

# Higher-Order Programming I: Map and Filter

**Principles of Programming Languages**  
**Lecture 8**

# Introduction

# Administrivia

Assignment 3 is due on Thursday by 11:59PM.

The midterm for this course is in two weeks (2/27).

# Objectives

Introduce the notion of `higher-order functions` and how they can help us write cleaner, more general code.

Examine two classic higher-order functions, `map` and `filter`.

# Keywords

higher-order functions

first-class values

functions as function parameters

the abstraction principle

map and filter

tail-recursive map and filter

# Practice Problem

*Implement a function **split\_sorted** which given*

***$l$  : a sorted  $\text{int}$  list***

***$i$  :  $\text{int}$***

*returns two sorted lists, one which has the elements of  **$l$**  which are at most  **$i$** , and the other which has the elements of  **$l$**  which are greater than  **$i$** .*

# Higher-Order Functions

# Higher-Order Programming



# Higher-Order Programming

In OCaml, functions are **first-class values**:

# Higher-Order Programming

In OCaml, functions are **first-class values**:

1. They can be given names with `let-definitions`.

# Higher-Order Programming

In OCaml, functions are **first-class values**:

1. They can be given names with `let-definitions`.
2. They can be `returned` by another function.

# Higher-Order Programming

In OCaml, functions are **first-class values**:

1. They can be given names with `let-definitions`.
2. They can be `returned` by another function.
3. They can be `passed as arguments` to another function.

# Higher-Order Programming

In OCaml, functions are **first-class values**:

1. They can be given names with `let-definitions`.
2. They can be `returned` by another function.
3. They can be `passed as arguments` to another function.

**Note.** Types are *not* first-class values.



# An Aside: Robin Popplestone

"He started a PhD at Manchester University before moving to Leeds University. His project was to develop a program for automated theorem proving, but he got caught up in **using the university computer to design a boat**. He built the boat and set sail for the University of Edinburgh, where he had been offered a research position. A storm hit while crossing the North Sea, and **the boat sank**. A widely believed story about Popplestone was that he never completed his PhD in mathematics because he **lost his thesis manuscript in the boat**, although Popplestone refused to corroborate this."



# Functions as Returned Values

```
# let f x y = x + y;;  
val f : int -> int -> int = <fun>  
# f 2;;  
- : int -> int = <fun>
```

# Functions as Returned Values

```
# let f x y = x + y;;  
val f : int -> int -> int = <fun>  
# f 2;;  
- : int -> int = <fun>
```

This is not interesting in OCaml...



# Functions as Returned Values

```
# let f x y = x + y;;  
val f : int -> (int -> int) = <fun>  
# f 2;;  
- : int -> int = <fun>
```

This is not interesting in OCaml...

# Functions as Returned Values

```
# let f x y = x + y;;  
val f : int -> (int -> int) = <fun>  
# f 2;;  
- : int -> int = <fun>
```

This is not interesting in OCaml...

Multi-argument functions in OCaml are really single-argument functions which **return functions**.

# Functions as Named Values

`let f x y = x + y`

**is shorthand for...**

`let f = fun x -> fun y -> x + y`

# Functions as Named Values

```
let f x y = x + y
```

**is shorthand for...**

```
let f = fun x -> fun y -> x + y
```

This is also not interesting in OCaml...

# Functions as Named Values

```
let f x y = x + y
```

**is shorthand for...**

```
let f = fun x -> fun y -> x + y
```

This is also not interesting in OCaml...

When we **let-define any function**, we're giving a (anonymous) function value a name.

# Functions as Named Values

```
let f x y = x + y
```

**is shorthand for...**

```
let f = fun x -> fun y -> x + y
```

anonymous function

This is also not interesting in OCaml...

When we **let-define any function**, we're giving a (anonymous) function value a name.

# Functions and parameters

```
# let apply f x = f x;;  
val apply : ('a -> 'b) -> 'a -> 'b = <fun>  
# apply add_five 10;;  
- : int = 15
```

# Functions and parameters

```
# let apply f x = f x;;  
val apply : ('a -> 'b) -> 'a -> 'b = <fun>  
# apply add_five 10;;  
- : int = 15
```

This is *very* interesting in OCaml...



# Functions and parameters

```
# let apply f x = f x;;  
val apply : ('a -> 'b) -> 'a -> 'b = <fun>  
# apply add_five 10;;  
- : int = 15
```

note the type

This is *very* interesting in OCaml...

