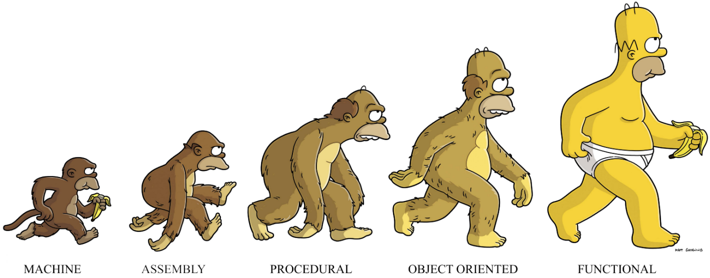
COMP302: Programming Languages and Paradigms

Prof. Brigitte Pientka (Sec 01)
bpientka@cs.mcgill.ca

Francisco Ferreira (Sec 02)
fferre8@cs.mcgill.ca

School of Computer Science
McGill University

Week 4-1, Fall 2017



"Higher-order functions; relatively simple once mastered yet very powerful and they make you look awesome! "

Jeremie Poisson (TA for COMP 302)

Office Hours: Wed 12:00pm - 2:00pm

"Higher order functions are super cool!"

Eric Zhang (TA for COMP 302)

Office Hours: Wed 2:00pm - 4:00pm

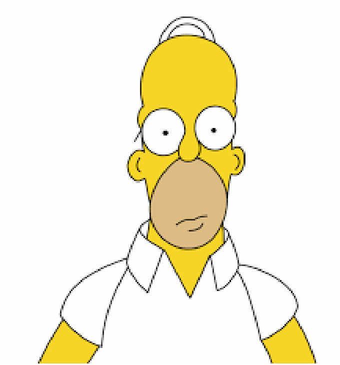
Slogan

Functions are first-class values!

Slogan

Functions are first-class values!

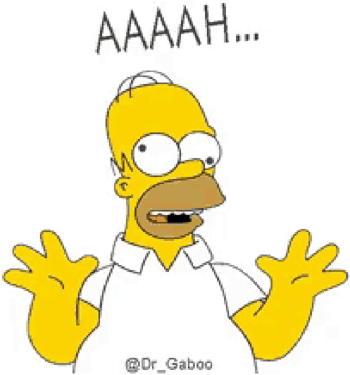
- Pass functions as arguments (Continued)
- Return them as results (Today)



Slogan

Functions are first-class values!

- Pass functions as arguments (Continued)
- Return them as results (Today)



Common Higher-Order Functions (Built-In)

Common Higher-Order Functions (Built-In)

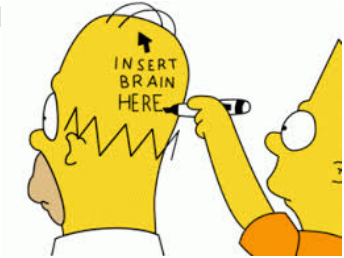
- `List.map: ('a -> 'b) -> 'a list -> 'b list`
- `List.filter: ('a -> bool) -> 'a list -> 'a list`
- `List.fold_right: ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b`
- `List.fold_left: ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a`
- `List.for_all: ('a -> bool) -> 'a list -> bool`
- `List.exists : ('a -> bool) -> 'a list -> bool`

Check the OCaml `List` library for more built-in higher-order functions! They make great practice questions!

Slogan – Revisited

Functions are first-class values!

- Pass functions as arguments (Continued
- **Return them as results (Today)**



What does it mean to return a function?

Let's go back to the beginning ... from the 1. week

```
(* We can also bind variable to functions. *)
let area : float -> float = function r -> pi *. r *. r

(* or more conveniently, we write usually *)
let area (r:float) = pi *. r *. r

```

- The variable name `area` is bound to the *value* `function r -> pi *. r *. r` which OCaml prints simply as `<fun>`.
- The type of the variable `area` is `float -> float`.

Switching your viewpoint

Write a function `curry` that

- takes as input a function `f:('a * 'b) -> 'c`
- returns as a result a function `'a -> 'b -> 'c`.

Haskell B. Curry

Switching your viewpoint

Write a function `curry` that

- takes as input a function `f:('a * 'b) -> 'c`
- returns as a result a function `'a -> 'b -> 'c`.

Haskell B. Curry

```
(* curry : (('a * 'b) -> 'c) -> 'a -> 'b -> 'c *)
(* Note : Arrows are right-associative.        *)
let curry f = (fun x y -> f (x,y))
```

Switching your viewpoint

Write a function `curry` that

- takes as input a function `f:('a * 'b) -> 'c`
- returns as a result a function `'a -> 'b -> 'c`.

