CSC 330 / Spring 2026

OCaml
First-class Functions

Ibrahim Numanagić /nu-ma-nagg-ich/

based on Dan Grossman's lecture materials

Example

```
let double x = x * 2 (* old-fashioned function *)
let incr   x = x + 1 (* old-fashioned function *)

let funcs = [ double; incr ] (* list of functions *)

let rec apply_funcs (fs, x) =
  match fs with
  | [] -> x
  | f :: fs' -> apply_funcs (fs', f x)

let foo = apply_funcs (funcs, 100)           (* 201 *)
let bar = apply_funcs (List.rev funcs, 100)  (* 202 *)
```

Demonstration: First-class functions

Demonstration: First-class functions

```
                Welcome to utop version 2.16.0 (using OCaml version 5.4.0)!

# let double x = x * 2
  let incr x = x + 1
  let funcs = [ double; incr ]

  let rec apply_funcs (fs, x) =
    match fs with
    | [] -> x
    | f :: fs' -> apply_funcs (fs', f x)

  (* binds foo to 201 *)
  let foo = apply_funcs (funcs, 100)

  (* binds bar to 202 *)
  let bar = apply_funcs (List.rev funcs, 100) ;;
```

Demonstration: First-class functions

```
                Welcome to utop version 2.16.0 (using OCaml version 5.4.0)!

# let double x = x * 2
  let incr x = x + 1
  let funcs = [ double; incr ]

  let rec apply_funcs (fs, x) =
    match fs with
    | [] -> x
    | f :: fs' -> apply_funcs (fs', f x)

  (* binds foo to 201 *)
  let foo = apply_funcs (funcs, 100)

  (* binds bar to 202 *)
  let bar = apply_funcs (List.rev funcs, 100) ;;

  val double : int -> int = <fun>
  val incr : int -> int = <fun>
  val funcs : (int -> int) list = [<fun>; <fun>]
  val apply_funcs : ('a -> 'a) list * 'a -> 'a = <fun>
  val foo : int = 201
  val bar : int = 202
```

Demonstration: First-class functions

```
                Welcome to utop version 2.16.0 (using OCaml version 5.4.0)!

# let double x = x * 2
  let incr x = x + 1
  let funcs = [ double; incr ]

  let rec apply_funcs (fs, x) =
    match fs with
    | [] -> x
    | f :: fs' -> apply_funcs (fs', f x)

  (* binds foo to 201 *)
  let foo = apply_funcs (funcs, 100)

  (* binds bar to 202 *)
  let bar = apply_funcs (List.rev funcs, 100) ;;
val double : int -> int = <fun>
val incr : int -> int = <fun>
val funcs : (int -> int) list = [<fun>; <fun>]
val apply_funcs : ('a -> 'a) list * 'a -> 'a = <fun>
val foo : int = 201
val bar : int = 202
# (* It should really bother us to write these functions separately... *)
  let rec increment_n_times_bothersome (n, x) =
    if n = 0 then x else 1 + increment_n_times_bothersome (n - 1, x)

  let rec double_n_times_bothersome (n, x) =
    if n = 0 then x else 2 * double_n_times_bothersome (n - 1, x)

  let rec tail_n_times_bothersome (n, xs) =
    if n = 0 then xs else List.tl (tail_n_times_bothersome (n - 1, xs)) ;;
```

Demonstration: First-class functions

```
                Welcome to utop version 2.16.0 (using OCaml version 5.4.0)!

# let double x = x * 2
  let incr x = x + 1
  let funcs = [ double; incr ]

  let rec apply_funcs (fs, x) =
    match fs with
    | [] -> x
    | f :: fs' -> apply_funcs (fs', f x)

  (* binds foo to 201 *)
  let foo = apply_funcs (funcs, 100)

  (* binds bar to 202 *)
  let bar = apply_funcs (List.rev funcs, 100) ;;

  val double : int -> int = <fun>
  val incr : int -> int = <fun>
  val funcs : (int -> int) list = [<fun>; <fun>]
  val apply_funcs : ('a -> 'a) list * 'a -> 'a = <fun>
  val foo : int = 201
  val bar : int = 202

# (* It should really bother us to write these functions separately... *)
  let rec increment_n_times_bothersome (n, x) =
    if n = 0 then x else 1 + increment_n_times_bothersome (n - 1, x)

  let rec double_n_times_bothersome (n, x) =
    if n = 0 then x else 2 * double_n_times_bothersome (n - 1, x)

  let rec tail_n_times_bothersome (n, xs) =
    if n = 0 then xs else List.tl (tail_n_times_bothersome (n - 1, xs)) ;;

  val increment_n_times_bothersome : int * int -> int = <fun>
  val double_n_times_bothersome : int * int -> int = <fun>
  val tail_n_times_bothersome : int * 'a list -> 'a list = <fun>
```