# Functions and parameters

```
# let apply f x = f x;;  
val apply : ('a -> 'b) -> 'a -> 'b = <fun>  
# apply add_five 10;;  
- : int = 15
```

note the type

This is *very* interesting in OCaml...

This allows us to create new functions which are **parametrized** by old ones.

# Higher-Order Functions Elsewhere

$$\text{fun } f \rightarrow \frac{f(x)}{dx} \qquad \text{e.g.} \qquad x^2 \mapsto 2x$$

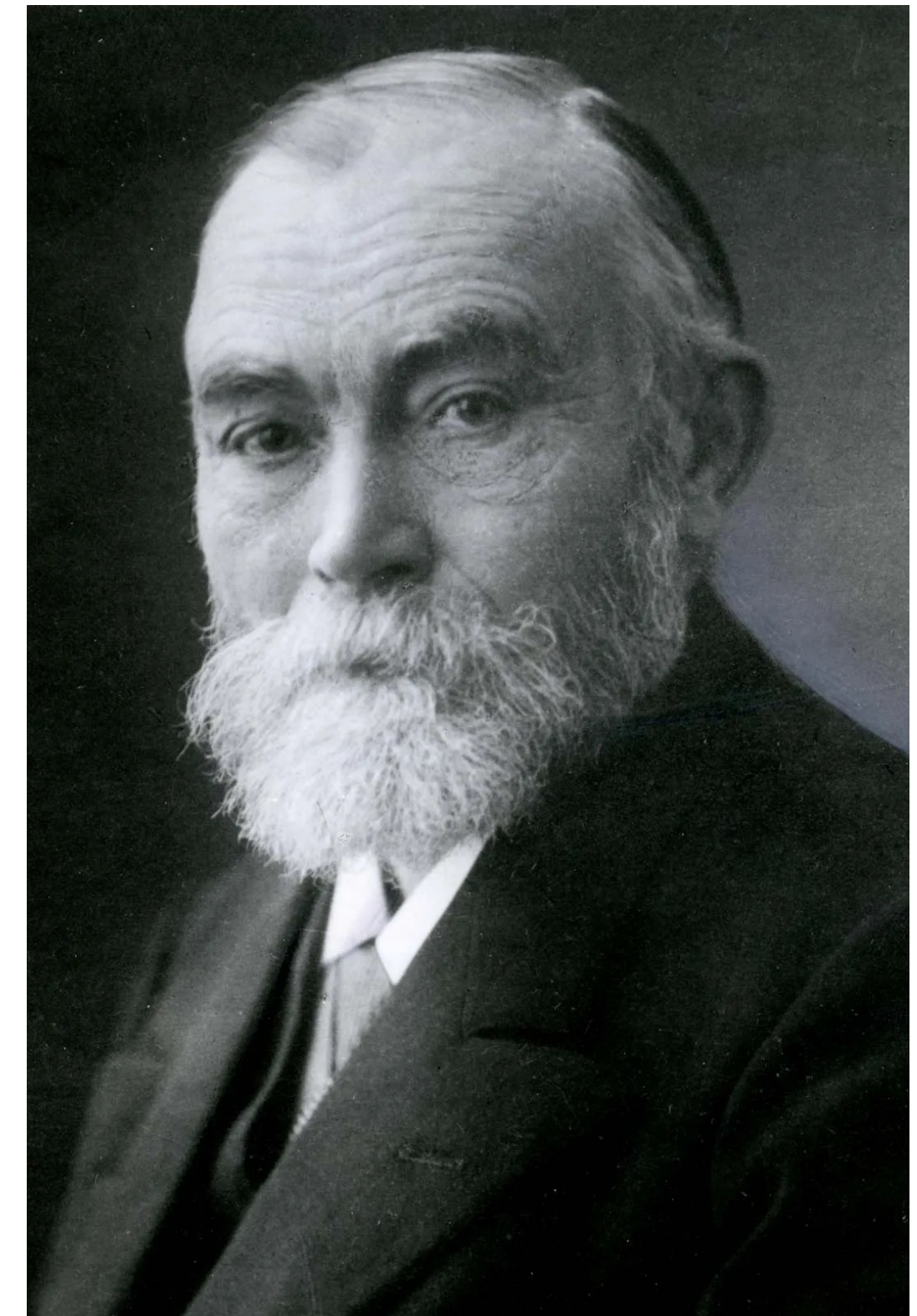
We might think of the type of an `derivative` as

$$(\mathbb{R} \rightarrow \mathbb{R}) \rightarrow \mathbb{R} \rightarrow \mathbb{R}$$

because it takes one function and produces a new function.

