# Datatypes, Trees and Structural Induction

15-150 Fall 2026

Lecture 6 – Part I

Dilsun Kaynar

# Recall: `minL` using type `extint`

```
datatype extint = PosInf | NegInf | Finite of int


(* minL: int list -> extint  *)


fun minL ([]:int list): extint = PosInf
  | minL (x::xs) = (case minL (xs) of
                      PosInf => Finite(x)
                    | Finite (y) => Finite(Int.min(x,y))
                    | _ => raise Fail "unreachable")
```

# Recall: `minL` using type `extint`

```
datatype extint = PosInf | NegInf | Finite of int


(* minL: int list -> extint  *)


fun minL ([]:int list): extint = PosInf
  | minL (x::xs) = (case minL (xs) of
                      PosInf => Finite(x)
                    | Finite (y) => Finite(Int.min(x,y))
                    | NegInf => NegInf )
```

# Options

SML provides the type `int option` with the following constructors:

```
NONE : int option

SOME : int -> int option
```

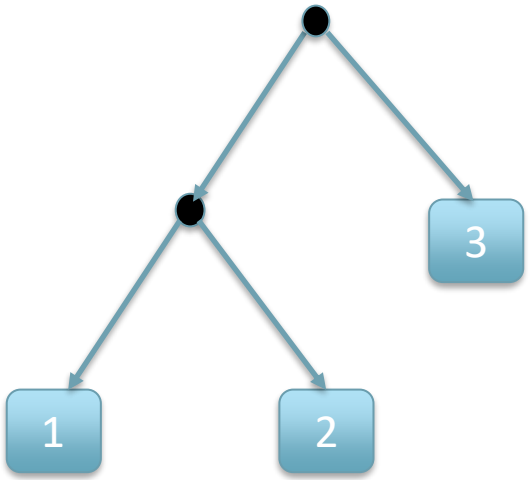
# `minL` using type `int option`

```
(* minL : int list -> int option  *)

fun minL ([]:int list): int option = NONE
  | minL (x::xs) = case minL (xs) of
                     NONE => SOME x
                    |SOME y => SOME (Int.min(x,y))
```

# Leafy trees

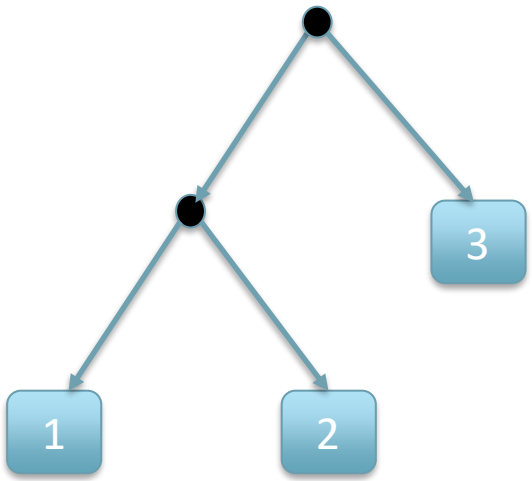
```
datatype tree = Leaf of int | Node of tree * tree
```



# flatten

```
datatype tree = Leaf of int | Node of tree * tree

(* flatten : tree -> int list
   REQUIRES: true
   ENSURES: flatten(t) returns a list of the leaf
            values as they are encountered in the
            inorder traversal of t
*)
```




[1,2,3]

# flatten

```
datatype tree = Leaf of int | Node of tree * tree

(* flatten : tree -> int list
   REQUIRES: true
   ENSURES: flatten(t) returns a list of the leaf
            values as they are encountered in the
            inorder traversal of t
*)


fun flatten (Leaf(x) : tree) : int list = [x]
  | flatten (Node(t1, t2)) = flatten (t1) @ flatten (t2)
```

# `flatten` with accumulator

```
(* flatten2 : tree * int list -> int list
   REQUIRES: true
   ENSURES: flatten2(t, acc) ≅ flatten(t) @ acc
*)

fun flatten2 (Leaf(x), acc) = x :: acc
  | flatten2 (Node(t1,t2), acc) =
                    flatten2(t1,(flatten2(t2,acc)))

fun flatten' (t: tree) : int list =
                                flatten2(t,[])
```

# `flatten` with accumulator

```
(* flatten2 : tree * int list -> int list
   REQUIRES: true
   ENSURES: flatten2(t, acc) ≅ flatten(t) @ acc
*)

fun flatten2 (Leaf(x), acc) = x :: acc
  | flatten2 (Node(t1,t2), acc) =
               flatten2(t1,(flatten2(t2,acc)))
```

Is `flatten2` tail recursive?

