CMSC 330
Organization of Programming Languages

OCaml
Higher Order Functions

CMSC 330 – Spring 2024
Anonymous Functions

- Use **fun** to make a function with no name

```
(fun x -> x + 3) 5
```

**Parameter**

**(fun x -> x + 3)**

**Body**

(in which parameter x is bound)

```
fun x -> x + 3
```

= 8
Anonymous Functions

- **Syntax**
  - fun \( x_1 \ldots x_n \rightarrow e \)

- **Evaluation**
  - An anonymous function is an expression
  - In fact, it is a value.

- **Type checking**
  - \((\text{fun } x_1 \ldots x_n \rightarrow e) : (t_1 \rightarrow \ldots \rightarrow t_n \rightarrow u)\)
    when \( e : u \) under assumptions \( x_1 : t_1, \ldots, x_n : t_n \).
      - (Same rule as \texttt{let } f x_1 \ldots x_n = e)
Quiz 1: What does this evaluate to?

```ml
let y = (fun x -> x+1) 2 in
(fun z -> z-1) y
```

A. Error
B. 2
C. 1
D. 0
Quiz 1: What does this evaluate to?

```
let y = (fun x -> x+1) 2 in
(fun z -> z-1) y
```

A. Error
B. 2
C. 1
D. 0
Quiz 2: What is this expression’s type?

\[(\text{fun } x \ y \ \rightarrow \ x) \ 2 \ 3\]

A. Type error
B. int
C. int \rightarrow\ int \rightarrow\ int
D. 'a \rightarrow\ 'b \rightarrow\ 'a
Quiz 2: What is this expression’s type?

(fun x y -> x) 2 3

A. Type error
B. int
C. int -> int -> int
D. 'a -> 'b -> 'a
Functions and Binding

- Functions are first-class, so you can bind them to other names as you like

```ocaml
let f x = x + 3;;
let g = f;;
g 5
```

Example Shorthands

- let for functions is a syntactic shorthand
  
  ```
  let f x = body
  ```

  is semantically equivalent to

  ```
  let f = fun x -> body
  ```

- let next x = x + 1
  
  Short for

  ```
  let next = fun x -> x + 1
  ```

- let plus x y = x + y
  
  Short for

  ```
  let plus = fun x y -> x + y
  ```
Quiz 3: What does this evaluate to?

```
let f = fun x -> 0 in
let g = f in
let h = fun y -> g (y+1) in
h 1
```

A. 0  
B. 1  
C. 2  
D. Error
Quiz 3: What does this evaluate to?

```
let f = fun x -> 0 in
let g = f in
let h = fun y -> g (y+1)
h 1
```

A. 0
B. 1
C. 2
D. Error
Nested Functions

(* Filter the odd numbers from a list *)
let filter lst =
    let rec aux l =
        match l with
        |[] -> []
        |h::t-> if h mod 2 <> 0 then h::aux t
            else aux t
    in
    aux lst

filter [1;2;3;4;5;6] (* int list = [1; 3; 5] *)
Passing Functions as Arguments

You can pass functions as arguments

```
let plus3 x = x + 3 (* int -> int *)

let twice f z = f (f z)
(* ('a->'a) -> 'a -> 'a *)

twice plus3 5 = 11
```