# An Aside: What does Higher-Order Mean?

"Like things and functions are different, so are functions whose **arguments are functions** radically different from functions whose **arguments must be things**. I call the latter functions of first order, the former functions of second order."



**Gottlob Frege**

# First-Order Function Types

`int -> string`

`t -> t`

`() -> bool`

`bool * bool -> bool`

# Second-Order Function Types

`(int -> string) -> (int -> string)`

`t -> (s -> t)`

`(( ) -> bool) -> bool`

`bool -> bool -> bool`

# Third-Order Function Types

`(int -> string) -> (int -> string) -> (int -> string)`

`(t -> (s -> t)) -> t`

`((()) -> bool) -> bool`

`(bool -> bool -> bool) * bool -> bool`

# And so on...

$t \rightarrow t \rightarrow t \rightarrow t \rightarrow t \rightarrow \dots$

The **higher-order** part comes from the fact that we can do this *ad infinitum*.

(In practice, we rarely use higher than third-order or fourth-order functions.)



# The Abstraction Principle

# Motivation

# Motivation

One of the three virtues of a great programmer  
is **laziness**.

# Motivation

One of the three virtues of a great programmer is **laziness**.

The **abstraction principle** helps use be lazy.

# Motivation

One of the three virtues of a great programmer is **laziness**.

The **abstraction principle** helps use be lazy.

When we write general programs, we **avoid rewriting programs** we've (pretty much) written before.

# A Simple Example

```
let rec reverse (l : int list) : int list =  
  match l with  
  | [] -> []  
  | x :: xs -> reverse xs @ [x]
```

# A Simple Example

```
let rec reverse (l : int list) : int list =  
  match l with  
  | [] -> []  
  | x :: xs -> reverse xs @ [x]
```

**Recall.** Polymorphism allows us to write general functions by being *agnostic* to the input type.

# A Simple Example

```
let rec reverse (l : 'a list) : 'a list =  
  match l with  
  | [] -> []  
  | x :: xs -> reverse xs @ [x]
```

**Recall.** Polymorphism allows us to write general functions by being *agnostic* to the input type.



# A Simple Example

```
let rec reverse (l : 'a list) : 'a list =  
  match l with  
  | [] -> []  
  | x :: xs -> reverse xs @ [x]
```

**Recall.** Polymorphism allows us to write general functions by being *agnostic* to the input type.

*It doesn't matter if we're reversing a list of **ints** or a list of **strings**, the function is the same.*

# A Simple Example

```
let rec fact n =  
  match n with  
  | 0 -> 1  
  | n -> n * fact (n - 1)
```

```
let rec sum n =  
  match n with  
  | 0 -> 0  
  | n -> n + sum (n - 1)
```

# A Simple Example

```
let rec fact n =  
  match n with  
  | 0 -> 1  
  | n -> n * fact (n - 1)
```

```
let rec sum n =  
  match n with  
  | 0 -> 0  
  | n -> n + sum (n - 1)
```

Some functions are cannot be polymorphic.

# A Simple Example

```
let rec fact n =  
  match n with  
  | 0 -> 1  
  | n -> n * fact (n - 1)
```

```
let rec sum n =  
  match n with  
  | 0 -> 0  
  | n -> n + sum (n - 1)
```

Some functions are cannot be polymorphic.

*But can we abstract the core functionality?*

# A Simple Example

```
let rec fact n =  
  match n with  
  | 0 -> 1  
  | n -> n * fact (n - 1)
```

```
let rec sum n =  
  match n with  
  | 0 -> 0  
  | n -> n + sum (n - 1)
```

Some functions are cannot be polymorphic.

*But can we abstract the core functionality?*

# A Simple Example

```
let rec upto f n start =  
  let rec go n =  
    match n with  
    | 0 -> start  
    | n -> f n (go (n - 1))  
  in go n
```

# A Simple Example

```
let rec upto f n start =  
  let rec go n =  
    match n with  
    | 0 -> start  
    | n -> f n (go (n - 1))  
  in go n
```

In order to generalize this function, we need to be able to take the **operation as a parameter**.

