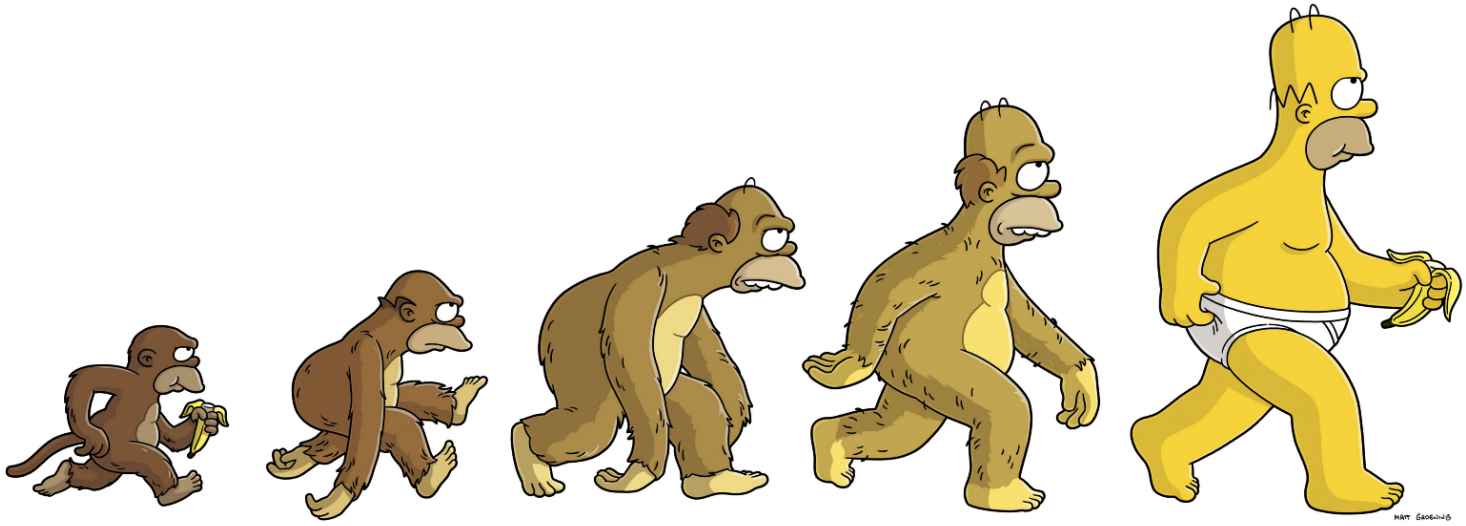
COMP302: Programming Languages and Paradigms

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 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:

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

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:

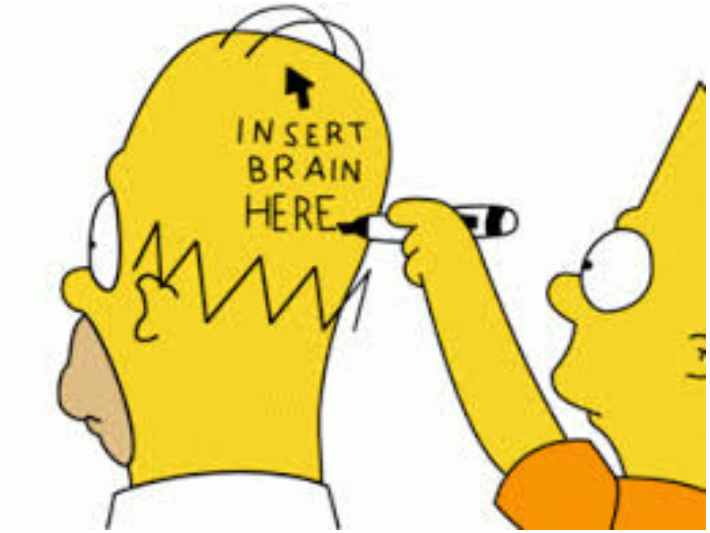
$$\begin{aligned} \mathbf{T} &= \lambda x.\lambda y.x \\ \mathbf{F} &= \lambda x.\lambda y.y \end{aligned}$$

Playing detective:
Find out how your instructors are related to Alonzo Church!
Hint: Check `http://www.genealogy.ams.org/`

Slogan – Revisited

Functions are first-class values!

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



What does it mean to return a function?