# Correctness of `flatten2`

**Theorem:** For all values `T : tree` and `acc : int list`,

`flatten2(t,acc) ≅ flatten(t)@ acc`.

**PLEASE READ THE NOTES**

# Auxiliary results

```
fun flatten2 (Leaf(x), acc) = x :: acc
  | flatten2 (Node(t1,t2), acc) =
               flatten2(t1,(flatten2(t2,acc)))
```

```
fun flatten (Leaf(x) : tree) : int list = [x]
  | flatten (Node(t1, t2)) = flatten (t1) @ flatten (t2)
```

**Lemma 1:** For all values `x : int` and `acc: int list`,

`x :: acc ≅ [x] @ acc`.

**Lemma 2:** The function `flatten` is total.

**Lemma 3:** The function `@` is total and associative.

# Proof

```
fun flatten2 (Leaf(x), acc) = x :: acc
  | flatten2 (Node(t1,t2), acc) =
               flatten2(t1,(flatten2(t2,acc)))
```

```
fun flatten (Leaf(x) : tree) : int list = [x]
  | flatten (Node(t1, t2)) = flatten (t1) @ flatten (t2)
```

By structural induction on `T`.

**Base case:** `T = Leaf(x)`

**NTS.** For all `acc:int list`,

`flatten2(Leaf(x), acc)` ≅ `flatten((Leaf(x)) @ acc.`

**Showing.**

`flatten2(Leaf(x), acc)` ≅ `x :: acc` [1st clause `flatten2`]

≅ `[x] @ acc` [Lemma 1]

≅ `flatten(Leaf(x)) @ acc`

[1st clause of `flatten`]

# Inductive step

**Inductive step:** `T = Node(t1,t2)`

**I.H.** For all `acc1:int list,`

`flatten2(t1, acc1) ≅ flatten(t1) @ acc1`

For all `acc2: int list`

`flatten2(t2, acc2) ≅ flatten(t2) @ acc2`

**NTS.** For all `acc: int list,`

`flatten2(Node(t1,t2), acc) ≅ flatten(Node(t1,t2)) @ acc`

# Inductive step

Showing.

```
   flatten2(Node(t1,t2), acc))
≅ flatten2(t1, flatten2(t2,acc))
```
[2nd clause of `flatten2`]

# Inductive step

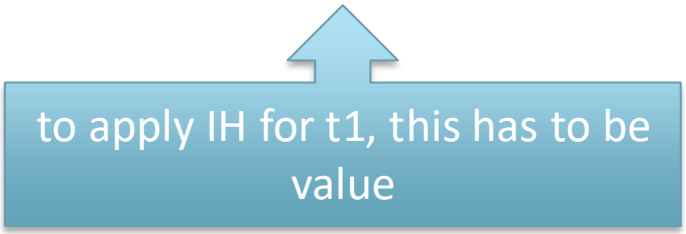
Showing.

```
   flatten2(Node(t1,t2), acc))
≅ flatten2(t1, flatten2(t2,acc))   [2nd clause of flatten2]
≅ flatten2(t1, flatten(t2) @ acc)  [I.H. for t2]
```

# Inductive step

Showing.

```
   flatten2(Node(t1,t2), acc))
≅ flatten2(t1, flatten2(t2,acc))   [2nd clause of flatten2]
≅ flatten2(t1, flatten(t2) @ acc)  [I.H. for t2 ]
```



# Inductive step

Showing.

```
   flatten2(Node(t1,t2), acc))
≅ flatten2(t1, flatten2(t2,acc))   [2nd clause of flatten2]
≅ flatten2(t1, flatten(t2) @ acc)  [I.H. for t2]
≅ flatten(t1) @ ((flatten(t2) @ acc))
                                   [I.H. for t1 , Lemmas 2 and 3 ]
```

# Inductive step

Showing.

```
   flatten2(Node(t1,t2), acc))
≅ flatten2(t1, flatten2(t2,acc))   [2nd clause of flatten2]
≅ flatten2(t1, flatten(t2) @ acc)  [I.H. for t2]
≅ flatten(t1) @ ((flatten(t2) @ acc))
                                   [I.H. for t1 , Lemmas 2 and 3 ]
≅ (flatten(t1) @ flatten(t2)) @ acc  [Lemma 3 ]
```

