CSC 330 / Spring 2023

# 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 *)
```

# Code demonstration

# 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

# Code demonstration

# 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!

# 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

# Code demonstration