What is *functional programming*?

- No crisp definition (you get to know it when you see it), but usually includes:
  - Avoiding mutation in most (or all) cases
  - Using functions as values: the topic of today's lecture
- Definition may also include:
  - Using recursion and recursive variant types
  - Style closer to mathematical definitions
  - Idioms using "laziness" (later topic, briefly)
  - (*bad definition*) Anything not OOP or C?
- A *functional language* just *encourages* functional programming
  - Often not a clear yes or no

First-class functions

- **First-class** means something is *in the language* of expressions:
  you can compute them, pass them around, put them in data structures, etc.
- Functions in many languages, including OCaml, are first class
  - I never said they weren't
  - OCaml functions are *almost* values
- Most common use is as an argument to another function
  - Lets us abstract over *what to compute* in certain situations: caller just passes in code to handle!
  - A function that takes or returns functions is called higher-order

Function closures

- Functions can use bindings from the enclosing environment where they were defined
  - Even if function is passed around and called somewhere else
  - Makes first-class functions *much* more powerful and useful
- Will study this carefully after some simpler examples
  - Will need function *values* to be not just functions, but also the environments where they were defined
  - *A function* **and** *its environment* is called a **function closure**
- In theory, a language could have first-class functions without closures or vice versa, but typically a language with one has the other

Plan for this lecture

1. How to use first-class functions and function closures?
2. The precise semantics for function closures and function calls
3. Multiple powerful idioms this semantics enables

Functions as arguments

- Can pass one function as an argument to another function
  - Again, not really a new feature: we just haven't done this before
- Elegant way to factor out common code
  - Replace *n* similar functions with calls to 1 function with *n* different (short) function arguments

```
let rec n_times (f, n, x) =
  if n = 0 then x else f (n_times (f, n - 1, x))
```

Types for `n_times`

- What's the type of `n_times`?

```
let rec n_times (f, n, x) =
  if n = 0
  then x
  else f (n_times (f, n - 1, x))
```

  - It's `val n_times : ('a -> 'a) * int * 'a -> 'a`
    - Types are *inferred* based on how arguments are used!
    - More useful than `(int -> int) * int * int -> int`

Relation to types

- Higher-order functions often "so reusable" that they have polymorphic types
  - i.e., types with **type variables** (e.g., `'a`)
- There are higher-order functions that are not polymorphic
  - e.g., `val times_until_zero : (int -> int) * int -> int`
- And there are polymorphic functions that are not higher-order
  - e.g., `val len : 'a list -> int`
- Always a good idea to understand a function's type, especially for higher-order functions

Demonstration: First-class functions

```
Welcome to utop version 2.16.0 (using OCaml version 5.4.0)!
```

Demonstration: First-class functions

```
Welcome to utop version 2.16.0 (using OCaml version 5.4.0)!

# (* Much better now: abstract the common pieces into a function *)
  let rec n_times (f, n, x) =
    if n = 0 then x else f (n_times (f, n - 1, x))

  (* we can now implement the 3 bothersome functions above in one line each *)
  let increment_n_times (n, x) = n_times (incr, n, x)
  let double_n_times (n, x) = n_times (double, n, x)
  let tail_n_times (n, xs) = n_times (List.tl, n, xs)

  (* and nothing stops us from using n_times in places we didn't originally plan *)
  let triple x = 3 * x
  let triple_n_times (n, x) = n_times (triple, n, x) ;;
```

