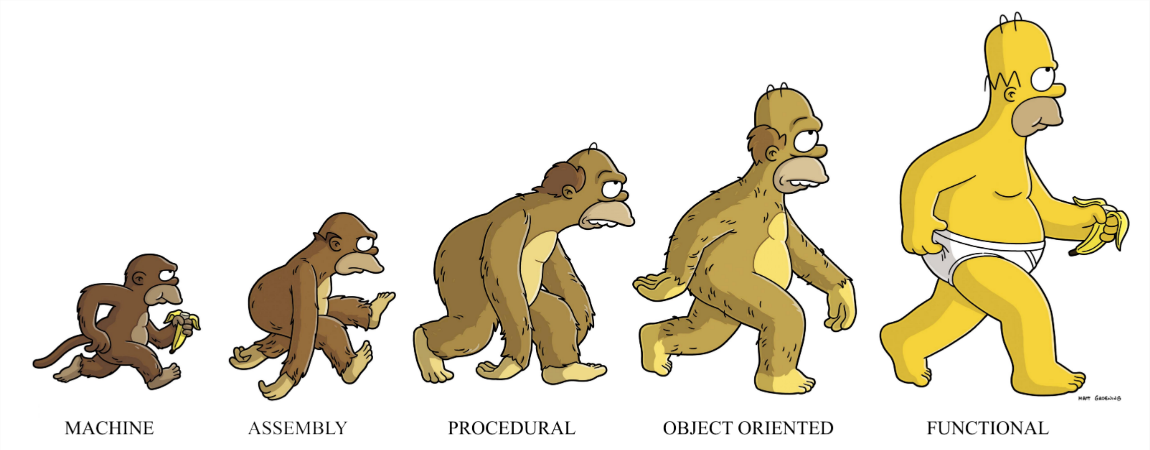
COMP302: Programming Languages and Paradigms

Week 4: Higher-Order Functions – Part 1

Prof. Brigitte Pientka <bpientka@cs.mcgill.ca>

School of Computer Science, McGill University



What is a higher-order function?

A higher-order function is a function that takes

- as input a function
- produces as an output a function

input 1 input 2 output

For example, `('a -> 'b) -> 'a list -> 'b list` is the type of a higher-order function for operating on lists.

MAP

It takes two arguments.

a function `'a -> 'b`

an input list of type `'a list`

Why are higher-order functions cool?

Whereas ordinary functions let us abstract over *data*, higher-order functions let us abstract over *functionality*.

- Programs can be very short and compact
- Programs are reusable, well-structured, modular!
- Each significant piece of functionality is implemented in one place.

Slogan

Functions are first-class values!

- Pass functions as arguments (Today)
- Return them as results (Next time)

Slogan

Functions are first-class values!

- Pass functions as arguments (Today)
- Return them as results (Next time)

Abstracting over common functionality

$$\sum_{k=a}^{k=b} k$$

```
(* sum: int * int -> int *)
let rec sum (a,b) =
if a > b then 0 else a + sum(a+1,b)
```

$$\sum_{k=a}^{k=b} k^2$$

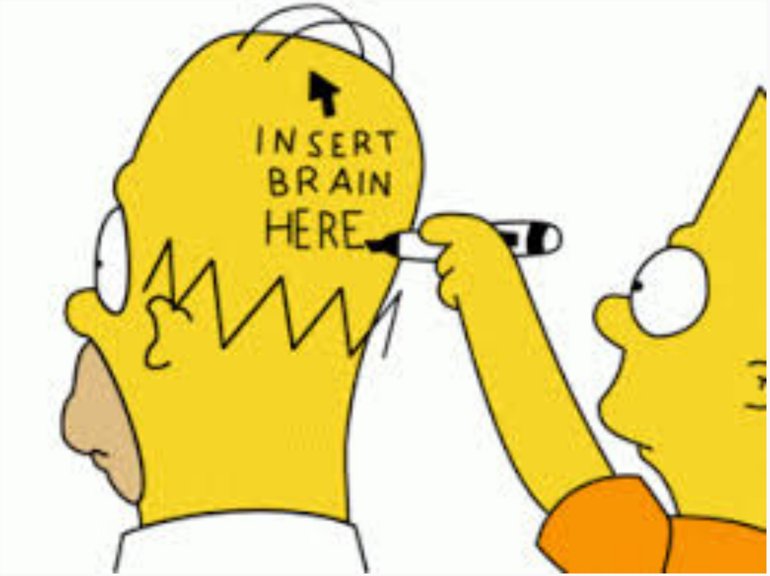
```
let rec sum (a,b) =
if a > b then 0 else square(a) + sum(a+1,b)
```