# A Simple Example

```
let rec upto f n start =  
  let rec go n =  
    match n with  
    | 0 -> start  
    | n -> f n (go (n - 1))  
  in go n
```

In order to generalize this function, we need to be able to take the **operation as a parameter**.



# A Simple Example

```
let rec upto f n start =  
  let rec go n =  
    match n with  
    | 0 -> start  
    | n -> f n (go (n - 1))  
  in go n
```

In order to generalize this function, we need to be able to take the **operation as a parameter**.

Now we have a single function which we can **reuse** elsewhere.

demo  
(fact\_sum.ml)

# Another Example: Sorting

```
let rec insert (x : 'a) (l : 'a list) : 'a list =  
  match l with  
  | [] -> [x]  
  | y :: ys -> if x <= y then x :: y :: ys else y :: insert x ys
```

```
let rec sort (l : 'a list) : 'a list =  
  match l with  
  | [] -> []  
  | x :: xs -> insert x (sort xs)
```

# Another Example: Sorting

```
let rec insert (x : 'a) (l : 'a list) : 'a list =  
  match l with  
  | [] -> [x]  
  | y :: ys -> if x <= y then x :: y :: ys else y :: insert x ys
```

```
let rec sort (l : 'a list) : 'a list =  
  match l with  
  | [] -> []  
  | x :: xs -> insert x (sort xs)
```

**Note.** Sorting *is* polymorphic.

# Another Example: Sorting

```
let rec insert (x : 'a) (l : 'a list) : 'a list =  
  match l with  
  | [] -> [x]  
  | y :: ys -> if x <= y then x :: y :: ys else y :: insert x ys  
  
let rec sort (l : 'a list) : 'a list =  
  match l with  
  | [] -> []  
  | x :: xs -> insert x (sort xs)
```

**Note.** Sorting *is* polymorphic.

But what if we want to sort in **reverse order**?

# Another Example: Sorting

```
let rec insert (x : 'a) (l : 'a list) : 'a list =  
  match l with  
  | [] -> [x]  
  | y :: ys -> if x >= y then x :: y :: ys else y :: insert x ys  
  
let rec sort (l : 'a list) : 'a list =  
  match l with  
  | [] -> []  
  | x :: xs -> insert x (sort xs)
```

**Note.** Sorting *is* polymorphic.

But what if we want to sort in reverse order?

# Another Example: Sorting

```
let rec insert (x : 'a) (l : 'a list) : 'a list =  
  match l with  
  | [] -> [x]  
  | y :: ys -> if x >= y then x :: y :: ys else y :: insert x ys  
  
let rec sort (l : 'a list) : 'a list =  
  match l with  
  | [] -> []  
  | x :: xs -> insert x (sort xs)
```

**Note.** Sorting *is* polymorphic.

But what if we want to sort in reverse order?

We shouldn't rewrite the whole function.

# Another Example: Sorting

```
let insert (le : 'a -> 'a -> bool) =  
  let rec go x l =  
    match l with  
    | [] -> [x]  
    | y :: ys ->  
      if le x y then  
        x :: y :: ys  
      else  
        y :: go x ys  
  in go  
  
let rec sort (le : 'a -> 'a -> bool) =  
  let rec go l =  
    match l with  
    | [] -> []  
    | x :: xs -> insert le x (go xs)  
  in go
```



# Another Example: Sorting

```
let insert (le : 'a -> 'a -> bool) =  
  let rec go x l =  
    match l with  
    | [] -> [x]  
    | y :: ys ->  
      if le x y then  
        x :: y :: ys  
      else  
        y :: go x ys  
  in go
```

```
let rec sort (le : 'a -> 'a -> bool) =  
  let rec go l =  
    match l with  
    | [] -> []  
    | x :: xs -> insert le x (go xs)  
  in go
```

Instead, we take a  
comparison function as  
a parameter.

# Another Example: Sorting

```
let insert (le : 'a -> 'a -> bool) =  
  let rec go x l =  
    match l with  
    | [] -> [x]  
    | y :: ys ->  
      if le x y then  
        x :: y :: ys  
      else  
        y :: go x ys  
  in go
```

```
let rec sort (le : 'a -> 'a -> bool) =  
  let rec go l =  
    match l with  
    | [] -> []  
    | x :: xs -> insert le x (go xs)  
  in go
```

Instead, we take a  
comparison function as  
a parameter.