Demonstration: First-class functions

```
Welcome to utop version 2.16.0 (using OCaml version 5.4.0)!

# (* Much better now: abstract the common pieces into a function *)
  let rec n_times (f, n, x) =
    if n = 0 then x else f (n_times (f, n - 1, x))

  (* we can now implement the 3 bothersome functions above in one line each *)
  let increment_n_times (n, x) = n_times (incr, n, x)
  let double_n_times (n, x) = n_times (double, n, x)
  let tail_n_times (n, xs) = n_times (List.tl, n, xs)

  (* and nothing stops us from using n_times in places we didn't originally plan *)
  let triple x = 3 * x
  let triple_n_times (n, x) = n_times (triple, n, x) ;;

  val n_times : ('a -> 'a) * int * 'a -> 'a = <fun>
  val increment_n_times : int * int -> int = <fun>
  val double_n_times : int * int -> int = <fun>
  val tail_n_times : int * 'a list -> 'a list = <fun>
  val triple : int -> int = <fun>
  val triple_n_times : int * int -> int = <fun>
```

Demonstration: First-class functions

```
Welcome to utop version 2.16.0 (using OCaml version 5.4.0)!

# (* Much better now: abstract the common pieces into a function *)
  let rec n_times (f, n, x) =
    if n = 0 then x else f (n_times (f, n - 1, x))

  (* we can now implement the 3 bothersome functions above in one line each *)
  let increment_n_times (n, x) = n_times (incr, n, x)
  let double_n_times (n, x) = n_times (double, n, x)
  let tail_n_times (n, xs) = n_times (List.tl, n, xs)

  (* and nothing stops us from using n_times in places we didn't originally plan *)
  let triple x = 3 * x
  let triple_n_times (n, x) = n_times (triple, n, x) ;;

  val n_times : ('a -> 'a) * int * 'a -> 'a = <fun>
  val increment_n_times : int * int -> int = <fun>
  val double_n_times : int * int -> int = <fun>
  val tail_n_times : int * 'a list -> 'a list = <fun>
  val triple : int -> int = <fun>
  val triple_n_times : int * int -> int = <fun>

# (* Polymorphic but not higher order *)
  let rec len xs =
    match xs with
    | [] -> 0
    | _ :: xs' -> 1 + len xs'

  let r = len [ 1; 2 ]
  let r = len [ "a"; "b" ]

  (* Higher order but not polymorphic *)
  let rec times_until_zero (f, x) =
    if x = 0 then 0 else 1 + times_until_zero (f, f x) ;;

  let f x = x - 2 in times_until_zero (f, 10) ;;
```

Demonstration: First-class functions

```
Welcome to utop version 2.16.0 (using OCaml version 5.4.0)!

# (* Much better now: abstract the common pieces into a function *)
  let rec n_times (f, n, x) =
    if n = 0 then x else f (n_times (f, n - 1, x))

  (* we can now implement the 3 bothersome functions above in one line each *)
  let increment_n_times (n, x) = n_times (incr, n, x)
  let double_n_times (n, x) = n_times (double, n, x)
  let tail_n_times (n, xs) = n_times (List.tl, n, xs)

  (* and nothing stops us from using n_times in places we didn't originally plan *)
  let triple x = 3 * x
  let triple_n_times (n, x) = n_times (triple, n, x) ;;

  val n_times : ('a -> 'a) * int * 'a -> 'a = <fun>
  val increment_n_times : int * int -> int = <fun>
  val double_n_times : int * int -> int = <fun>
  val tail_n_times : int * 'a list -> 'a list = <fun>
  val triple : int -> int = <fun>
  val triple_n_times : int * int -> int = <fun>

# (* Polymorphic but not higher order *)
  let rec len xs =
    match xs with
    | [] -> 0
    | _ :: xs' -> 1 + len xs'

  let r = len [ 1; 2 ]
  let r = len [ "a"; "b" ]

  (* Higher order but not polymorphic *)
  let rec times_until_zero (f, x) =
    if x = 0 then 0 else 1 + times_until_zero (f, f x) ;;

  let f x = x - 2 in times_until_zero (f, 10) ;;

  val len : 'a list -> int = <fun>
  val r : int = 2
  val times_until_zero : (int -> int) * int -> int = <fun>
  - : int = 5
```

