Lecture 13

Functions as inputs

What is functional programming?

1. value-oriented programming

2. Functions as data
   - as input to other functions

Avoid

repeated → add an input

code

to

abstract

over a

pattern

→ recover

original repeats

as instances

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

add [1,2,3]

=

[2,3,4]

```
fun add2(l:int list):int list =
    case l of
        [] => []                     amount
      | 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 list = add(l, 1)
    fun add2(l:int list):int list = add(l, 2)
```

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

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

so double: int->int

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

```
   fun doubAll(l:int list):int list=
      case l of
(3)      [] => []
       | x::xs => double(x) :: doubAll(xs)
```

doubAll
   [1,2,3]

=

[2,4,6]

"Higher-order function"

Function that takes another function as input

Function type

$$\underbrace{\text{int}}_{\text{input}} \to \underbrace{\text{int}}_{\text{output}}$$

$\text{int} * (\text{int} \to \text{int})$

$\text{string} \to \text{int}$

$\text{int} \to \text{string}$

$\vdots$

int → int

Values fun f(x:int):int = - - - .

Operations

$$\underbrace{f}_{\text{int} \to \text{int}}(\underbrace{x}_{\text{int}}) : \text{int}$$

"function application"

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

Recover 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 l
   =
     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 :: ______________
↦ . . . _
```

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

$$=$$

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

```
fun add1num(x) = x+1
```

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

```
——> fun add1(l) =
      map(add1num, l)
```

```
fun add2num(x) = x+2
```

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

```
——> 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           ):int = x+a
in
   map( adda , l)
        int -> int
end
```

"closure"

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

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

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

Anonymous function

alternative to local <u>named</u> helper functions

<u>Idea</u>

<u>values</u> int → int

free variable in e

$$\underline{\text{fn}}\ \underline{x}:\text{int} \Rightarrow e$$

↳ "function"

fun double(x) = x * 2

named helper

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

↓

anonymous

fun doubAll(l) =

$$\text{map}(\;\text{fn}\ \underset{\substack{\uparrow\\ \text{input}}}{x} \Rightarrow \underbrace{x*2}_{\text{body}}\;,\ l)$$

doubAll ([1, 2, 3])

$\mapsto$ map(fn x => x * 2, [1, 2, 3])

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

$\mapsto$ 1 * 2 :: ______

$\mapsto$ 2 :: (fn x => x * 2) 2 :: map(______, [3]) fn $x \Rightarrow e$ has type $int \rightarrow int$

when assumig $x : int$

$e : int$

and $(fn\ x \Rightarrow e)\ v$

steps to e with v for x

```
fun add (l: int list, a: int): int list =
let
                    named helper
    fun adda( x: int          ): int = x+a
in
    map( adda , l)
         int -> int
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 doubAll(l) =
  case l of
    [] => []
  | x::xs => double(x)::doubAll(xs)
```

→

```
fun doubAll(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)
```

: int list

↪ doubAll, add .... still work { 'a = int }

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

: (int list) list

: int list

(int list) -> int

(int list) list

: int list

'a = int list

'b = int

"for any types 'a and 'b"

```
fun zip(l1 : ~~int~~ 'a list, l2 : ~~strg~~ 'b list) : (~~int~~ 'a * ~~str~~ 'b) list=
    case (l1, l2) of
        ([], _) => []
      | (_, []) => []
      | (x::xs, y::ys) => (x,y) :: zip(xs, ys)
```

'a = string
'b = int

string list   int list   (string * int) list

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

```
fun pluralize(l: string list) =

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

map^ ( fn x => x ^ "s", l )
  'a = string
```

any

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

not

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

map : (int -> string * ~~bool~~ int list) -> ~~real~~ string list

'a = int

'b = string

'c = bool

'd = real