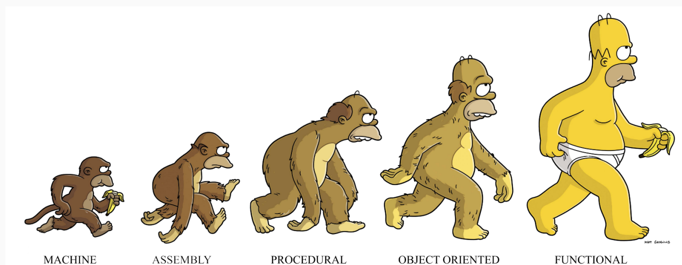
COMP302: Programming Languages and Paradigms

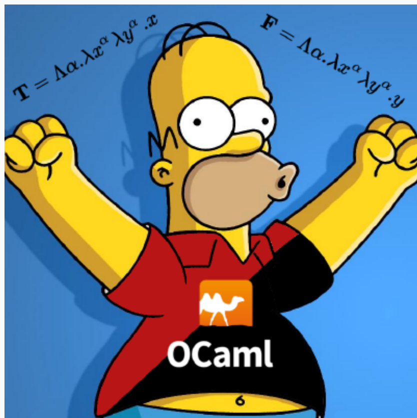
Week 4: Higher-Order Functions – Part 1

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

School of Computer Science, McGill University



Functional Tidbit: Words of Wisdom



"Higher-order functions are cool!"
- Kelvin Tagoe (Former TA for COMP 302)

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

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

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

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*.

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.

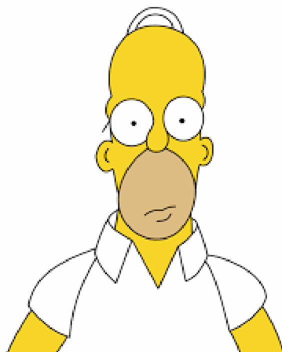
Slogan

Functions are first-class values!

Slogan

Functions are first-class values!

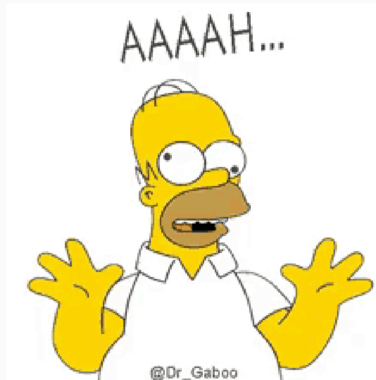
- Pass functions as arguments (Today)
- Return them as results (Next time)



Slogan

Functions are first-class values!

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



Abstracting over common functionality

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

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)
```

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)
```

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

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)
```

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

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)
```

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?

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)
```

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

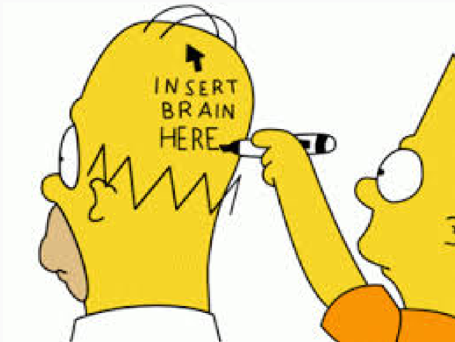
```
let rec sum (a,b) =
if a > b then 0 else square(a) + sum(a+1,b)
```

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

```
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

```
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)
```

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)
```

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)        (* a was odd *)
 else
   sum (fun x -> x) (a+1, b)     (* a was even *)
```

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
```

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 how we combine numbers in each step

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

How about only multiplying numbers between `a` and `b`?

Abstracting over common functionality how we combine numbers in each step

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

How about only multiplying numbers between `a` and `b`?

```
let rec product f (a, b) inc =
if (a > b) then 1 else (f a) *  product f (inc(a), b) inc
```

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)
```

How about only multiplying numbers between `a` and `b`?

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)
```

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)
```

Take away

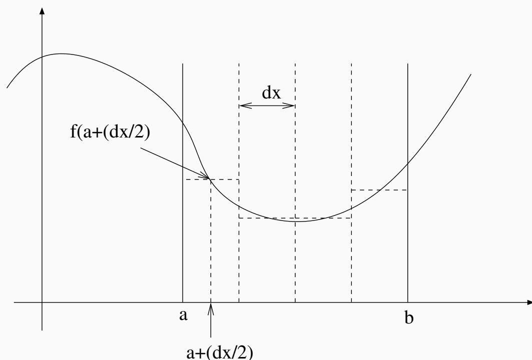
Abstraction and higher-order functions are very powerful mechanisms for writing reusable prorgrams.

Computing a series

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

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

Beauty of Higher-Order Functions



Let $I = a + dx/2$.

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

Beauty of Higher-Order Functions

Let $l = a + dx/2$.

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

Beauty of Higher-Order Functions

Let $l = a + dx/2$.

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

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

where

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

Beauty of Higher-Order Functions

Let $l = a + dx/2$.

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

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

where

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

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! 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