Demonstration: First-class functions

```
Welcome to utop version 2.16.0 (using OCaml version 5.4.0)!
```

```
# (* Much better now: abstract the common pieces into a function *)
  let rec n_times (f, n, x) =
    if n = 0 then x else f (n_times (f, n - 1, x))

  (* we can now implement the 3 bothersome functions above in one line each *)
  let increment_n_times (n, x) = n_times (incr, n, x)
  let double_n_times (n, x) = n_times (double, n, x)
  let tail_n_times (n, xs) = n_times (List.tl, n, xs)

  (* and nothing stops us from using n_times in places we didn't originally plan *)
  let triple x = 3 * x
  let triple_n_times (n, x) = n_times (triple, n, x) ;;

  val n_times : ('a -> 'a) * int * 'a -> 'a = <fun>
  val increment_n_times : int * int -> int = <fun>
  val double_n_times : int * int -> int = <fun>
  val tail_n_times : int * 'a list -> 'a list = <fun>
  val triple : int -> int = <fun>
  val triple_n_times : int * int -> int = <fun>

# (* Polymorphic but not higher order *)
  let rec len xs =
    match xs with
    | [] -> 0
    | _ :: xs' -> 1 + len xs'

  let r = len [ 1; 2 ]
  let r = len [ "a"; "b" ]

  (* Higher order but not polymorphic *)
  let rec times_until_zero (f, x) =
    if x = 0 then 0 else 1 + times_until_zero (f, f x) ;;

  let f x = x - 2 in times_until_zero (f, 10) ;;

  val len : 'a list -> int = <fun>
  val r : int = 2
  val times_until_zero : (int -> int) * int -> int = <fun>
  - : int = 5

# (* ⚠️ This fails! *)
  let f x = x ^ "1"
  let r = times_until_zero (f, 10) ;;
```

Demonstration: First-class functions

```
Welcome to utop version 2.16.0 (using OCaml version 5.4.0)!
```

```
# (* Much better now: abstract the common pieces into a function *)
  let rec n_times (f, n, x) =
    if n = 0 then x else f (n_times (f, n - 1, x))

  (* we can now implement the 3 bothersome functions above in one line each *)
  let increment_n_times (n, x) = n_times (incr, n, x)
  let double_n_times (n, x) = n_times (double, n, x)
  let tail_n_times (n, xs) = n_times (List.tl, n, xs)

  (* and nothing stops us from using n_times in places we didn't originally plan *)
  let triple x = 3 * x
  let triple_n_times (n, x) = n_times (triple, n, x) ;;

  val n_times : ('a -> 'a) * int * 'a -> 'a = <fun>
  val increment_n_times : int * int -> int = <fun>
  val double_n_times : int * int -> int = <fun>
  val tail_n_times : int * 'a list -> 'a list = <fun>
  val triple : int -> int = <fun>
  val triple_n_times : int * int -> int = <fun>

# (* Polymorphic but not higher order *)
  let rec len xs =
    match xs with
    | [] -> 0
    | _ :: xs' -> 1 + len xs'

  let r = len [ 1; 2 ]
  let r = len [ "a"; "b" ]

  (* Higher order but not polymorphic *)
  let rec times_until_zero (f, x) =
    if x = 0 then 0 else 1 + times_until_zero (f, f x) ;;

  let f x = x - 2 in times_until_zero (f, 10) ;;

  val len : 'a list -> int = <fun>
  val r : int = 2
  val times_until_zero : (int -> int) * int -> int = <fun>
  - : int = 5

# (* ⚠️ This fails! *)
  let f x = x ^ "1"
  let r = times_until_zero (f, 10) ;;

  Error: The value f has type string -> string
         but an expression was expected of type int -> int
         Type string is not compatible with type int
```

Map

```
let rec map (f, xs) =
  match xs with
  | [] -> []
  | x :: xs' -> f x :: map (f, xs')
(* ('a -> 'b) * 'a list -> 'b list *)
```

- Hall-of-fame higher-order function
- Name is standard (used for any sort of collection data structure)
  - Generally provided in standard libraries for standard collections
- Used all the time in functional code, thus *idiomatic*
  - Using it where appropriate isn't just shorter code but *communicates what you are doing*
  - Conversely, not using it when "doing a map" confuses your reader