$$\sum_{k=a}^{k=b} 2^k$$

```
let rec sum (a,b) =
if a > b then 0 else exp(2,a) + sum(a+1,b)
```

Can we write a generic sum function?

Non-Generic `sum` (old)	Generic `sum` (new) with a function as an argument
`sum: int * int -> int`	`sum: (int -> int) -> int * int -> int`


INSERT
BRAIN
HERE

Abstracting over common functionality

(int -> int) -> (int * int) -> int

```
let rec sum f (a, b) =
if (a > b) then 0 else (f a) + sum f (a+1, b)
```

How about only summing up odd numbers between `a` and `b`?

Abstracting over common functionality

```
let rec sum f (a, b) =
if (a > b) then 0 else (f a) + sum f (a+2, b)
```

odd

How about only summing up odd numbers between `a` and `b`?

let id = fun x -> x
equivalent to
let id x = x

```
let rec sumOdd (a, b) =
 if (a mod 2) = 1 then
  sum (fun x -> x) (a, b)         (* a was odd *)
 else
  sum (fun x -> x) (a+1, b)       (* a was even *)
```

anonymous function

Abstracting over common functionality (increment)

```
let rec sum f (a, b) inc =
if (a > b) then 0 else (f a) + sum f (inc(a), b) inc
```

int -> int

How about only summing up odd numbers between `a` and `b`?

```
let rec sumOdd (a, b) =
 if (a mod 2) = 1 then
  sum (fun x -> x) (a, b) (fun x -> x + 2)          (* a was odd *)
 else
  sum (fun x -> x) (a+1, b) (fun x -> x + 2)        (* a was even *)
```

Abstracting over common functionality (tail-recursively) how we combine numbers in each step

```
let rec sum f (a, b) inc acc =
if (a > b) then acc else sum f (inc(a), b) inc (f a + acc)
```

0

How about only multiplying numbers between a and b?

```
let rec product f (a, b) inc acc =
if (a > b) then acc else product f (inc(a), b) inc (f a * acc)
```

1

Take away

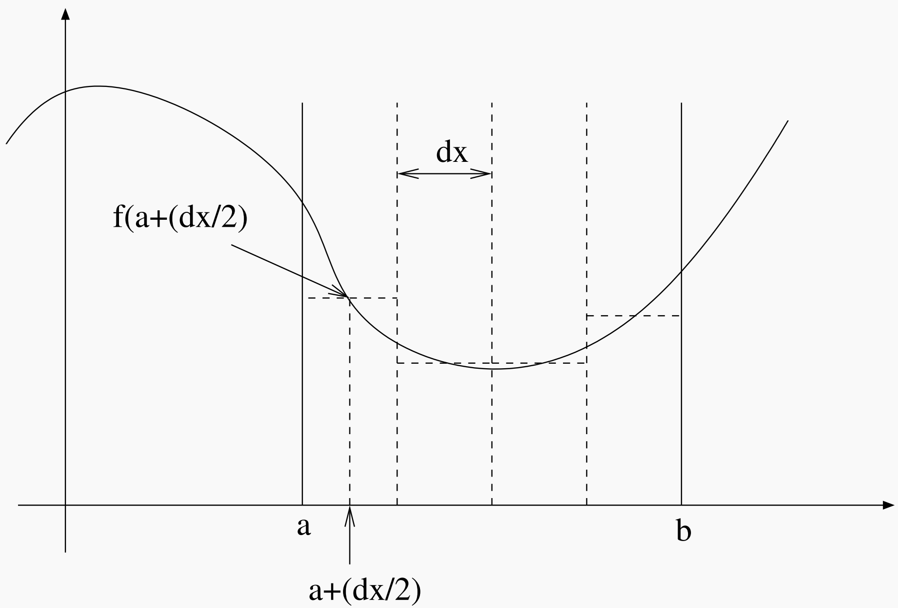
Abstraction and higher-order functions are very powerful mechanisms for writing reusable prorgrams.

