Lecture 14:

Higher-order functions:

Use functions as data

Value-oriented programming

↳ functions map values to values

VS

imperative programming (all = C)

Avoid

repeated

code

single point of control

```
fun add1(l:int list):int list =
    (case l of
        [] => []
      | x::xs => (x+1) :: add1(xs))
```

amount

add1([1,2,3]) = [2,3,4]

```
fun add2(l:int list):int list =
    (case l of
        [] => []
      | x::xs => (x+2) :: add2(xs))
```

amount

add2([1,2,3]) = [3,4,5]

```
fun add(l:int list, a:int):int list =
    case l of
        [] => []
      | x::xs => (x+a) :: add(xs, a)
```

abstraction

```
fun add1(l:int list):int list = add(l, 1)
fun add2(l:int list):int list = add(l, 2)
```

recover originals as instances

```
fun add1(l:int list):int list =
    (case l of
        [] => []
      | x::xs => (x+1) :: add1(xs))
```

```
fun add2(l:int list):int list =
    case l of
        [] => []
      | x::xs => (x+2) :: add2(xs)
```

```
fun double(x:int):int =
    2 * x
```

E.g.
doubAll[1,2,3]
= [2, 4, 6]

```
fun doubAll(l:int list):int list =
    case l of
        [] => []
      | x::xs => double(x) :: doubAll(xs)
```

Higher-order function:

functions can take other functions as input

Function type:

input → output

int → int

(is a function whose input is int and output is also int

e.g. int → string

string → int

stands for doubling or adding 1

```
fun map (f: int -> int, l: int list): int list =
    case l of
        [] => []
      | x::xs => [f(x)] :: map(f, xs)
```

abstraction

map (f, [x1, x2, ... xn])

= [f(x1), f(x2), f(x3) ... f(xn)]

```
fun map (f: int -> int, l: int list): int list =
    case l of
      [] => []
    | x::xs => f(x) :: map (f, xs)
```

```
fun doubAll( l:int list):int list =
    case l of
      [] => []
    | x::xs => double(x) :: doubAll(xs)
```

```
fun doubAll(l:int list): int list =
    map ( double , l)
```

```
fun map (f: int -> int, l: int list): int list =
    case l of
        [] => []
      | x::xs => f(x) :: map (f, xs)
```

```
fun add1 (l: int list): int list =
    case l of
        [] => []
      | x::xs => (x+1) :: add1(xs)
```

```
fun add1num (x) = x + 1
fun add1 (l: int list) : int list =
    map ( add1num , l)
```

```
fun map (f: int -> int, l: int list): int list =
    case l of
        [] => []
      | x::xs => [f(x)] :: map (f, xs)
```

```
fun add2 (l: int list): int list =
    case l of
        [] => []
      | x::xs => (x+2) :: add1(xs)
```

```
fun add2num (x) = x+2
fun add1 (l: int list) : int list =
    map ( add2num , l)
```

```
fun map (f: int -> int, l: int list): int list =
    case l of
        [] => []
      | x::xs => [f(x)] :: map (f, xs)
```

```
fun doubAll (l: int list): int list =
    map ( double , l)
```

doubAll [1, 2, 3]

⟶ map (double, [1, 2, 3])

⟶ case [1, 2, 3] of [] => []
    | x::xs => double(x) :: map (double, xs)

⟶ double(1) :: map (double, [2, 3])

```
fun add(l:int list, a:int):int list =
  case l of
    [] => []
  | x::xs => (x+a) :: add(xs, a)
```

~~fun addTheNum(x:int, a:int):int = x+a~~

```
fun add(l:int list, a:int):int list =
  let fun addTheNum(x:int) = x+a
  in
    map(addTheNum, l)
  end
```

int → int

"closure" makes a new function every time add is called

```
fun add(l:int list, a:int):int list =
  let fun addTheNum(x:int) = x+a
  in
      map( addTheNum , l)
  end
```
(addTheNum : int → int)

```
add([1,2,3], 2)
↦ let fun addTheNum(x:int) = x+2
  in
     map(addTheNum, [1,2,3])
  end
```

Dynamically evaluated

```
add([1,2,3], 7)
↦ let fun addTheNum(x:int):int = x+7
  in
     map(addTheNum, [1,2,3])
  end
```