Filter

```
let rec filter f xs =
  match xs with
  | [] -> []
  | x :: xs' -> if f x
                then x :: filter f xs'
                else filter f xs'
(* ('a -> bool) * 'a list -> 'a list *)
```

- Another hall-of-famer
  - Keep only some elements from a collection
  - Again an idiom, so you should use whenever “doing a filter”

Anonymous functions

Anonymous functions

- **Syntax:**

  `fun p -> e`

  where `p` is a pattern and `e` is an expression

- Function as an expression!
  - Type checking, evaluation rules "the same" as function bindings
  - Note that use `->` instead of `=` to separate parameters from function body
  - No function name!
  - *Just an expression*: can appear anywhere expressions allowed!
- You've probably seen these before
  - Hint: Python's `lambda a: e`

Using anonymous functions

- Usually use anonymous functions as arguments to other functions
  - No need to name a function that you use only once
- **Limitation**: anonymous functions cannot call themselves
  - No name to make a recursive call with
- Non-recursive function bindings are really just syntactic sugar
  - These two lines are equivalent!

```
let f p = e
let f = fun p -> e
```

  - Again, doesn't work for recursive functions
  - And as usual, better style to use the syntactic sugar available

A point on style

BAD

```
if e then true else false
fun x -> f x
n_times ((fun x -> List.tl x), 3, xs)
```

GOOD

```
e
f
n_times (List.tl, 3, xs)
```

Generalizing

- Our examples so far have been awfully similar
  - Take one function as argument then process a number or list
- We can do much more!
  - Pass several functions as arguments
  - Put functions in data structures
  - Return functions as results
  - Write higher-order functions over variant types
- Useful whenever abstracting over “what to compute with”

Returning functions

- Functions are first-class, so they can be returned from (other) functions
- Silly example:

```
let double_or_triple f =
  if f 7 then fun x -> 2 * x else fun x -> 3 * x
```

  - Has type `(int -> bool) -> (int -> int)`
  - But REPL reports `(int -> bool) -> int -> int`!
  - Well, this is **the same thing**:
    - REPL never prints unnecessary parentheses
    - `t1 -> t2 -> t3 -> t4` means `t1 -> (t2 -> (t3 -> t4))`
    - We will learn soon why this *precedence* is convenient

Other data structures

- Higher-order functions are not just for lists
- They work great for common traversals over your own data structures
- See the code demonstration for an example

Demonstration: Anonymous functions

```
Welcome to utop version 2.16.0 (using OCaml version 5.4.0)!
```

Demonstration: Anonymous functions

```
                Welcome to utop version 2.16.0 (using OCaml version 5.4.0)!

# (* Two very useful and very common higher-order functions *)
  let rec map (f, xs) =
    match xs with [] -> [] | x :: xs' -> f x :: map (f, xs')

  let rec filter (f, xs) =
    match xs with
    | [] -> []
    | x :: xs' ->
      if f x then x :: filter (f, xs') else filter (f, xs') ;;
```

Demonstration: Anonymous functions

```
Welcome to utop version 2.16.0 (using OCaml version 5.4.0)!

# (* Two very useful and very common higher-order functions *)
  let rec map (f, xs) =
    match xs with [] -> [] | x :: xs' -> f x :: map (f, xs')

  let rec filter (f, xs) =
    match xs with
    | [] -> []
    | x :: xs' ->
      if f x then x :: filter (f, xs') else filter (f, xs') ;;
val map : ('a -> 'b) * 'a list -> 'b list = <fun>
val filter : ('a -> bool) * 'a list -> 'a list = <fun>
```

Demonstration: Anonymous functions

```
Welcome to utop version 2.16.0 (using OCaml version 5.4.0)!

# (* Two very useful and very common higher-order functions *)
  let rec map (f, xs) =
    match xs with [] -> [] | x :: xs' -> f x :: map (f, xs')

  let rec filter (f, xs) =
    match xs with
    | [] -> []
    | x :: xs' ->
      if f x then x :: filter (f, xs') else filter (f, xs') ;;
val map : ('a -> 'b) * 'a list -> 'b list = <fun>
val filter : ('a -> bool) * 'a list -> 'a list = <fun>

# (* Motivating anonymous functions *)
  (* If we just need is_even for only_evens, better to use local function *)
  let only_evens2 xs =
    let is_even x = x mod 2 = 0 in
    filter (is_even, xs)

  (* Actually, we could define it *right* where we need it *)
  let only_evens3 xs =
    filter ((let is_even x = x mod 2 = 0 in is_even), xs)

  (* Seems kind of silly to have a let expression whose body is the variable...
     Can we do better? *) ;;
```