Computing a series

```
series:  (int -> int -> int)   (*  comb     *)
       -> (int -> int)         (*  f        *)   input
       -> (int * int)          (* (a,b)     *)
       -> (int -> int)         (*  inc      *)
       -> int                  (*  acc      *)
       -> int                  (*  result   *)   output
```

```
let sum  f (a,b) inc = series (fun x y -> x + y) f (a,b) inc 0
let prod f (a,b) inc = series (fun x y -> x * y) f (a,b) inc 1
```

Beauty of Higher-Order Functions



Let $l = a + dx/2$.

$$\begin{aligned} \int_a^b f(x)\,dx &\approx f(l) * dx + f(l + dx) * dx + f(l + dx + dx) * dx + \ldots \\ &= dx * (f(l) + f(l + dx) + f(l + 2 * dx) + f(l + 3 * dx) \ldots) \end{aligned}$$

Beauty of Higher-Order Functions

Let $l = a + dx/2$.

$$\begin{aligned} \int_a^b f(x)\,dx &\approx f(l) * dx + f(l + dx) * dx + f(l + dx + dx) * dx + \dots \\ &= dx * (f(l) + f(l + dx) + f(l + 2 * dx) + f(l + 3 * dx) \dots) \end{aligned}$$

```
let integral f (lo,hi) dx =
    dx *. iter_sum f
                 (lo +. (dx /. 2.0) , hi)
                 (fun x -> x +. dx)
```

where

```
iter_sum: (float -> float)   (*  f      *)
       -> (int * int)        (* (a,b)   *)
       -> (int -> int)       (*  inc    *)
```

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! And we'll discuss how to implement them during class!

Take-Away

Passing functions as arguments

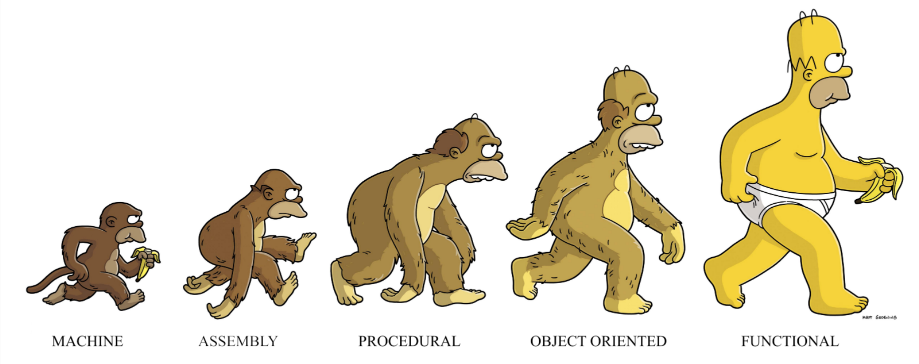
- allows us abstract over common functionality.
- enables code reuse
- means functionality is implemented in one place

COMP302: Programming Languages and Paradigms

Week 4: Higher-Order Functions – Part 2

Prof. Brigitte Pientka <bpientka@cs.mcgill.ca>

School of Computer Science, McGill University



Slogan

Functions are first-class values!

Slogan

Functions are first-class values!

- Pass functions as arguments (Last Time and Today)
- 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
                            fun 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

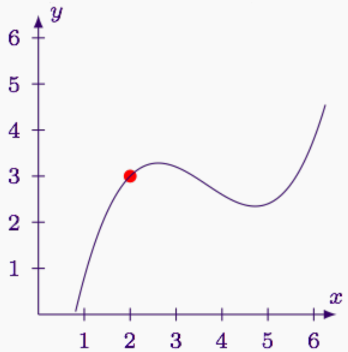
- takes as input a function `f : ('a * 'b) -> 'c`
- returns as a result a function `'a -> 'b -> 'c`.

