

CM20167

Topic 4: Map, Lambda, Filter

Guy McCusker

1W2.1

Outline

- 1 Introduction to higher-order functions
- 2 Map
- 3 Lambda
- 4 Filter

Putting the fun in functional programming

The power of functional programming comes from *higher order functions*.

Higher-order functions are those functions which take other functions as arguments.

This idea lets us write functions which embody commonly used algorithms or processes, and then employ them in specific cases by supplying appropriate helper functions as argument.

Example: sorting

Consider the problem of sorting a list.

As long as we have some way of comparing the elements of a list, so that we know what order they should be sorted into, we can write our sorting algorithm.

Suppose we're given a list of data concerning the marks for students on a particular degree programme. For each student we have their name and all their marks for all the courses they've taken. How should we sort them?

Example: sorting

We could sort them

- ▶ in alphabetical order, by name: useful for publishing marks lists
- ▶ in descending order of average marks: useful for deciding who gets a First, who gets a 2:1 etc
- ▶ in descending order of project marks: useful for deciding who should win special prizes and so on.

There are any number of ways to sort them.

In all cases, the method for *sorting* should be the same. It's only the method of *comparing* that changes.

Higher-order functions to the rescue

With higher-order functions, we can do this:

```
(defun sort (compare list) (  
  ...  
  ;; using the compare function to compare elements  
  ;; sort the list using your favourite sorting algorithm  
)
```

We can now supply this function with the same list but different compare functions, and get different sortings.

(In Java you can do something similar using the `Comparator` interface. Object-oriented programming is another way of harnessing the power of higher-order programming.)

Map, filter, lambda

We are going to look at two well-used examples of higher-order functions, called map and filter.

We'll also look at a related idea: functions that *return other functions*. To do this we'll need a construct called lambda.

Outline

- 1 Introduction to higher-order functions
- 2 Map
- 3 Lambda
- 4 Filter

Map

One of the major benefits of computers is their ability to process large amounts of data. That is to say, to take a list of things and *do something to each of them*.

The map function does just that: it takes a function and a list, and applies the function to each element of the list, returning a new list.

So if square takes a number and computes its square, then

```
(map square '(1 2 3 4 5))
```

returns

```
(1 4 9 16 25)
```

Definition of map

In EuScheme, `map` is a built-in function. But if it were not, we could define it easily:

```
(defun map (f l) (  
  if (null l)  
      ()  
      (cons (f (car l)) (map f (cdr l)))  
  )  
)
```

Revisiting some old examples

The functions `firsts` and `seconds` from previous lectures can easily be rewritten in terms of `map`.

`firsts` takes a list of pairs and returns the list of all the first elements, so `(firsts l)` is the same as

```
(map car l)
```

Similarly, `(seconds l)` is the same as `(map cadr l)`.

New function definitions

We can make this connection explicit, and take advantage of `map`, by defining `firsts` and `seconds` differently:

```
(defun (firsts l)  
  (map car l)  
  )  
  
(defun (seconds l)  
  (map cadr l)  
  )
```

This is saying that to compute `(firsts l)` for any list `l`, we need to compute `(map car l)`; and similarly for `seconds`.

An idea

When we say that every `(firsts 1)` is the same as `(map car 1)`, we're almost saying that the *function* `firsts` is the same as `map car`.

Wouldn't it be nice if we could say that directly in Lisp?

Currying

Unfortunately, `map car` is not really a function.

`map` is a function that takes two arguments. If you give it just one argument, Lisp gives you an error.

But in spirit, `map car` is something that you could give one more argument to, and it would compute an answer.

Isn't that what a function is?

Currying

Currying, named after a mathematician called Haskell B. Curry, is an important idea in functional programming.

It's the idea that you can think of a "function of two arguments" as a "function that takes one argument and gives back a new function".

So the curried version of `+` takes one argument, like `2`, and gives back a function which adds two to things, for instance.

Similarly, a curried version of `map` could be used to define `firsts` directly: `firsts` would be the function `map car`.

Some curried functions

Warning: this is not real Lisp.

- ▶ `+ 2` is a function that adds two to things: if you give it an argument `3`, you get `(+ 2 3)` which evaluates to `5`.
- ▶ `map car` is like `firsts`: if you give it an argument `l` you get `(map car l)` which is the same as `(firsts l)`.
- ▶ If `sort` is the generalized sorting function we discussed earlier, then `sort <` is a function which takes a list and sorts it using `<` for the comparison operator; that is, it's the usual ascending-order sorting function for lists of numbers.

How can we write this kind of function in Lisp?

Outline

- 1 Introduction to higher-order functions
- 2 Map
- 3 Lambda
- 4 Filter

Anonymous functions

So far, all our functions have had *names*, such as `firsts`. The name was given to the function when we defined it:

```
(defun firsts (l)
  ...;; compute (firsts l) somehow, and return it
)
```

If we want to write a curried version of `map`, it will have to look something like

```
(defun curried-map (f)
  ... ;; return a function that takes a list
  ;; and applies f to each element of it
)
```

That is, we need to return a function, which will not have a name.