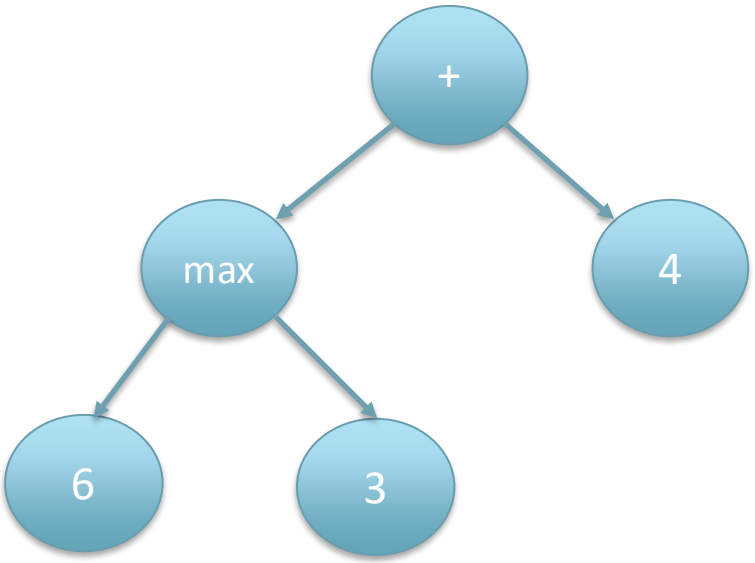
# Inductive step

Showing.

```
   flatten2(Node(t1,t2), acc))
≅ flatten2(t1, flatten2(t2,acc))   [2nd clause of flatten2]
≅ flatten2(t1, flatten(t2) @ acc)  [I.H. for t2]
≅ flatten(t1) @ ((flatten(t2) @ acc))
                                   [I.H. for t1 , Lemmas 2 and 3 ]
≅ (flatten(t1) @ flatten(t2)) @ acc  [Lemma 3 ]
≅ (flatten(Node(t1,t2)) @ acc  [2nd clause of flatten ]
```

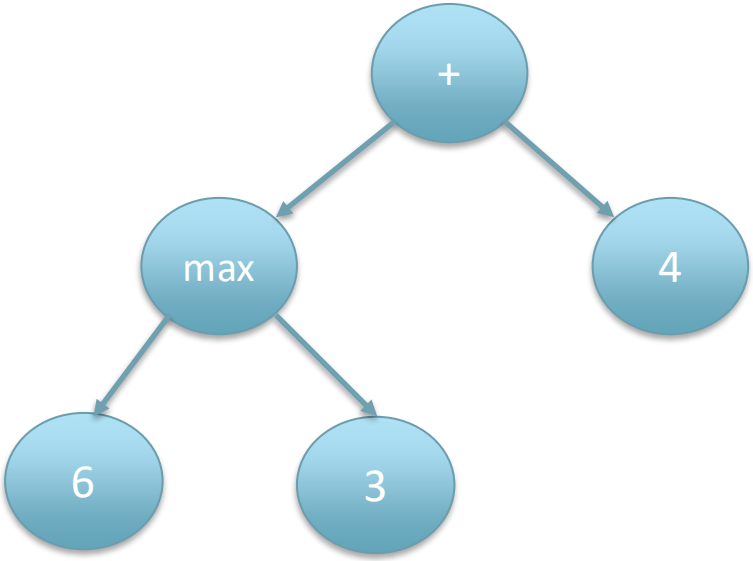
# Another kind of tree

```
(Int.max(6,3)) + 4
```



# Operator/operand tree

```
datatype optree = Op of optree * (int * int -> int) * optree
                | Val of int
```




Could also write `(fn (x,y)=> x+y)`

```
Op(Op(Val 6,Int.max,Val 3), (op +) ,Val 4)
```

# Operator/operand tree

```
datatype optree = Op of optree * (int * int -> int) * optree
                | Val of int

(* eval : optree -> int
   REQUIRES: all functions in T are total
   ENSURES: eval(T) reduces to the integer value that is the
            result of the computation
            described by T (assuming post-order traversal)
*)


fun eval(Val x : optree ) : int = x
   |eval(Op(l,f,r)) = __________
```

# Operator/operand tree

```
datatype optree = Op of optree * (int * int -> int) * optree
                | Val of int

(* eval : optree -> int
   REQUIRES: all functions in T are total
   ENSURES: eval(T) reduces to the integer value that is the
            result of the computation
            described by T (assuming post-order traversal)
*)


fun eval(Val x : optree ) : int = x
   |eval(Op(l,f,r)) = f(eval l, eval r)
```