      (cons (car (car mylist)) (firsts (cdr mylist))))
  )
)
```

to

```
(defun firsts (mylist)
  (if (null mylist)
      '()
      (cons (caar mylist) (firsts (cdr mylist))))
  )
)
```

It's shorter. Is it better? Is it clearer? Is it faster?

Some more list-building functions

Here are some simple exercises. Try to write the following functions:

- ▶ `insertR` takes a list `mylist` and two atoms `old` and `new`. It builds a new list which is like `mylist` but if there is an occurrence of `old`, it inserts `new` just after the first one. For example,
`(insertR 'anchovy 'ham '(tomato mozzarella ham caper))`
should return the list
`(tomato mozzarella ham anchovy caper)`
- ▶ `insertL` is like `insertR` but inserts the new element *before* the old one.
- ▶ `subst` is similar, but instead of inserting the new element next to the old one, it *replaces* the old element with the new one.

And some more

When you've written those functions, see how easy it is to write these ones:

- ▶ `multiinsertR` is like `insertR` but inserts the new element after *every* occurrence of the old one, not just the first. For example,
`(multiinsertR 'anchovy 'ham '(ham tomato more ham))`
should return the list
`(ham anchovy tomato more ham anchovy)`
- ▶ `multiinsertL` is like `insertL` but inserts the new element in front of every occurrence of the old one.
- ▶ `multisubst` is like `subst` but replaces all occurrences of the old element with the new one.