Anonymous functions

In Lisp, we can write an anonymous function using the `lambda` expression.

Here is an anonymous function which adds two to its argument:

```
(lambda (x) (+ 2 x))
```

And here it is in use:

```
user> (lambda (x) (+ 2 x))
#<Procedure #1d5858>
user> ((lambda (x) (+ 2 x)) 5)
7
```

When I just type in the expression, Lisp evaluates it. Because it evaluates to a function, Lisp tells me that the “answer” is a procedure stored at a particular address. Not much use.

When I *use* the expression, by applying it to something, it does what we expect.

Lambda

An expression `(lambda (x y z) (expression))` evaluates to an anonymous function which

- ▶ takes three arguments (in this case)
- ▶ when applied to three values, evaluates `expression` with the variables `x`, `y` and `z` bound to those three variables.

It's called `lambda` because Lisp was inspired by the lambda-calculus, which we'll study later. The Greek letter λ is used to introduce functions in just the same way in the lambda-calculus.

Returning functions

Now we know how to write functions which have no name, we can start returning them as answers from other functions.

```
(defun curried-plus (n)
  (lambda (m) (+ n m))
)
```

Here it is in action:

```
user> curried-plus
#<Procedure curried-plus>
user> (curried-plus 4)
#<Procedure #1d4e08>
user> ((curried-plus 4) 7)
11
```

Curried map

We can now define the curried version of map, too.

```
(defun curried-map (f)
  (lambda (l) (map f l))
)
```

The expression `(curried-map car)` now really is a function, and it's the same as `firsts`:

```
>(curried-map car)
#<Procedure #1d4208>
> ((curried-map car) '((Joey Ramone) (Johnny Ramone)
                      (Tommy Ramone) (DeeDee Ramone)))
(Joey Johnny Tommy DeeDee)
```

But how can we define `firsts` as `(curried-map car)`?

setq

In Lisp, we put a value into a variable using `setq`.

```
(setq mysymbol (expression))
```

turns `mysymbol` into a variable which holds the value of `(expression)`.

```
user> (setq firsts (curried-map car))
#<Procedure #1d4208>
user> (firsts '((Joey Ramone) (Johnny Ramone)
              (Tommy Ramone) (DeeDee Ramone)))
(Joey Johnny Tommy DeeDee)
```

defun is setq and lambda

- ▶ `lambda` creates functions.
- ▶ `setq` stores things in variables, i.e. gives names to things.
- ▶ `defun` creates a function and gives it a name.

In fact `defun` is just a handy shorthand for a combination of `lambda` and `setq`.

```
(defun myfunction (arglist) (expression))
```

is the same as

```
(setq myfunction (lambda (arglist) (expression)))
```

Programming hint: use setq to name things

If you're developing a program, you often want to test your functions on some input data.

It's a big time-saver to give that input data a name using `setq`. For instance, if I put

```
(setq ramones '((Joey Ramone) (Johnny Ramone)
                (Tommy Ramone) (DeeDee Ramone)))
```

in my program file, I can test out my `firsts` function with

```
(firsts ramones)
```

instead of typing the whole test-list every time.

Outline

- 1 Introduction to higher-order functions
- 2 Map
- 3 Lambda
- 4 Filter

Filter

The `filter` function is similar to `map`, but it is used not to manipulate every element of a list. Instead, it is used to *select* elements for later manipulation.

It takes as arguments a *predicate*—a function which returns a truth value telling us whether to keep an element or not—and a list to process.

`(filter p l)` goes through the list `l` and returns a new list containing those elements of `l` which make the predicate `p` true.

So if `even` is a predicate that returns true when it is given an even number, and false otherwise, then `(filter even l)` gives you the list of all even numbers in `l`.

Filter and Map

```
user> (setq onetoten '(1 2 3 4 5 6 7 8 9 10))
(1 2 3 4 5 6 7 8 9 10)
user> (defun even (n) (zerop (% n 2)))
even
user> (even 2)
#t
user> (even 5)
()
user> (map even onetoten)
(() #t () #t () #t () #t)
user> (filter even onetoten)
(2 4 6 8 10)
```

Defining filter

Unlike `map`, `filter` is not built in to EuScheme so we have to define it ourselves.

(In other functional languages it is built in).

It's easy to define recursively. Follow the usual pattern!

Defining filter

```
(defun filter (p l)
  (if (null l)
      ()
      (if (p (car l))
          (cons (car l) (filter p (cdr l)))
          (filter p (cdr l)))
      )
  )
)
```

Exercise

Simplify this definition: replace the nested ifs with a `cond`.

Example program: the Sieve of Eratosthenes

Let's write a reasonably interesting program using `filter`, `lambda` and recursion.