Haskell B. Curry

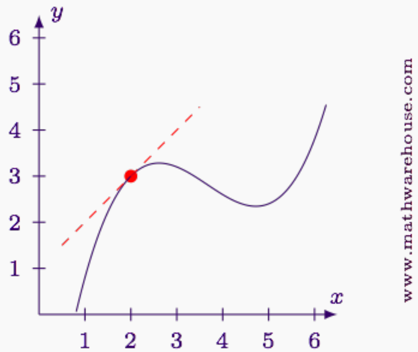
NOT necessary

```
(* 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)
```

Example 1: 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`.

Example 1: 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` that computes the derivative of `f` at a given point.

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

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

Returning functions: code generation

We can use higher-order functions to perform partial evaluation or code generation.

Using Partial Evaluation Effectively

Partial evaluation evaluates a function by only passing some of its inputs.

```
# let plusSq = fun x -> fun y -> x * x + y * y;;
val plusSq : int -> int -> int = <fun>
plusSq 3;;
- : int -> int = <fun>
```

What does `<fun>` stand for? How does it look like?

Morally it looks like: `fun y -> 3 * 3 + y*y`

- Never evaluates inside the function body
- Stop evaluation as soon as a value is reached
- Remember: Functions are values!

Using Partial Evaluation Effectively

Partial evaluation evaluates a function by only passing some of its inputs.

```
# let plusSq = fun x -> fun y -> x * x + y * y;;
val plusSq : int -> int -> int = <fun>
plusSq 3;;
- : int -> int = <fun>
```

What does `<fun>` stand for? How does it look like?

Morally it looks like: `fun y -> 3 * 3 + y*y`

How do we generate a function that does compute `3 * 3`?

Postpone creating the function `fun y -> ....`

Using Partial Evaluation Effectively

Rewrite

```
# let plusSq = fun x -> fun y -> x * x + y * y;;
val plusSq : int -> int -> int = <fun>
```

to

```
let plusSq = fun x -> let r = x * x in fun y ->  r + y * y;;
```

When we evaluate

```
plusSq 3  ⟹  let r = 3 * 3 in fun y -> r + y * y
          ⟹  let r = 9 in fun y -> r + y * y
          ⟹  fun y -> 9 + y * y
```

NOT yet a value.

Programmers control at what point a function is created and returned.

Returning functions: code generation

Consider the function `pow : int -> int -> int` that computes n^k.

```
let rec pow k n =
  if k = 0 then 1
  else n * pow (k-1) n

```

The expression `pow 2` does not evaluate further; functions wait until all their arguments are given before reducing.

By cleverly refactoring, we can get this to compute even given only one argument!

```
(* pow : int -> int -> int *)
let rec pow k =
  if k = 0 then (fun n -> 1)
  else let r = pow (k-1) in fun n -> n * r n
```

delay the creation
of the function waiting for n.

Code Generation in Action

```
(* pow : int -> int -> int *)
let rec pow k =
  if k = 0 then (fun n -> 1)
  else let r = pow (k-1) in fun n -> n * r n
```

```
pow 1  ⟹  if 1 = 0 then (fun n -> 1) else let r = pow (1-1) in fun n -> n * r n
       ⟹  let r = pow (1-1) in fun n -> n * r n
       ⟹  let r = pow (0) in fun n -> n * r n
       ⟹  let r = fun n -> 1 in fun n -> n * r n
       ⟹  fun n -> n * (fun n -> 1) n
```

- We have generated code that is independent ot `pow`.
- The code computes essentially `fun n -> n * 1`

Higher-order functions are super cool!

T = func x → func y → x

F = func x → func y → y

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:

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

Take Away

Functions are first-class values

- We do not evaluate inside functions.
- Stop evaluation as soon as a value is reached
- We control when and where functions are created.
- Returning functions allows us partial evaluation which can lead to substantial performance gains