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

```
1 (* We can also bind variable to functions. *)
2 let area : float -> float = function r -> pi *. r *. r
3
4 (* or more conveniently, we write usually *)
5 let area (r:float) = pi *. r *. r
6
```

- 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

```
1 (* curry : (('a * 'b) -> 'c) -> 'a -> 'b -> 'c *)
2 (* Note : Arrows are right-associative.         *)
3 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

```
1 (* curry : (('a * 'b) -> 'c) -> 'a -> 'b -> 'c *)
2 (* Note : Arrows are right-associative.         *)
3 let curry f = (fun x y -> f (x,y))
4
5 let curry_version2 f x y = f (x,y)
6
7 let curry_version3 = fun f -> fun x -> fun y -> f (x,y)
```

Switching your viewpoint

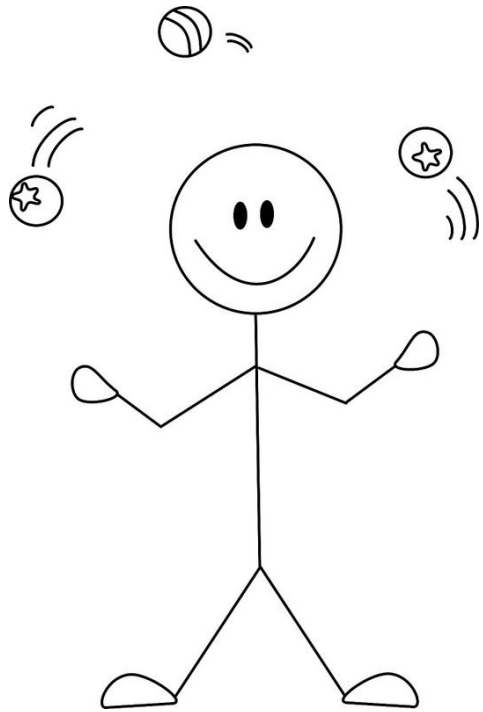
Write a function `curry` that

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

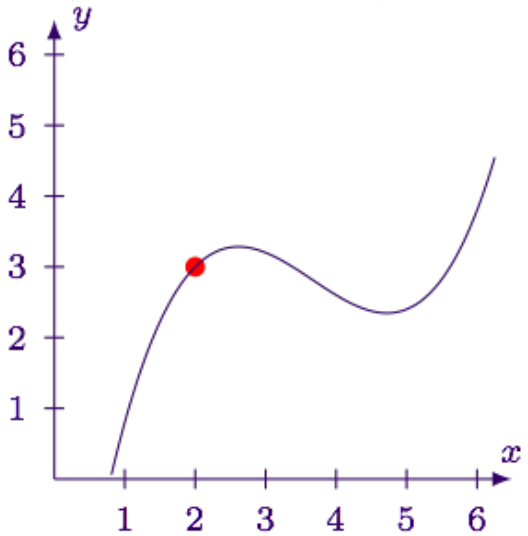
Haskell B. Curry