Type int → int

values

1. function declarations
2. anonymous functions
   - → functions that don't have names

operations

- function application

f : int → int

f(z : int) : int

```
fun double(x:int):int = 2*x

fun doubAll(l:int list):intlist = map(double, l)
```

Declarations

Anonymous functions:

```
let
  fun f(x) = 2*x
in
  f
end
```

```
fun doubAll(l:intlist):int list =
  map( fn x => 2*x , l)
```

input: x — body: 2*x

"anonymous": send x to 2*x

fn $x \Rightarrow e$ has type $int \to int$

when $e : int$

assuming $x : int$

$(\text{fn } x \Rightarrow e)\, v$

$\longmapsto$ e with v plugged in for x

$\{ x \longmapsto x^2 \}$

$\{ x \longmapsto \begin{array}{ll} x^2 & \text{if } x \text{ is even} \\ x^3 & \text{if } x \text{ is odd} \end{array} \}$

```
fun add(l:int list, a:int):int list =
    case l of
        [] => []
      | x::xs => (x+a) :: add(xs, a)
```

~~fun addTheNum(x:int, a:int):int = x+a~~

```
fun add(l:int list, a:int):int list =
```

```
map(fn x => x+a, l)
```

x+a: input → output

```
fun add (l: int list, a: int): int list =
    map (fn x => x + a, l)
```

input: `x`

output: `x + a`

add([1, 2, 3], 2)

⟼ map(fn x => x + 2, [1, 2, 3])

⟼ (fn x => x + 2) 1 :: map(fn x => x + 2, [2, 3])

⟼ (1 + 2) :: map (fn x => x + 2, [2, 3])

⟼ 3 :: ...

```
fun add(l:int list, a:int):int list =
  case l of
    [] => []
  | x::xs => (x+a) :: add(xs, a)
```

add([1,2,3], 7)
= [8, 9, 10]

```
fun doubAll(l:int list):int list =
  case l of
    [] => []
  | x::xs => double(x) :: doubAll(xs)
```

doubAll([1, 2, 3])
= [2, 4, 6]

```
fun last(l:int list) : int = ...
fun lasts(l:int list list):int list =
  case l of
    [] => []
  | x::xs => last(x) :: lasts(xs)
```

lasts [[1,2,3], [4,5]]
= [3, 5]

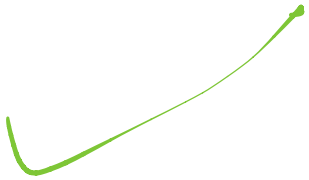
→ map(last, l)

```
fun map (f: (int list) -> int, l: (int list) list): int list
fun map (f: int -> int, l: int list): int list =
  case l of
    [] => []
  | x::xs => f(x) :: map(f, xs)
```

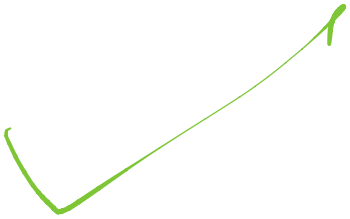
"Polymorphism": same code at different types

mark the parts of the type that can vary with type variables

```
map( last, [[1,2,3],[4,5]])
```



```
map( fn x => 2 * x, [1,2,3])
```



```
map(fn x => 2 * x, [[1,2,3],[4,5]])
```

"for any types 'a, 'b"

'a "type variable"

```
fun map ( f: 'a -> 'b , l: 'a list): 'b list =
    case l of
        [] => []
      | x :: xs => f(x) :: map(f, xs)
```

---

map (last, [[1,2,3],[4,5]]) ⟶ [3,5]

'a = int list

map (double, [1,2,3]) ⟶ [2,4,6]

'a = int

fun intToString (i : int) : string

intToString 4 = "4"

map (intToString, [1, 7]) = ["1", "7"]

intToString : int → string

[1, 7] : int list

["1", "7"] : string list

'a = int

'b = string

```
map : ('a -> 'b) * 'a list
    -> 'b list
```

placeholder/var for a type

---

lifting a function from 'a's to 'b's to 'a lists to 'b lists

Type $T_1 \rightarrow T_2$ (function type)

- int → string
- string → int
- int → int
- int list → int

(int → int) list

(int → int) * string