CSci 2041
Advanced Programming Principles
L10: Higher Order Functions, part 2

Eric Van Wyk

Fall 2014

So far,

- We've seen how to pass "helper" equality or ordering functions into list and tree processing functions such as `find_by` and `sort_by`.
- We've seen how to specify functions in a number of ways:
  - let-expr declarations
  - lambda expressions
  - using curried functions
  - converting operators into functions
- We now consider functions that implement different "design patterns" of computations over lists.

Map, Filter, and Fold

We can use higher order functions to perform computations over lists where we might otherwise write a recursive function.

For example,

```
let inc x = x + 1
let r1 = map inc [1;2;3;4;5]

let even n = n mod 2 = 0
let evens = filter even [1;2;3;4;5;6;7]

let sum xs = fold (+) 0 xs
```

Paradelle for Susan

I remember the quick, nervous bird of your love.
I remember the quick, nervous bird of your love.
Always perched on the thinnest, highest branch.
Always perched on the thinnest, highest branch.
Thinnest love, remember the quick branch.
Always nervous, I perched on your highest bird the.

It is time for me to cross the mountain.
It is time for me to cross the mountain.
And find another shore to darken with my pain.
And find another shore to darken with my pain.
Another pain for me to darken the mountain.
And find the time, cross my shore, to with it is to.

The weather warm, the handwriting familiar.

Another pain for me to darken the mountain.
And find the time, cross my shore, to with it is to.

The weather warm, the handwriting familiar.
The weather warm, the handwriting familiar.
Your letter flies from my hand into the waters below.
Your letter flies from my hand into the waters below.
The familiar waters below my warm hand.
Into handwriting your weather flies you letter the from the.

I always cross the highest letter, the thinnest bird.
Below the waters of my warm familiar pain,
Another hand to remember your handwriting.
The weather perched for me on the shore.
Quick, your nervous branch flew from love.
Darken the mountain, time and find was my into it was with to to.

NOTE: The paradelle is one of the more demanding French fixed forms, first appearing in the *langue d'oc* love poetry of the eleventh century. It is a poem of four six-line stanzas in which the first and second lines, as well as the third and fourth lines of the first three stan-

I always cross the highest letter, the thinnest bird.
Below the waters of my warm familiar pain,
Another hand to remember your handwriting.
The weather perched for me on the shore.
Quick, your nervous branch flew from love.
Darken the mountain, time and find was my into it was with to to.

NOTE: The paradelle is one of the more demanding French fixed forms, first appearing in the *langue d'oc* love poetry of the eleventh century. It is a poem of four six-line stanzas in which the first and second lines, as well as the third and fourth lines of the first three stanzas, must be identical. The fifth and sixth lines, which traditionally resolve these stanzas, must use *all* the words from the preceding lines and *only* those words. Similarly, the final stanza must use *every* word from *all* the preceding stanzas and *only* those words.

While poetry is nice, we might also be intrigued by the idea of writing a program to check if a poem is a valid paradelle or not.

Perhaps: `is_paradelle: string -> bool`

Maybe return reasons that a string is not a paradelle instead of just `true` and `false`.

And more interestingly, can we do it using `map`, `filter`, and `fold` only, never writing a recursive function ourselves to solve this problem?

Yes, we can. This is homework 6.

The concepts of map, filter, and various folds are common in functional languages and their libraries.

However, the types given to these higher order functions varies in their implementations in OCaml's standard library and in the Jane Street Core libraries used in our book.

They are also different from those in Haskell.

We'll define out own implementations and later compare them to some standard library implementations.

Map

It is common to need to apply a function to every individual element of a list, returning a list with the results of those applications.

For example,

```
let inc x = x + 1
let r1 = map inc [1;2;3;4;5]

let r2 = map int_of_char [ 'a', '^', '4' ]

let r3 = map Char.lowercase [
   'H'; 'e'; 'l'; 'l'; 'o'; ' '; 'W'; 'O'; 'R'; 'L'; 'D
```

See examples in the utop-histories of use of `map` in `map.ml`.

Exercise L10, #1.

What is the type of `map`?

Recall our examples:

```
map inc [1;2;3;4]
```

or

```
map int_of_char [ 'a', '^', '4' ]
```

Exercise L10, #2.

What is the OCaml implementation of `map`?

Recall our examples:

```
map inc [1;2;3;4]
```

or

```
map int_of_char [ 'a', '^', '4' ]
```

Parametric polymorphism

The importance of parametric polymorphism is hard to understate here:

The type of `map` is
`('a -> 'b) -> 'a list -> 'b list`.

Without this kind of polymorphism, we would be left writing individual functions for each type:

- `map_int_int: (int -> int) -> int list -> int list`
- `map_int_char: (int -> char) -> int list -> char list`
- ...