Demonstration: Anonymous functions

```
Welcome to utop version 2.16.0 (using OCaml version 5.4.0)!

# (* Two very useful and very common higher-order functions *)
  let rec map (f, xs) =
    match xs with [] -> [] | x :: xs' -> f x :: map (f, xs')

  let rec filter (f, xs) =
    match xs with
    | [] -> []
    | x :: xs' ->
      if f x then x :: filter (f, xs') else filter (f, xs') ;;

  val map : ('a -> 'b) * 'a list -> 'b list = <fun>
  val filter : ('a -> bool) * 'a list -> 'a list = <fun>

# (* Motivating anonymous functions *)
  (* If we just need is_even for only_evens, better to use local function *)
  let only_evens2 xs =
    let is_even x = x mod 2 = 0 in
    filter (is_even, xs)

  (* Actually, we could define it *right* where we need it *)
  let only_evens3 xs =
    filter ((let is_even x = x mod 2 = 0 in is_even), xs)

  (* Seems kind of silly to have a let expression whose body is the variable...
     Can we do better? *) ;;

  val only_evens2 : int list -> int list = <fun>
  val only_evens3 : int list -> int list = <fun>
```

Demonstration: Anonymous functions

```
Welcome to utop version 2.16.0 (using OCaml version 5.4.0)!

# (* Two very useful and very common higher-order functions *)
  let rec map (f, xs) =
    match xs with [] -> [] | x :: xs' -> f x :: map (f, xs')

  let rec filter (f, xs) =
    match xs with
    | [] -> []
    | x :: xs' ->
      if f x then x :: filter (f, xs') else filter (f, xs') ;;
val map : ('a -> 'b) * 'a list -> 'b list = <fun>
val filter : ('a -> bool) * 'a list -> 'a list = <fun>

# (* Motivating anonymous functions *)
  (* If we just need is_even for only_evens, better to use local function *)
  let only_evens2 xs =
    let is_even x = x mod 2 = 0 in
    filter (is_even, xs)

  (* Actually, we could define it *right* where we need it *)
  let only_evens3 xs =
    filter ((let is_even x = x mod 2 = 0 in is_even), xs)

  (* Seems kind of silly to have a let expression whose body is the variable...
     Can we do better? *) ;;
val only_evens2 : int list -> int list = <fun>
val only_evens3 : int list -> int list = <fun>

# (* ⚠ Not like this. a let binding is not an expression! *)
  let only_evens4 xs = filter ((let is_even x = x mod 2 = 0), xs)
  (* NOTE: let != let in! *) ;;
```

Demonstration: Anonymous functions

```
Welcome to utop version 2.16.0 (using OCaml version 5.4.0)!

# (* Two very useful and very common higher-order functions *)
  let rec map (f, xs) =
    match xs with [] -> [] | x :: xs' -> f x :: map (f, xs')

  let rec filter (f, xs) =
    match xs with
    | [] -> []
    | x :: xs' ->
      if f x then x :: filter (f, xs') else filter (f, xs') ;;

  val map : ('a -> 'b) * 'a list -> 'b list = <fun>
  val filter : ('a -> bool) * 'a list -> 'a list = <fun>

# (* Motivating anonymous functions *)
  (* If we just need is_even for only_evens, better to use local function *)
  let only_evens2 xs =
    let is_even x = x mod 2 = 0 in
    filter (is_even, xs)

  (* Actually, we could define it *right* where we need it *)
  let only_evens3 xs =
    filter ((let is_even x = x mod 2 = 0 in is_even), xs)

  (* Seems kind of silly to have a let expression whose body is the variable...
     Can we do better? *) ;;

  val only_evens2 : int list -> int list = <fun>
  val only_evens3 : int list -> int list = <fun>

# (* ⚠️ Not like this. a let binding is not an expression! *)
  let only_evens4 xs = filter ((let is_even x = x mod 2 = 0), xs)
  (* NOTE: let != let in! *) ;;

  Error: Syntax error
```