# Another Example: Sorting

```
let insert (le : 'a -> 'a -> bool) =  
  let rec go x l =  
    match l with  
    | [] -> [x]  
    | y :: ys ->  
      if le x y then  
        x :: y :: ys  
      else  
        y :: go x ys  
  in go
```

```
let rec sort (le : 'a -> 'a -> bool) =  
  let rec go l =  
    match l with  
    | [] -> []  
    | x :: xs -> insert le x (go xs)  
  in go
```

Instead, we take a  
comparison function as  
a parameter.

Otherwise, the code is  
pretty much identical.

# Another Example: Sorting

```
let insert (le : 'a -> 'a -> bool) =  
  let rec go x l =  
    match l with  
    | [] -> [x]  
    | y :: ys ->  
      if le x y then  
        x :: y :: ys  
      else  
        y :: go x ys  
  in go
```

```
let rec sort (le : 'a -> 'a -> bool) =  
  let rec go l =  
    match l with  
    | [] -> []  
    | x :: xs -> insert le x (go xs)  
  in go
```

Instead, we take a  
comparison function as  
a parameter.

Otherwise, the code is  
pretty much identical.

**An Aside.** Also note the  
arguments. Why doesn't  
sort take a list?

# The Abstraction Principle

# The Abstraction Principle

As noted in the text, the abstraction principle comes from MacLennan's ***Functional Programming: Theory and Practice.***

# The Abstraction Principle

As noted in the text, the abstraction principle comes from MacLennan's ***Functional Programming: Theory and Practice.***

It's not a concrete principle. Roughly it says:

# The Abstraction Principle

As noted in the text, the abstraction principle comes from MacLennan's ***Functional Programming: Theory and Practice***.

It's not a concrete principle. Roughly it says:

- Abstract out **core functionality**.



# The Abstraction Principle

As noted in the text, the abstraction principle comes from MacLennan's ***Functional Programming: Theory and Practice***.

It's not a concrete principle. Roughly it says:

- Abstract out **core functionality**.
- Use higher-order functions to **parametrize** by functionality specific to the problem.

# The Abstraction Principle

As noted in the text, the abstraction principle comes from MacLennan's ***Functional Programming: Theory and Practice***.

It's not a concrete principle. Roughly it says:

- Abstract out **core functionality**.
- Use higher-order functions to **parametrize** by functionality specific to the problem.
- (Try to understand the **algebra** of programming.)

# Understanding Check

```
let rec next_word (cs : char list) =  
  match cs with  
  | [] -> []  
  | x :: xs ->  
    if (not (x = ' ')) then x :: next_word xs else []
```

```
let rec pos_prefix (l : int list) : int list =  
  match l with  
  | [] -> []  
  | x :: xs ->  
    if x > 0 then x :: pos_prefix xs else []
```

*Write a function which factors out the core functionality of the above two functions.*

# Map

# Simple Example: Update All Users

```
type user = {  
  name : string ;  
  id : int ;  
}
```

```
let capitalize = ...
```

```
let fix_usernames (us : user list) =  
  List.map (fun u -> { u with name = capitalize u.name }) us
```

**map** is used to apply a function to every element in a list (or other structure).

# Definition of Map

```
let rec map f l =  
  match l with  
  | [] -> []  
  | x :: xs -> f x :: map f xs
```

# Definition of Map

```
let rec map f l =  
  match l with  
  | [] -> []  
  | x :: xs -> f x :: map f xs
```

*If the list is empty there is nothing to do.*

# Definition of Map

```
let rec map f l =  
  match l with  
  | [] -> []  
  | x :: xs -> f x :: map f xs
```

*If the list is empty there is nothing to do.*

*If the list is nonempty, we apply  $f$  to its first element, and recurse.*



# Tail-Recursive Map

```
let rec map_t f l =  
  let rec go l acc =  
    match l with  
    | [] -> List.rev acc  
    | x :: xs -> go xs (f x :: acc)  
  in go l []
```

# Tail-Recursive Map

```
let rec map_t f l =  
  let rec go l acc =  
    match l with  
    | [] -> List.rev acc  
    | x :: xs -> go xs (f x :: acc)  
  in go l []
```

For a tail-recursive version we can build the list in reverse in **acc** and then **reverse it at the end**.