Lambda expressions are commonly used with applications of `map`.
Why write

```
let inc x = x + 1

  ...  map inc [1;2;3;4;5] ...
```

when you could just write

```
...  map (fun x -> x + 1) [1;2;3;4;5] ...
```

over strings

There are a number of simple examples of higher order functions that work over strings, when strings are lists of characters.

But the OCaml type `string` is a built-in type.

We'll define our own string type:

```
type estring = char list
```

Some sample functions over strings:

- `get_excited : estring -> estring`
  Convert all periods to exclamation marks (bangs) !

- `chill : estring -> estring`
  Convert bangs to periods.

- `freshman: estring -> estring`
  Convert all periods and bangs to question marks.

See examples in `estrings.ml` in the code examples directory of the pubic repository.

Filtering elements from a list

It is also common to filter some elements from a list.

```
let even n = n mod 2 = 0
let evens = filter even [1;2;3;4;5;6;7]

let positive x = x > 0.0
let pos_nums = filter positive
                 [1.2; 3.4; -5.6; -7.8; 9.0]

let is_blank_or_tab ch = ch = ' ' || ch = '\t'
let ws = filter is_blank_or_tab
           (string_to_estring "a b\t c d")
```

See examples in filter.ml

Exercise L10, #3.

What is the type of `filter`?

Recall our examples:

```
filter even [1;2;3;4;5;6;7]
```

or

```
filter positive [1.2; 3.4; -5.6; -7.8; 9.0 ]
```

Exercise L10, #4.

What is OCaml implementation of `filter`?

Recall our examples:

```
filter even [1;2;3;4;5;6;7]
```

or

```
filter positive [1.2; 3.4; -5.6; -7.8; 9.0 ]
```

- Let's consider filters over strings and revisit `estrings.ml`
- Perhaps a function, `smush`, that removes all whitespace.
- Or a function to remove all punctuation. We will choose to disregard punctuation in our paradelle program, so this might be useful.

Exercise L10, #5.

Write a function that returns its input `char list` after removing all upper case letters from it.

(Well, lets just consider `'A'`, `'B'`, `'C'`, and `'D'` to keep this simple.)

And do it without using an `if-then-else` expression. Use `match`.

And, of course, use `filter`.

Your solution should look something like the following:

```
let removeABCD cs =
  let notABCD = function
    | 'A' | 'B' | 'C' | 'D' -> false
    | _ -> true
  in
  filter notABCD cs
```

Folding lists

Another common idiom is to "fold" list elements up into a, typically, single value.

```
let a_sum = fold (+) 0 [1;2;3;4]

let sum xs = fold (+) 0 xs
```

Exercise L10, #6.

What is the type of `fold`?

Recall our example:

```
fold (+) 0 [1;2;3;4]
```

Exercise L10, #7.

What is OCaml implementation of `fold`?

Recall our example:
`fold (+) 0 [1;2;3;4]`
We can see this as
`1 + (2 + (3 + (4 + 0)))`

Exercise L10, #8.

What is OCaml implementation of `fold`

when we see
`fold (+) 0 [1;2;3;4]`
as
`((((0 + 1) + 2) + 3) + 4)`

Folding from the left or the right

Folding from the left, we first apply `f` to the first element `x` and the accumulator `accum` and this result is passed in as the accumulator for the next step.

```
let rec foldl f accum l = match l with
  | [] -> accum
  | x::xs -> foldl f (f accum x) xs
```

Folding from the right, we apply `f` to the first element `x` and the result of folding up the rest of the list.

```
let rec foldr f accum l = match l with
  | [] -> accum
  | x::xs -> f x (foldr f accum xs)
```

Some more examples

- `length: 'a list -> int`
- `and: bool list -> bool`, also `or`
- `max: int list -> int option`, also `min`
- `is_elem: 'a -> 'a list -> bool`
- `split_by: 'a list -> 'a list -> 'a list list`
- `lebowski: char list -> char list`
  replace all `'.'` with
  `[',';' '; 'd'; 'u'; 'd'; 'e'; '.']`

Let's write some of these, both as recursive functions and usign `foldl` or `foldr`.

We'll ask: Is `foldl` or `foldr` better for any of these? Why?

These are found in `fold.ml` in the code examples directory in the public course repository.

Seeing folds as loops

It may be helpful to initially think of imperative solutions and consider what state is updated each time through the loop.
In C:

```
sum = 0;
for (i=0; i<N; i++) {
  sum = sum + array[i];
}
```

The "accumulator" value in a fold is this state.

Of course, `i` is just an index so we don't see it in a functional implemantation using lists.

In Python we see a closer match since there is no index variable:

```
and_of = True
for b in array:
    and_of = and_of and b
```

Clearly this can be seen in any imperative language.

This may help get you started on writing folds if it isn't clear what the accumulator and folding functions should be.