Demonstration: Anonymous functions

```
Welcome to utop version 2.16.0 (using OCaml version 5.4.0)!

# (* Two very useful and very common higher-order functions *)
  let rec map (f, xs) =
    match xs with [] -> [] | x :: xs' -> f x :: map (f, xs')

  let rec filter (f, xs) =
    match xs with
    | [] -> []
    | x :: xs' ->
      if f x then x :: filter (f, xs') else filter (f, xs') ;;

val map : ('a -> 'b) * 'a list -> 'b list = <fun>
val filter : ('a -> bool) * 'a list -> 'a list = <fun>

# (* Motivating anonymous functions *)
  (* If we just need is_even for only_evens, better to use local function *)
  let only_evens2 xs =
    let is_even x = x mod 2 = 0 in
    filter (is_even, xs)

  (* Actually, we could define it *right* where we need it *)
  let only_evens3 xs =
    filter ((let is_even x = x mod 2 = 0 in is_even), xs)

  (* Seems kind of silly to have a let expression whose body is the variable...
     Can we do better? *) ;;

val only_evens2 : int list -> int list = <fun>
val only_evens3 : int list -> int list = <fun>

# (* ⚠️ Not like this. a let binding is not an expression! *)
  let only_evens4 xs = filter ((let is_even x = x mod 2 = 0), xs)
  (* NOTE: let != let in! *) ;;

Error: Syntax error

# (* ... but an anonymous function is an expression! *)
  let only_evens5 xs = filter ((fun x -> x mod 2 = 0), xs)

  (* 😭 Bad style: the 'if e then true else false' of functions *)
  (* "unnecessary function wrapping" *)
  let tail_n_times_bad_style (n, xs) =
    n_times ((fun ys -> List.tl ys), n, xs)

  (* 😎 Good style *)
  let tail_n_times_good_style (n, xs) =
    n_times (List.tl, n, xs) ;;
```

Demonstration: Anonymous functions

```
Welcome to utop version 2.16.0 (using OCaml version 5.4.0)!

# (* Two very useful and very common higher-order functions *)
  let rec map (f, xs) =
    match xs with [] -> [] | x :: xs' -> f x :: map (f, xs')

  let rec filter (f, xs) =
    match xs with
    | [] -> []
    | x :: xs' ->
      if f x then x :: filter (f, xs') else filter (f, xs') ;;

val map : ('a -> 'b) * 'a list -> 'b list = <fun>
val filter : ('a -> bool) * 'a list -> 'a list = <fun>

# (* Motivating anonymous functions *)
  (* If we just need is_even for only_evens, better to use local function *)
  let only_evens2 xs =
    let is_even x = x mod 2 = 0 in
    filter (is_even, xs)

  (* Actually, we could define it *right* where we need it *)
  let only_evens3 xs =
    filter ((let is_even x = x mod 2 = 0 in is_even), xs)

  (* Seems kind of silly to have a let expression whose body is the variable...
     Can we do better? *) ;;

val only_evens2 : int list -> int list = <fun>
val only_evens3 : int list -> int list = <fun>

# (* ⚠️ Not like this. a let binding is not an expression! *)
  let only_evens4 xs = filter ((let is_even x = x mod 2 = 0), xs)
  (* NOTE: let != let in! *) ;;

Error: Syntax error

# (* ... but an anonymous function is an expression! *)
  let only_evens5 xs = filter ((fun x -> x mod 2 = 0), xs)

  (* 🤔 Bad style: the 'if e then true else false' of functions *)
  (* "unnecessary function wrapping" *)
  let tail_n_times_bad_style (n, xs) =
    n_times ((fun ys -> List.tl ys), n, xs)

  (* 😎 Good style *)
  let tail_n_times_good_style (n, xs) =
    n_times (List.tl, n, xs) ;;

val only_evens5 : int list -> int list = <fun>
val tail_n_times_bad_style : int * 'a list -> 'a list = <fun>
val tail_n_times_good_style : int * 'a list -> 'a list = <fun>
```