# Tail-Recursive Map

```
let rec map_t f l =  
  let rec go l acc =  
    match l with  
    | [] -> List.rev acc  
    | x :: xs -> go xs (f x :: acc)  
  in go l []
```

For a tail-recursive version we can build the list in reverse in **acc** and then **reverse it at the end**.

# Tail-Recursive Map

```
let rec map_t f l =  
  let rec go l acc =  
    match l with  
    | [] -> List.rev acc  
    | x :: xs -> go xs (f x :: acc)  
  in go l []
```

For a tail-recursive version we can build the list in reverse in **acc** and then **reverse it at the end**.

This may seem inefficient, but its just a **constant factor** slower.

# Tail-Recursive Map

```
let rec map_t f l =  
  let rec go l acc =  
    match l with  
    | [] -> List.rev acc  
    | x :: xs -> go xs (f x :: acc)  
  in go l []
```

For a tail-recursive version we can build the list in reverse in **acc** and then **reverse it at the end**.

This may seem inefficient, but its just a **constant factor** slower.

**An Aside.** The standard library map is *not* tail-recursive.

# Additional Notes

# Additional Notes

The text mentions two additional things about **map**:

# Additional Notes

The text mentions two additional things about **map**:

1. There is a function **rev\_map**, which is tail-recursive and does give the output in reverse order.



# Additional Notes

The text mentions two additional things about **map**:

1. There is a function **rev\_map**, which is tail-recursive and does give the output in reverse order.
2. **map** is defined somewhat differently to account for **side-effects**.

# Additional Notes

The text mentions two additional things about **map**:

1. There is a function **rev\_map**, which is tail-recursive and does give the output in reverse order.
2. **map** is defined somewhat differently to account for **side-effects**.

We won't dwell on these for now, but it may be worth reading about.

demo  
(normalize.ml)

# Understanding Check

*Using a single call to map, implement function **pointwise\_max** which, given*

*$f : 'a \rightarrow int$   
 $g : 'a \rightarrow int$   
 $l : 'a\ list$*

*returns  $l$  with  $f$  or  $g$  applied to each element, whichever gives a larger value.*

**Filter**

# Simple Example:

```
type user = {  
  name : string ;  
  id : int ;  
  num_likes : int ;  
}
```

```
let popular (us : user list) (cap : int) =  
  List.filter (fun u -> u.num_likes > cap) us
```

**filter** is used to grab all elements in a list which **satisfy a given property**.

# Predicates

# Predicates

A Boolean predicate on 'a is a function of type

`'a -> bool`



# Predicates

A Boolean predicate on 'a is a function of type

`'a -> bool`

A predicate is a function which represents a  
property.

# Predicates

A Boolean predicate on 'a is a function of type

`'a -> bool`

A predicate is a function which represents a **property**.

Example. `let even n = n mod 2 = 0`

# Definition of Filter

```
let rec filter p l =  
  match l with  
  | [] -> []  
  | x :: xs ->  
    (if p x then [x] else []) @ filter p xs
```

*If the list is empty there is nothing to do.*

*If the first element satisfies our predicate we keep it and recurse.*

*Otherwise, we drop it and recurse.*

# Tail Recursive Filter

```
let filter_tail p =  
  let rec go acc l =  
    match l with  
    | [] -> List.rev acc  
    | x :: xs -> go ((if p x then [x] else []) @ acc) xs  
  in go []
```

# Tail Recursive Filter

```
let filter_tail p =  
  let rec go acc l =  
    match l with  
    | [] -> List.rev acc  
    | x :: xs -> go ((if p x then [x] else []) @ acc) xs  
  in go []
```

As with map, we have to **reverse** the output before returning it.

# Tail Recursive Filter

```
let filter_tail p =  
  let rec go acc l =  
    match l with  
    | [] -> List.rev acc  
    | x :: xs -> go ((if p x then [x] else []) @ acc) xs  
  in go []
```

As with map, we have to **reverse** the output before returning it.

The standard library implementation of **filter** *is* tail-recursive.

demo  
(primes.ml)

# Understanding Check

What do the following functions do?

```
let f = List.filter (fun i -> i mod 2 <> 0)
```

```
let g l = l |> List.map (fun x -> x * x) |> f
```

```
let h p q = List.filter (fun i -> p i && q i)
```



# Summary

Higher-order function allow for better `abstraction` because we can parameterize functions by other functions.

`map` and `filter` are very common patterns which can be used to write clean and simple code.