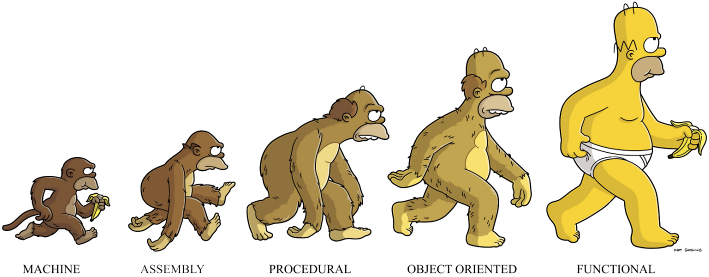
COMP302: Programming Languages and Paradigms

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

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

School of Computer Science
McGill University

Week 3-3, Fall 2017



Functional Tidbit: Words of Wisdom

"Higher-order functions are super cool!"
- Eric Zhang (TA for COMP 302)

Why are higher-order functions cool?

Higher-order functions allow us to abstract over common functionality.

Why are higher-order functions cool?

Higher-order functions allow us to abstract over common functionality.

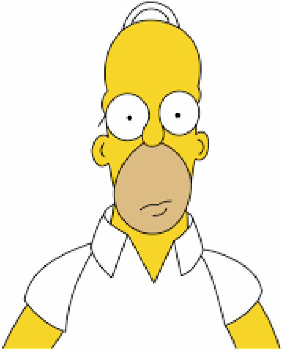
- Programs can be very short and compact
- Programs are reusable, well-structured, modular!
- Each significant piece of functionality is implemented in one place.

Slogan

Functions are first-class values!

Functions are first-class values!

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



Slogan

Functions are first-class values!

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

Abstracting over common functionality

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

Abstracting over common functionality

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

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

Abstracting over common functionality

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

```
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

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

```
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

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

```
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

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

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

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

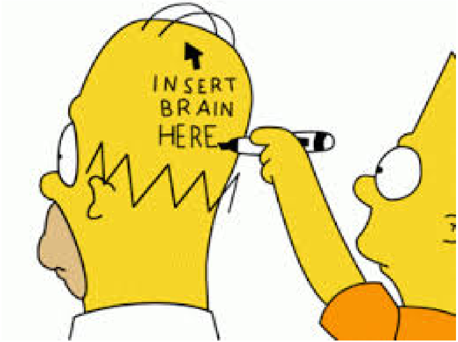
```
let rec sum (a,b) =
if a > b then 0 else square(a) + sum(a+1,b)
```

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

```
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 using a function as an argument
`sum: int * int -> int`	`sum: (int -> int) -> int * int -> int`



Demo

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 even 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 even 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 0 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 0 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 1 else product f (inc(a), b) inc (f a * acc)
```

Demo

Take away

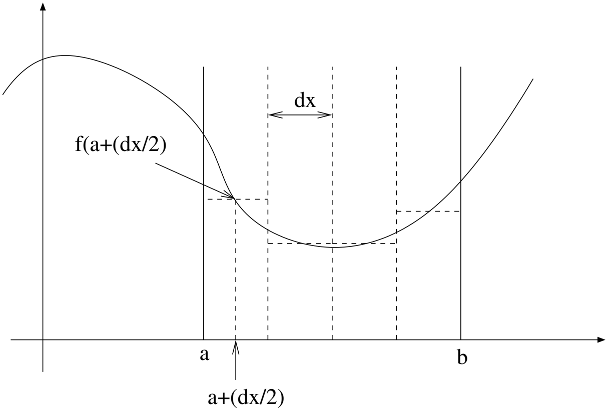
Abstraction and higher-order functions are very powerful mechanisms for writing reusable prorgrams.

Computing a series

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

```
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
```

Bonus: Approximating the integral!



Let $l = a + dx/2$.

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

Higher-order functions on data types

More higher-order functions next week!