# OCaml: Functions, Lists, and Control Flow

***Programming Languages***

*William Killian*

Millersville University

# Functions

- The basic building-block of Ocaml
  - Functions **are expressions**
  - Functions **have a type**
  - Functions (when fully invoked) **yield a value**

- Looks like a binding at first

```
let x = 4;;
let my_function x y = (* hidden *);;
```

# Function Syntax

Abbreviated:

```
let fn x =
  (* code that uses x *)
;;
```

Full:

```
let fn = fun x ->
  (* code that uses x *)
;;
```

# Function Syntax (two params)

Abbreviated:

```
let fn x y =
  (* code that uses x and y *)
;;
```

Full:

```
let fn = fun x -> fun y ->
  (* code that uses x and y *)
;;
```

# Function Syntax (three params)

Abbreviated:

```
let fn x y z =
  (* code that uses x, y, and z *)
;;
```

Full:

```
let fn = fun x -> fun y -> fun z ->
  (* code that uses x, y, and z *)
;;
```

# Function Evaluation

```
let add2 x =
  2 + x
;;

add2 3  =  2 + 3  =  5
```

# Function Evaluation (two params)

```
let add x y =
  x + y
;;

add 2 3 =  2 + 3  =  5
```

# Partial Function Evaluation

```
let add x y =
  x + y
;;

let add2 = add 2
;;
```

# Partial Function Evaluation

```
let add = fun x -> fun y ->
  x + y
;;

let add2 = add 2 (* substitute 2 for x *)
;;
```

# Partial Function Evaluation

```
let add = fun x -> fun y ->
  x + y
;;
```

`let add2 =` ~~`fun x ->`~~ `fun y ->`

`  2 + y`

`;;`

# Partial Function Evaluation

```
let add = fun x -> fun y ->
  x + y
;;

let add2 = fun y ->
  2 + y
;;
```

# Aside: Local Binding

- Bindings are applied at the **global scope**
- If we want a local binding that is temporarily used, we have a special syntax `let … in`
- You can view this like a "local variable"

```
let x = 4;;
let x4 =
  let x2 = x * x in
    x2 * x2;;
(* x2 not visible *)
```

# Basic Control Flow

- In Programming 1 we learn about conditionals
  - Basic constructs: if, else
  - Ideas: Boolean expression

```
let even_odd val =
  let is_even = val mod 2 = 0 in
    if is_even then "even" else "odd"
```

# Basic Control Flow: Operators

| | |
|---|---|
| = | equality |
| != | Inequality (can also use <>) |
| > | Greater |
| < | Less |
| >= | Greater or equal |
| <= | Less or equal |

**NOTE:** always must compare the same types

All comparisons return a `bool` (true or false)

# Recursive Functions

Almost the same syntax

Just need to tell OCaml a function is recursive

```
let rec sumToN n =
  if n = 0 then
    0
  else
    n + sumToN (n - 1)
;;
```

# Lists `a' list`

- Immutable (cannot be changed)
- Finite sequence of elements
- All elements must be the same type

Empty list:

```
[]
```

List with three ints:

```
[1; 2; 3]
1::2::3::[]
1::(2::(3::([])))
```

# List Operators `Cons ::`

Prepend an element to a list

- Does ***not*** modify the original list
- The original list can be empty
- The ***types*** must match

```
module List
let cons (val : a') (lst : a' list) =
    val::lst
```

# List Operators Append @

Appends a list to the end of another list

- Does ***not*** modify either original list
- The ***types*** must match

```
[1] @ [2; 3; 4]            [1; 2; 3; 4]
[1; 2] @ [3]               [1; 2; 3]
```

```
let (@) (l1 : a' list) (l2 : a' list) =
    (* implementation hidden *)
```

# List Operators hd

Extract the first element of the list

- Returns the “left side” of the cons

```
List.hd [1; 2; 3]                    1
```

```
module List
let hd (lst : a' list) =
  match lst with
  | hd::_ -> hd
  | [] -> raise (Failure "empty list")
```

# List Operators `tl`

Extract the remaining elements of the list

- Returns the “right side” of the cons

```
List.tl [1; 2; 3]                    [2; 3]
```

```
module List
let tl (lst : a' list) =
  match lst with
  | _::tl -> tl
  | [] -> raise (Failure "empty list")
```

# Advanced Control Flow

- What was `match ... with` ?
- Language feature called “pattern matching”
- **SUPER POWERFUL**
- OCaml will try to do a lot for you
  - If the value matches -> use it
  - If the type matches -> use it
  - If it would be a well-formed expression -> use it

# Basic Pattern Matching

```
if expr then valT else valF
```

Can be rewritten as:

```
match expr with
| true -> valT
| false -> valF
```

Or:

```
match expr with
| true -> valT
| _ -> valF
```

# List Pattern Matching

- Let’s revisit List.hd

```
let hd (lst : a' list) =
  match lst with
    (* we can extract the front *)
  | hd::_ -> hd
    (* have an empty list – bad *)
  | [] -> raise (Failure "empty list")
```

# Value Pattern Matching

*Print out a number. But for multiples of three it should output "Fizz" instead of the number and for the multiples of five output "Buzz". For numbers which are multiples of both three and five output "FizzBuzz".*

```
let fizzbuzz n =
  match (n mod 3, n mod 5) with
  | (0, 0) -> "FizzBuzz"
  | (0, _) -> "Fizz"
  | (_, 0) -> "Buzz”
  | _      -> string_of_int n
```

# Value Pattern Matching

*Print out a number. But for multiples of three it should output “Fizz” instead of the number and for the multiples of five output “Buzz”. For numbers which are multiples of both three and five output “FizzBuzz”.*

```
let fizzbuzz n =
  match (n mod 3, n mod 5) with
  | (0, 0) -> "FizzBuzz"
  | (0, _) -> "Fizz"
  | (_, 0) -> "Buzz”
  | _      -> string_of_int n
```