Demonstration: Anonymous functions

```
Welcome to utop version 2.16.0 (using OCaml version 5.4.0)!
```

Demonstration: Anonymous functions

```
Welcome to utop version 2.16.0 (using OCaml version 5.4.0)!

# (* Returning functions: look at the types! *)
  let double_or_triple f =
    if f 7 then fun x -> 2 * x else fun x -> 3 * x
  let roundabout_double = double_or_triple (fun x -> x - 3 = 4)
  let roundabout_nine = (double_or_triple (fun x -> x = 42)) 3 ;;
```

Demonstration: Anonymous functions

```
Welcome to utop version 2.16.0 (using OCaml version 5.4.0)!

# (* Returning functions: look at the types! *)
  let double_or_triple f =
    if f 7 then fun x -> 2 * x else fun x -> 3 * x
  let roundabout_double = double_or_triple (fun x -> x - 3 = 4)
  let roundabout_nine = (double_or_triple (fun x -> x = 42)) 3 ;;

  val double_or_triple : (int -> bool) -> int -> int = <fun>
  val roundabout_double : int -> int = <fun>
  val roundabout_nine : int = 9
```

Demonstration: Anonymous functions

```
Welcome to utop version 2.16.0 (using OCaml version 5.4.0)!

# (* Returning functions: look at the types! *)
  let double_or_triple f =
    if f 7 then fun x -> 2 * x else fun x -> 3 * x
  let roundabout_double = double_or_triple (fun x -> x - 3 = 4)
  let roundabout_nine = (double_or_triple (fun x -> x = 42)) 3 ;;

  val double_or_triple : (int -> bool) -> int -> int = <fun>
  val roundabout_double : int -> int = <fun>
  val roundabout_nine : int = 9

# (* Higher-order functions over our own recursive variant types *)
  type exp =
    | Const of int
    | Negate of exp
    | Add of exp * exp
    | Multiply of exp * exp

  let rec true_of_all_constants (f, e) =
    match e with
    | Const n -> f n
    | Negate e' -> true_of_all_constants (f, e')
    | Add (e1, e2) ->
        true_of_all_constants (f, e1) && true_of_all_constants (f, e2)
    | Multiply (e1, e2) ->
        true_of_all_constants (f, e1) && true_of_all_constants (f, e2)

  let all_even_constants e =
    true_of_all_constants ((fun x -> x mod 2 = 0), e)
  in
  all_even_constants (Add (Const 6, Multiply (Const 5, Const 10))) ;;
```

Demonstration: Anonymous functions

```
Welcome to utop version 2.16.0 (using OCaml version 5.4.0)!

# (* Returning functions: look at the types! *)
  let double_or_triple f =
    if f 7 then fun x -> 2 * x else fun x -> 3 * x
  let roundabout_double = double_or_triple (fun x -> x - 3 = 4)
  let roundabout_nine = (double_or_triple (fun x -> x = 42)) 3 ;;

  val double_or_triple : (int -> bool) -> int -> int = <fun>
  val roundabout_double : int -> int = <fun>
  val roundabout_nine : int = 9

# (* Higher-order functions over our own recursive variant types *)
  type exp =
    | Const of int
    | Negate of exp
    | Add of exp * exp
    | Multiply of exp * exp

  let rec true_of_all_constants (f, e) =
    match e with
    | Const n -> f n
    | Negate e' -> true_of_all_constants (f, e')
    | Add (e1, e2) ->
        true_of_all_constants (f, e1) && true_of_all_constants (f, e2)
    | Multiply (e1, e2) ->
        true_of_all_constants (f, e1) && true_of_all_constants (f, e2)

  let all_even_constants e =
    true_of_all_constants ((fun x -> x mod 2 = 0), e)
  in
  all_even_constants (Add (Const 6, Multiply (Const 5, Const 10))) ;;

  type exp =
      Const of int
    | Negate of exp
    | Add of exp * exp
    | Multiply of exp * exp
  val true_of_all_constants : (int -> bool) * exp -> bool = <fun>
  val all_even_constants : exp -> bool = <fun>
  - : bool = false
```