

CM20167 Topic 3: Recursion on other data

Guy McCusker

1W2.1

Recursive functions on lists

We've seen how to write recursive functions on lists.

The key ingredients were:

- ▶ test whether your list is empty. This is the *termination condition* for the recursion: once we get to an empty list, we don't make any more recursive calls.
- ▶ if the list is not empty, perform a recursion on the `cdr` of the list, and compute the answer using that, plus information from the `car`.

Recursive functions on numbers

In this lecture, we'll see how to write recursive functions which operate on other kinds of data, starting with *numbers*.

For the purposes of this lecture, we're talking about the *natural numbers*, that is, integers from 0 up.

Plus, recursively

$$4 + 0 = 4$$

$$\begin{aligned}4 + 2 &= 1 + (4 + (2 - 1)) \\ &= 1 + (4 + 1) \\ &= 1 + 1 + (4 + (1 - 1)) \\ &= 1 + 1 + (4 + 0) \\ &= 1 + 1 + 4 \\ &= 6.\end{aligned}$$

So if we can *detect zero*, *subtract one from things* and *add one to things* then we can do arbitrary additions.

Building blocks

```
user> (zerop 0)
#t
user> (zerop 1)
()
```

In EuScheme, `zerop` tests for zero.

The `p` in `zerop` stands for *predicate*, as in predicate logic. A predicate is something that tells you whether a value has a particular property.

Predicates in EuScheme

There are lots of built-in predicates in EuScheme. For instance:

```
listp
numberp
booleanp
consp
symbolp
keywordp
complexp
floatp
```

More building blocks

```
;; add one to a number, to build a bigger one

(defun add1 (n) (+ 1 n))

;; subtract one from a number, to break
;; a bigger number into a smaller one

(defun sub1 (n) (- n 1))
```

A pattern for recursion

Remember the pattern for recursion on lists:

- ▶ test whether your **list is empty**; if so return an answer
- ▶ if not, make a **recursive call on the smaller list** you get from `cdr`.
- ▶ if the answer you're computing is a list, **build it up using cons**.

The corresponding pattern for numbers is:

- ▶ test whether your **number is zero**; if so return an answer
- ▶ if not, make a **recursive call on the smaller number** you get from `sub1`.
- ▶ if the answer you're computing is a number, **build it up using add1**.

Let's add!

```
(defun o+ (n m)
  (if (zerop m)
      n
      (add1 (o+ n (sub1 m)))))
)
```

Try it out

```
user> (o+ 3 4)
7
user> (o+ 3 0)
3
user> (o+ 0 3)
3
user> (o+ -1 4)
3
user> (o+ 4 -1)
Abort: value stack overflow
happened in: #<Code o+>
```

Negative numbers do not work well! At least not in the argument where we're doing the recursion.

This should not be a big surprise.

Testing strategy

It's worth remarking on the choices I made for values to test the function:

- ▶ some ordinary values which I expect will work fine: 3 and 4
- ▶ some *boundary cases*: 0 with 3 and 3 with 0—this will help me check that my termination conditions are okay, for example.
- ▶ some cases on the other side of the boundary, where things might break: putting -1 in there.

In this case I got a pleasant surprise ((o+ -1 4) works) and a slightly annoying non-surprise ((o+ 4 -1) blows up).

Thanks to a decent testing strategy I know pretty much everything I need to know about how my function behaves.

Go forth and multiply!

Can we write a multiplication function in the same way?

Here's a big hint.

$$\begin{aligned}n \times 0 &= 0 \\ n \times m &= n + (n \times (m - 1))\end{aligned}$$

Multiplication, recursively

```
(defun o* (n m)
  (if (zerop m)
      0
      (o+ n (o* n (sub1 m)))
  )
)

user> (o* 3 4)
12
user> (o* 3 1)
3
user> (o* 3 0)
0
user> (o* 2 -12)
Abort: value stack overflow
happened in: #<Code o+>
```

More testing

```
user> (o* -12 2)
Abort: value stack overflow
happened in: #<Code o+>
```

Were you surprised that supplying a negative number in either argument caused a stack overflow? Can you see why it did? What about this:

```
user> (o* -12 1)
-12
```

Surprised now?

Recursion on pairs of things

Now for something a little different: a recursion on two arguments. We're going to "zip together" two lists of the same length:

```
user> (zip '(1 2 3 4) '(a b c d))
((1 a) (2 b) (3 c) (4 d))
```

This is a useful sort of thing to do: if I have a list containing all the students names, and another containing all their email addresses, zipping them together will give me an address book.

Ziping

This is just like the other recursive functions we wrote on lists, but now there are two lists we need to work our way down.

We therefore

- ▶ test *both lists* for emptiness when we terminate
- ▶ make a recursive call on the *cdrs* of *both lists*.