```
1 (* curry : (('a * 'b) -> 'c) -> 'a -> 'b -> 'c *)
2 (* Note : Arrows are right-associative.         *)
3 let curry f = (fun x y -> f (x,y))
4
5 let curry_version2 f x y = f (x,y)
6
7 let curry_version3 = fun f -> fun x -> fun y -> f (x,y)
```

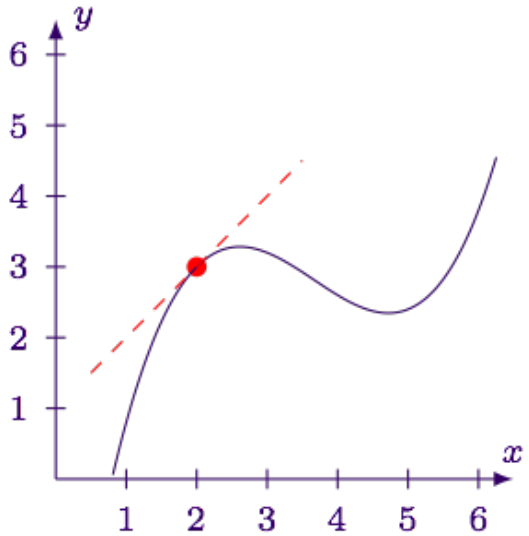
Let's play!



Bonus: Approximating the Derivative



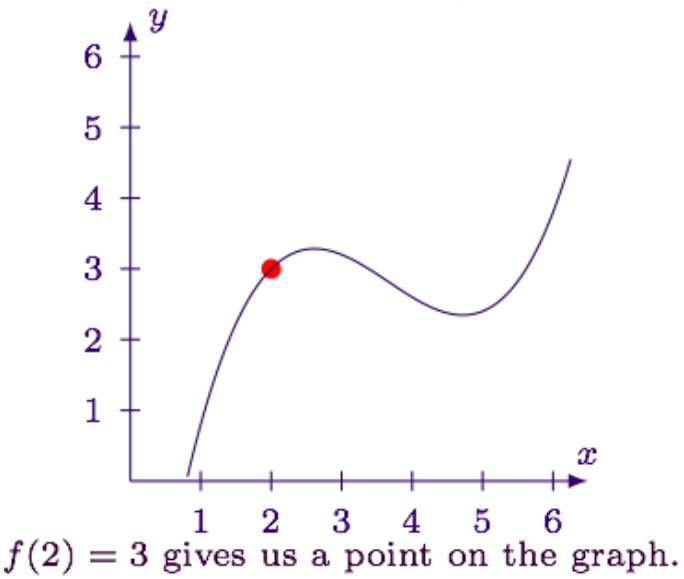
$f(2) = 3$ gives us a point on the graph.



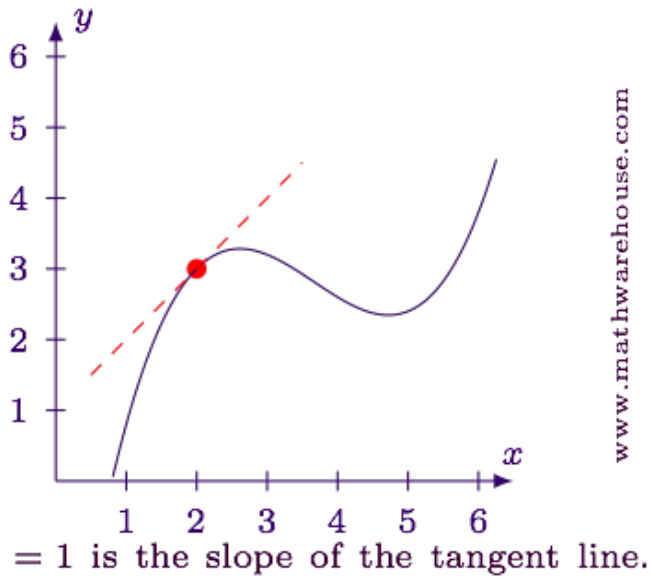
$f'(2) = 1$ is the slope of the tangent line.

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

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`.

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

Demo

Partial Evaluation

- A technique for optimizing and specializing programs
- Generate programs from other programs!
- Produce new programs which run faster than the originals while being guaranteed to behave in the same way!

What is the result of evaluation?

```
1 (* plus : int -> int -> int *)
2
3 let plusSq x y =  x * x + y * y
4
5 (* plus3 : int -> int *)
6 let plus3 = (plusSq 3)
```

What is the result of evaluation?

```
1 (* plus : int -> int -> int *)
2
3 let plusSq x y =  x * x + y * y
4
5 (* plus3 : int -> int *)
6 let plus3 = (plusSq 3)
```

$\Longrightarrow$ `fun y -> 3 * 3 + y * y`

What is the result of evaluation?

```
1 (* plus : int -> int -> int *)
2
3 let plusSq x y =  x * x + y * y
4
5 (* plus3 : int -> int *)
6 let plus3 = (plusSq 3)
```

$\Longrightarrow$ `fun y -> 3 * 3 + y * y`

OK – OCaml actually just shows you:

```
1 val plusSq : int -> int -> int = <fun>
2 val plus3 : int -> int = <fun>
```

What is the result of evaluation?

```
1 (* plus : int -> int -> int *)
2
3 let plusSq x y =  x * x + y * y
4
5 (* plus3 : int -> int *)
6 let plus3 = (plusSq 3)
```

$\Longrightarrow$ `fun y -> 3 * 3 + y * y`

What is important to remember:

- We do not evaluate inside function bodies
- We only evaluate the function body when we have **all** arguments

The operational semantics (i.e. how your program is executed) matters!

Let's see how to take advantage of it!