Haskell B. Curry

```
(* curry : (('a * 'b) -> 'c) -> 'a -> 'b -> 'c *)
(* Note : Arrows are right-associative.        *)
let curry f = (fun x y -> f (x,y))

let curry_version2 f x y = f (x,y)

let curry_version3 = fun f -> fun x -> fun y -> f (x,y)
```

Switching your viewpoint

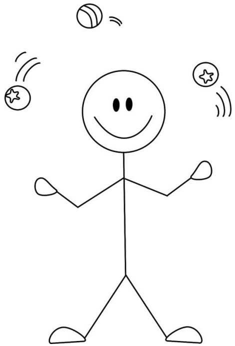
Write a function `curry` that

- takes as input a function `f:('a * 'b) -> 'c`
- returns as a result a function `'a -> 'b -> 'c`.

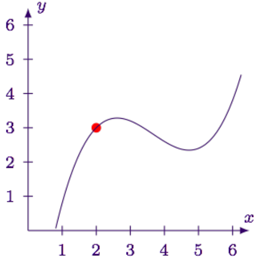
Haskell B. Curry

```
(* curry : (('a * 'b) -> 'c) -> 'a -> 'b -> 'c *)
(* Note : Arrows are right-associative.        *)
let curry f = (fun x y -> f (x,y))

let curry_version2 f x y = f (x,y)

let curry_version3 = fun f -> fun x -> fun y -> f (x,y)
```

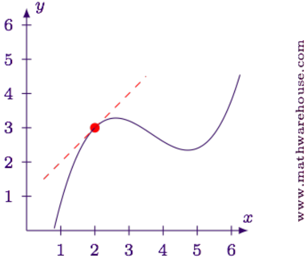
Let's play!



Bonus: Approximating the Derivative



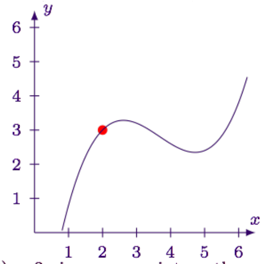
$f(2) = 3$ gives us a point on the graph.



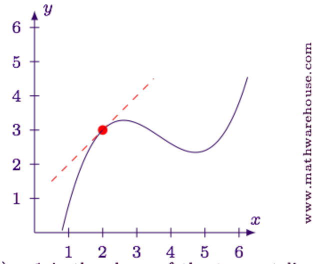
$f'(2) = 1$ is the slope of the tangent line.

$$f'(x) = \frac{df}{dx} = \lim_{\epsilon \to 0} \frac{f(x+\epsilon) - f(x)}{\epsilon}$$

Bonus: Approximating the Derivative



$f(2) = 3$ gives us a point on the graph.



$f'(2) = 1$ is the slope of the tangent line.

$$f'(x) = \frac{df}{dx} = \lim_{\epsilon \to 0} \frac{f(x+\epsilon) - f(x)}{\epsilon}$$

Implement a function `deriv : (float -> float) * float -> float -> float` which

- given a function `f:float -> float` and an epsilon `dx:float`
- returns a function `float -> float` describing the derivative of `f`.

Bonus: Approximating the Derivative

$$f'(x) = \frac{df}{dx} = \lim_{\epsilon \to 0} \frac{f(x+\epsilon) - f(x)}{\epsilon}$$

Implement a function `deriv : (float -> float) * float -> float -> float` which

- given a function `f:float -> float` and an epsilon `dx:float`
- returns a function `float -> float` describing the derivative of `f`.

Bonus: Approximating the Derivative

$$f'(x) = \frac{df}{dx} = \lim_{\epsilon \to 0} \frac{f(x+\epsilon) - f(x)}{\epsilon}$$

Implement a function `deriv : (float -> float) * float -> float -> float` which

- given a function `f:float -> float` and an epsilon `dx:float`
- returns a function `float -> float` describing the derivative of `f`.

```
let deriv (f, dx) = fun x -> (f (x +. dx) -. f x) /. dx
```

Higher-order functions are super cool!

Higher-order functions are super cool!

Question: Do you know what the functions in the picture mean?

Functional Tidbit: Church and the Lambda-Calculus

- Logician and Mathematician
- June 14, 1903 – August 11, 1995
- Most known for the Lambda-Calculus:
  - a simple language consisting of variables, functions (written as $\lambda x.t$) and function application
  - we can define all computable functions in the Lambda-Calculus!

Church Encoding of Booleans:

$$
\begin{array}{rcl}
\mathbf{T} & = & \lambda x.\lambda y.x \\
\mathbf{F} & = & \lambda x.\lambda y.y
\end{array}
$$