Ziping

```
(defun zip (l m)
  (if (and (null l) (null m))
      ()
      (...))
  )
```

What's the recursive call?

We can zip together the two *cdrs* easily enough:

```
(zip (cdr l) (cdr m))
```

Ziping

Now we need to add something to the beginning.

It should be a list containing the two cars.

We can form this using

```
(list (car l) (car m))
```

`list` takes a list of arguments, evaluates them, and returns the list containing their values.

Ziping

```
(defun zip (l m)
  (if (and (null l) (null m))
      ()
      (cons (list (car l) (car m)) (zip (cdr l) (cdr m))))
  )
)

user> (zip '(1 2 3 4) '(a b c d))
((1 a) (2 b) (3 c) (4 d))
user> (zip '(1 2 3) '(a b c d))
Continuable error---calling default handler:
Condition class is #<class bad-type>
message:      "incorrect type in cdr"
value:        ()
```

A little extra robustness

Although we don't have to, we can add a little robustness to this function.

At the moment it causes errors if the two lists are not the same length. That's not surprising: we've coded it so that it only handles the case when they are the same length.

With lists of differing lengths, what happens?

Lists of different lengths

The error message tells us what happens: we end up trying to take the cdr of an empty list.

That's because, in the recursion, one list has become empty and the other has not.

What could we do in this case? One option is to throw away the rest of the non-empty list:

```
user> (zip '(1 2 3) ())
()
user> (zip () '(1 2 3))
()
```

A more robust recursion

- ▶ If both lists are empty: return ()
- ▶ If one is empty but the other is not: return () too
- ▶ If they're both non-empty: perform our recursive call.

Simple! Just replace `and` with `or`.

Another choice

An alternative would be not to throw away the remainder but to keep it in the zipped version, unpaired:

```
user> (zip '(1 2 3) '(a b c d))  
((1 a) (2 b) (3 c) d)  
user> (zip '(1 2 3 4) '(a b c))  
((1 a) (2 b) (3 c) 4)
```

Another more robust recursion

- ▶ If both lists are empty: return ()
- ▶ If `l` is empty but `m` is not: return `m`
- ▶ If `m` is empty but `l` is not: return `l`
- ▶ If they're both non-empty: perform our recursive call.

This gives us *four cases* to consider, not just two.

We can do that using a `cond` test, rather than `if`.

Conditional

The general form of `cond` is

```
(cond (test1 expr1)
      (test2 expr2)
      ...
      (testn exprn))
```

Tests are evaluated in order until one of them comes out true. Then the corresponding expression is evaluated, and the rest are ignored.

This means that if you put `t` as the final test, you get an “else” clause which is executed when all the others fail.

Zip again

```
(defun zip (l m)
  (cond ((and (null l) (null m)) ())
        ((null l) m)
        ((null m) l)
        (t (cons (list (car l) (car m)) (zip (cdr l) (cdr m))))))
)
```

We don't really need the first line of the conditional because the second (and the third!) do the job just as well.

```
(defun zip (l m)
  (cond ((null l) m)
        ((null m) l)
        (t (cons (list (car l) (car m)) (zip (cdr l) (cdr m))))))
)
```

Recursion on pairs of numbers

Of course we can use a similar pattern on pairs of numbers.

When is $n > m$?

- ▶ If they're both zero, the answer is false.
- ▶ If m is zero but n is not, the answer is true (assuming we're working with natural numbers).
- ▶ If n is zero but m is not, the answer is false.
- ▶ If they're both non-zero, the answer is the same as the answer to $n - 1 > m - 1$.

Inequality

```
(defun o> (n m)
  (cond ((and (zerop n) (zerop m)) ())
        ((zerop m) t)
        ((zerop n) ())
        (t (o> (sub1 n) (sub1 m)))))
)
```

```
user> (o> 3 4)
```

```
()
```

```
user> (o> 4 3)
```

```
#t
```

Can we drop the first test in this conditional, like we did before? Think about it carefully...

Some exercises

Write some more functions along these lines:

- ▶ the $<$ test
- ▶ test for equality of natural numbers
- ▶ the exponentiation function: raise n to the power m
- ▶ integer division and modulo

The message

There are two things you should take from the lectures so far:

- ▶ recursion is easy, and there are patterns everywhere
- ▶ if we have a *test for zero* and *successor and predecessor* (add and subtract one) operations, we can do a lot of computation with numbers.

A slightly longer message

Although we haven't said as much, lists are often implemented simply as pairs of values: a non-null list is just the pair of its `car` and `cdr`. So we can add to our message:

- ▶ if we have a *test for null* and *pairing and unpairing* operators, we can do a lot of computation with lists.

When we come to study the λ -calculus, which is a very small, simple model of computation, we'll show how to implement recursion, successor, predecessor, pairing and splitting.

If we remember this message, we'll know this means we can do just about anything.

