

# Lecture 13

Functions

---

as

inputs

# What is functional programming?

---

① value-oriented programming

② Functions as data  
as input to  
other functions

Avoid

repeated

code

→ add an input  
to  
abstract  
over a  
pattern

↳ recover

original repeats  
as instances

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

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

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

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)

Recover originals as instances:

fun add1(l: int list): int<sup>list</sup> = add(l, 1)

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

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

fun double(n: int)  
= 2 \* n

so  
double: int -> int

doubleAll  
[1, 2, 3]  
= [2, 4, 6]

# "Higher-order function"

---

Function that takes another function as input

Function type

int  $\rightarrow$  int  
input output

int \* (int  $\rightarrow$  int)

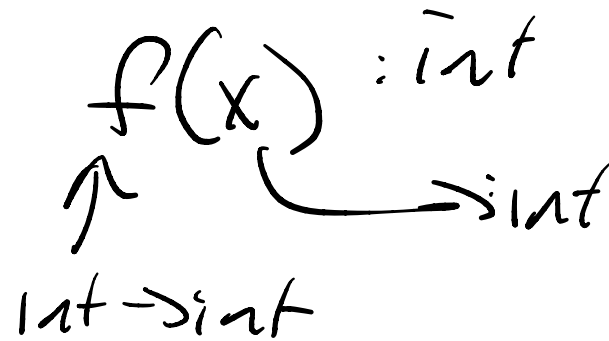
string  $\rightarrow$  int

int  $\rightarrow$  string  
:

$\text{int} \rightarrow \text{int}$

values       $\text{fun } f(x:\text{int}):\text{int} = \dots$

operations



"function application"



fun map (f: int → int, l: int list): int list =

case l of

[] ⇒ []

| x :: xs ⇒ f(x) :: map (f, xs)

Recall originals as instances:

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)

map (double, [1, 2, 3])

↳ case [1, 2, 3] of  
[] => []

) x :: xs => double(x) :: map (double, xs)

↳ double(1) :: map (double [2, 3])

↳ 2 :: \_\_\_\_\_

↳ . . . -

$$\text{map}(f, [x_1, x_2, x_3, \dots, x_{n-1}, x_n])$$

=

$$[f(x_1), f(x_2), f(x_3), \dots, f(x_{n-1}), f(x_n)]$$

fun add1num(x) = x + 1

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

*@ amount*

→ fun add1(l) =  
map(add1num, l)

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

*amount*

fun add2num(x) = x + 2  
→ fun add2(l) =  
map(add2num, l)

fun add(l: int list, a: int): int list =  
case l of

[] => []

| x::xs => (x+a)::add(xs, a)

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

fun adda(x: int) = x+a

in

map(  $\frac{\text{adda}}{\text{int} \rightarrow \text{int}}$  ) l

end

"closure"

```
fun add (l: int list, a: int): int list =  
  let  
  in  
  map(  $\frac{\text{adda}}{\text{int} \rightarrow \text{int}}$  l )  
end
```

$\text{adda}(x: \text{int}) = x + a$

$\text{add}([1, 2, 3], 2)$

$\mapsto \text{let fun adda}(x) = x + 2$   
in  
end map(adda, [1, 2, 3])

$\text{add}([1, 2, 3], 7)$

$\mapsto \text{let fun adda}(x) = x + 7$   
in  
end map(adda, [1, 2, 3])

# Anonymous function

alternative  
to local named

## Idea

values     $\text{int} \rightarrow \text{int}$

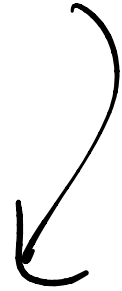
fn  $x:\text{int} \Rightarrow e$     free variable in  $e$

↳ "function"

fun double (x) = x \* 2

named  
helper

fun doubleAll (l) = map(double, l)



fun doubleAll (l) =

anonymous

map ( fn x => x \* 2 , l )

input      body



Job All ([1, 2, 3])

↳ map(fn x => x \* 2, [1, 2, 3])

↳ ~~(fn x => x \* 2)~~ 1 :: map(fn x => x \* 2, [2, 3])

↳ 1 \* 2 :: \_\_\_\_\_

↳ 2 :: (fn x => x \* 2) 2 :: map(\_\_\_\_\_, [3])

$\text{fn } x \Rightarrow e$  has type  $\text{int} \rightarrow \text{int}$

when assuming  $x : \text{int}$   
 $e : \text{int}$

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

Steps  $\rightarrow e$  with  $v$  for  $x$

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

*named helper*  
fun adda (x: int) = x + a

in

map( adda ) l)

end

fun add (l, a) =

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

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

fun add(l, a) =  
 map(fn x => x+a, l)

fun doubleAll(l) =  
 case l of  
 [] => []  
 | x::xs => double(x)::doubleAll(xs)

fun doubleAll(l) =  
 map(fn x => 2\*x, l)

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

last [1, 2, 3, 12, 4] = [4]

lasts ([[1, 2, 3],  
 (4, 5, 6)])  
 = [3, 6]

"Polymorphism": code that can work for any type  
 for any type 'a, 'b

fun map(f: 'a -> 'b, l: 'a list): 'b list =

case l of

[] => []

| x :: xs => f(x) :: map(f, xs)

↳ Double, add ... still work 'a = int

?

fun lasts(l) = map(last, l)

:: (int list) list      :: int list

(int list) -> int      (int list) list

'a = int list  
 'b = int

"for any types 'a and 'b"

fun zip(l1: ~~int~~<sup>'a</sup> list, l2: ~~string~~<sup>'b</sup> list): (~~int~~<sup>'a</sup> \* ~~string~~<sup>'b</sup>) list =

case (l1, l2) of

([], \_) => []

| (\_, []) => []

| (x::xs, y::ys) => (x, y)::zip(xs, ys)

zip(["a", "b"], [1, 2]) = (["a", 1], ...)

*Annotations:*  
- <sup>'a = string</sup>  
- <sup>'b = int</sup>  
- string list  
- int list (circled)  
- (string \* int) list (circled)

fun pluralize (l: string list) =

e.g. pluralize (["cat", "dog"]) = ["cats", "dogs"]

map<sub>a = string</sub> (fn x => x ^ "s", l)

any

for any type 'a, 'b, 'c, 'd

fun map(f: 'a -> 'b, l: 'c list): 'd list =

not

case l of

() => []

| x::xs => f(x)::map(f, xs)



map : (int -> string) \* ~~int~~ list

'a = int

'b = string

'c = bool

'd = real

→ string ~~int~~ list