From Wikipedia:

Definition

In mathematics, the Sieve of Eratosthenes is a simple, ancient algorithm for finding all prime numbers up to a specified integer.

The Algorithm

Here's how Wikipedia describes the sieve algorithm.

- 1 Write a list of numbers from 2 to the largest number you want to test for primality. Call this List A.
- 2 Write the number 2, the first prime number, in another list for primes found. Call this List B.
- 3 Strike off 2 and all multiples of 2 from List A.
- 4 The first remaining number in the list is a prime number. Write this number into List B.
- 5 Strike off this number and all multiples of this number from List A.
- 6 Repeat steps 4 and 5 until no more numbers are left in List A.

A functional version

We will write a function `sieve`, which we'll apply to a list of numbers like (2 3 4 5 6 7 8 9 10).

The function will return a list of the primes that it finds in this list.

That is to say, "List A" is the *argument* to `sieve`. "List B" is the output of `sieve`.

"Write this number into List B" means "Add this number to the output list using `cons`".

"Strike this number and all multiples from List A" means "Compute a new argument for `sieve` by filtering out this number and all multiples".

"Repeat steps 4 and 5" means "Make a recursive call".

An outline of sieve

```
(defun sieve (l)
  (if (null l)
      ()
      (cons (car l) (sieve (something)))
  )
)
```

- ▶ When List A (the input) is empty, we've finished and there's no more to output.
- ▶ When List A is not empty, we know its first element is prime, so output it using cons.
- ▶ Then make a recursive call to keep going...

What should the recursive call be?

The recursive call means "carry on sieving".

Before we carry on sieving, "List A" is supposed to be purged of all multiples of the number we just output.

We can compute this purged list using a filter:

```
sieve (filter something l)
```

will carry on sieving, with a filtered input list, i.e. a purged List A.

But what should the filter be?

Filtering out multiples

Suppose we've just output the prime number 17.

We need to tell filter to throw away all multiples of 17.

That is, we need it to *keep* anything that is not divisible by 17.

A number is not divisible by 17 if $(\% n 17)$ is not zero. So our filter could be:

```
(defun filterfor17 (n)
  (not (zerop (% n 17)))
)
```

Too many defuns

But we can't write all the filters we need in this way!

We'd need to define a function `filterforNNN` for every prime number. That's clearly impossible.

Lambda to the rescue!

Instead of defining each of these functions with a `defun`, produce it with a `lambda`. We can write a generic function to produce them.

Indivisibility using lambda

```
(defun indivisible (m)
  (lambda (n)
    (not (zerop (% n m))))
  )
```

Now `(indivisible 17)` gives us the filter we need to purge multiples of 17; `(indivisible 39)` does the same for 39; and so on.

The finished sieve

```
(defun sieve (l)
  (if (null l)
      ()
      (cons (car l) (sieve (filter (indivisible (car l))
                                   (cdr l)))))
  )

user> (sieve '(2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19))
(2 3 5 7 11 13 17 19)
```

Generating lists to sieve

It's inconvenient to have to type out that long list of numbers from 2 to a billion.

We can produce them with a function:

```
(defun fromto (n m)
  (if (> n m)
      ()
      (cons n (fromto (+ n 1) m))
  )
)
```

And then we can produce as many primes as we like:

```
(defun primes (n)
  (sieve (fromto 2 n))
)
```

Or can we?

Trying it out

```
user> (primes 100)
(2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73
79 83 89 97)
```

```
user> (primes 1663)
(2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73
79 83 89 97 101 103 107 109 113 127 131 137 139 149 151 157
163 167 173 179 181 191 193 197 199 211 223 227 229 233 239
241 251 257 263 269 271 277 281 283 293 307 311 313 317 331
337 347 349 353 359 367 373 379 383 389 397 401 409 419 421
431 433 439 443 449 457 461 463 467 479 487 491 499 503 509
521 523 541 547 557 563 569 571 577 587 593 599 601 607 613
617 619 631 641 643 647 653 659 661 673 677 683 691 701 709
719 727 733 739 743 751 757 761 769 773 787 797 809 811 821
823 827 829 839 853 857 859 863 877 881 883 887 907 911 919
929 937 941 947 953 967 971 977 983 991 997 1009 1013 1019
1021 1031 1033 1039 1049 1051 1061 1063 1069 1087 1091 1093
1097 1103 1109 1117 1123 1129 1151 1153 1163 1171 1181 1187
1193 1201 1213 1217 1223 ...)
```

```
user> (primes 1664)
Abort: value stack overflow
happened in: #<Code fromto>
```

Maybe my computer needs a beer

A lot of recursion

There's an awful lot of recursion in these definitions.

Lots of recursion means lots of use of the stack.

Lots of use of the stack means you run out of space.

We can improve on this by making all the functions *tail recursive*: make sure the recursive call is the very last thing the function does.

Then Lisp is clever enough to realise it does not need to use the stack at all, and you can handle much longer chains of calls.

More on this another time.