Lecture 13:

Single point of control!

avoid copy and paste!

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

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

```
fun double(x:int):int = 2*x
fun doubAll(l:int list):int list =
    case l of
      [] => []
    | x::xs => double(x)::doubAll(xs)
```

Higher-order functions

Functions as inputs to other functions!

type

int → int

int list → int

int → string

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

```
fun double(x:int):int = 2*x        double: int -> int
fun doubAll(l:int list):int list =
    case l of
        [] => []
      | x::xs => double(x)::doubAll(xs)
```

↙ what to do to each element

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

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

(f: x => 2*x)

: int -> int

type $T_1 \to T_2$

values functions

operations apply it to a T_1

to get a T_2

$$\left.\begin{array}{l} f: T_1 \to T_2 \\ a: T_1 \end{array}\right] \; f(a) : T_2$$

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

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])

↦ 2:: map(double, [2,3])

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

```
fun add1int(x:int): int = x+1
fun add1(l:intlist): intlist = map(add1int, l)
```

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

```
fun add2int(x) = x+2
fun add2(l) = map(add2int, 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 addint(x:int):int = x+a
  in
    map(addint, l)
  end
```

"closure"

makes a new function every time add is run

add([1,2,3], 1) should be [2,3,4]

⊢ let fun addint(x) = x+1

in

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

end

⊢* [2,3,4]

add([1,2,3], 2) should be [3,4,5]

⊢ let fun addint(x) = x+2

in

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

end

⊢*

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

local decl.

Anonymous functions [not recursive]

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

expression.

$(fn\ x \Rightarrow e)$ is a value of type $T_1 \to T_2$

if for $x : T_1$, $e : T_2$

$(fn\ x \Rightarrow e)\ v$ steps to e with v substituted for x

i.e.

```
let fun f(x) = e in
    f(v)
end
```

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

↦ map(fn x => x + 1, [1,2,3])

↦ case [1,2,3] of

[] => []

| y :: xs => (fn x => x+1) y

:: map(fn x => x+1, xs)

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

↦ 1+1 :: ———

↦ 2 :: ———

Function as inputs + polymorphism

```
fun allLasts(l:(int list) list) : int list
    case l of
        [] => []
      | l1 :: ls => last(l1) :: allLasts(ls)

fun last(l:int list) :int = ...
```

e.g. last [1, 2, 3 4] = 4

all lasts [[1,2,3], [4,5,6]] = [3,6]

↙ what to do to each element

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

```
fun allLasts(l: (int list) list): int list =
    map(last, l)
```

↳ int list → int

"higher order"

map : $('a \to 'b) * 'a\ list$

$\longrightarrow 'b\ list$

```
fun hasEven(l:int list) : bool =
    case l of
        [] => false
      | x::xs => case evenP(x) of
                   true => true
                 | false => hasEven(xs)
```

evenP(x) orelse hasEven(xs)

hasEven([1,2,3]) = true

hasEven([1,3,5]) = false

andalso

```
fun hasA (l: string list) : bool =
    case l of
        [] => false
      | x::xs => x = "A" orelse hasA(xs)
```

```
fun hasA(l) = exists(fn x => x = "A", l)
```

```
fun hasEven'(l: int list) : bool =
    case l of
        [] => false
      | x::xs => evenP(x) orelse hasEven(xs)
```

```
fun hasEven(l) = exists(evenP, l)
```

```
fun exists( check: 'a -> bool,
            l: 'a list) : bool =
case l of
     [] => false
   | x::xs => check x orelse
              exists (check, xs)
```