Datatypes, Trees and Structural Induction

15-150 Fall 2026

Lecture 5

Dilsun Kaynar

So far in the course

- Basic ML programming
  - Write well-typed functions with recursion
  - Aggregate data structures such as tuples and lists
- Specifications
- Proofs
  - Reasoning with evaluation and equivalence
  - Simple and strong induction
  - Structural induction

Today

- How to define your own types (recursive/non-recursive) using **`datatype`** declarations
- Represent trees and compute with them in ML
- More specifications and proofs

Synonyms for existing types

```
type pair = int * int
```

Declaring your own types

DATATYPES

Example: comparing integers

```
x < y : bool
x > y : bool
x = y : bool
```

```
compare: int * int -> ____
```

Datatype declaration

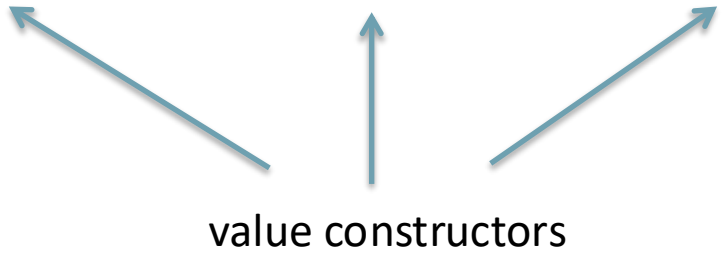
```
x < y          LESS
x > y          GREATER
x = y          EQUAL
```

```
datatype order = EQUAL | LESS | GREATER
```

```
Int.compare: int * int -> order
```

Introduces a **new** type that is distinct from all other types

```
datatype order = EQUAL | LESS | GREATER
```




```
EQUAL : order
LESS : order
GREATER : order
```

Pattern matching

```
(case (Int.compare (x,y))of   =
        LESS => …
      | EQUAL  => …
      | GREATER => …)
```

Example: total function `minL`

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

fun minL ([]:int list): ________ = __________
```

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

Examples:

```
PosInf : extint
Finite : int -> extint


[PosInf, Finite (~3)] :  extint list
```

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


(* minL: int list -> extint  *)


fun minL ([]:int list): extint = _______
  | minL (x::xs) = ________________
```

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


(* minL: int list -> extint  *)


fun minL ([]:int list): extint = PosInf
  | minL (x::xs) = ________________
```

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


(* minL: int list -> extint  *)


fun minL ([]:int list): extint = PosInf
  | minL (x::xs) = (case minL (xs) of

                                                  )
```

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


(* minL: int list -> extint  *)


fun minL ([]:int list): extint = PosInf
  | minL (x::xs) = (case minL (xs) of

                                                  )
```

```
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 (y) => _______
                    | _ => _________________ )
```

```
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) =>_______
                    | _ => ________________)
```

```
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))
                    | _ => _________________)
```

```
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 how we defined lists

A list of integers is either

`nil` (also written as `[]`), or
`x :: xs` where `x : int` and `xs : int list`

You can think of it as the result of the following declaration:

```
datatype list = nil
              | :: of int * list
infixr ::
```

in reality it is polymorphic – we will learn about it in 2 weeks

Representing trees with datatypes

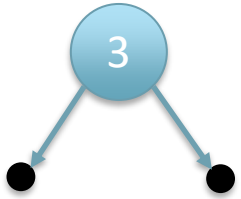
BINARY TREES

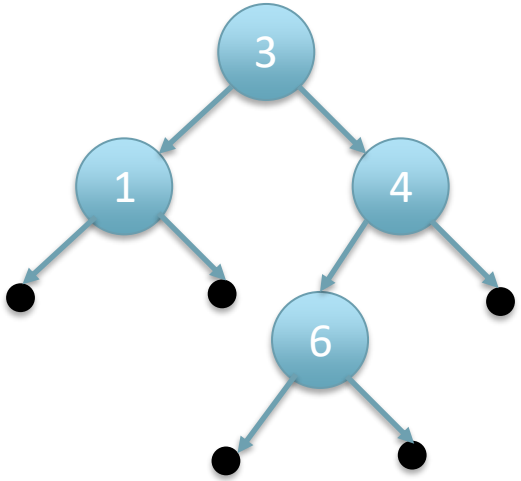
Examples

empty

Examples

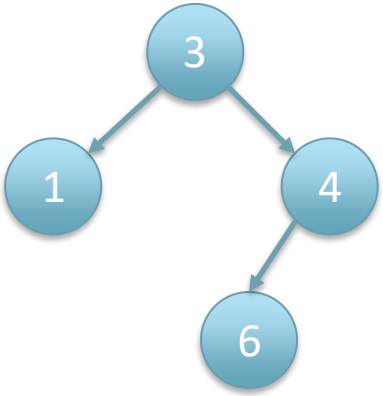
empty





Examples

empty



We sometimes don’t show the empty subtrees in our pictures

Recursive `datatype` declaration

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

Recursive `datatype` declaration

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

constant constructor

constructor that takes an argument

Recursive `datatype` declaration

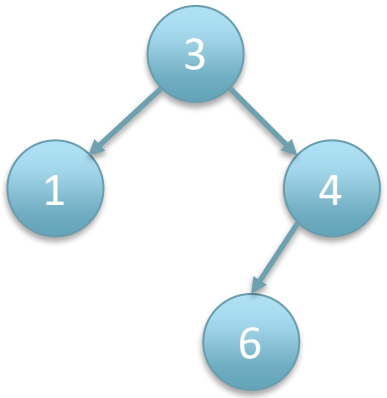
```
datatype tree = Empty | Node of tree * int * tree
```

A `tree` is

- either `Empty`
- or `Node(l,x,r)`
  where `l` is a `tree`, `x` is an `int` and `r` is a `tree`
- and that's it.

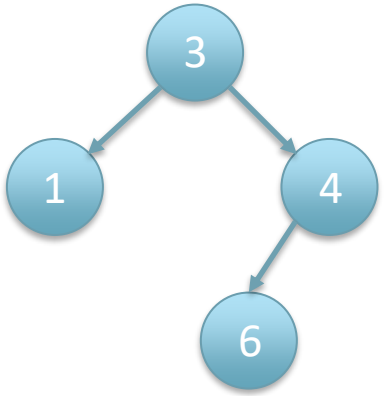
Recursive `datatype` declaration

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



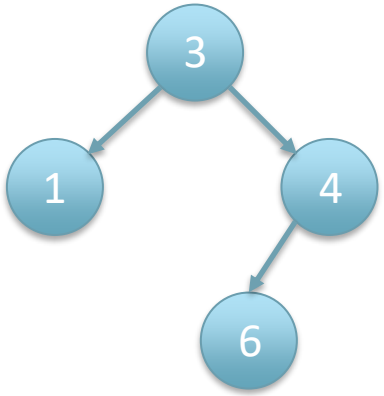

```
Node(Node(Empty,1,Empty), 3,
     Node(Node(Empty,6,Empty),4,
          Empty))
```

depth




```
(* depth : tree -> int
   REQUIRES: true
   ENSURES: depth returns the depth of t
            with depth(Empty) being 0
*)
```

depth




```
(* depth : tree -> int
   REQUIRES: true
   ENSURES: depth returns the depth of t
            with depth(Empty) being 0
*)

fun depth (Empty : tree) : int = 0
  | depth (Node(t1, x, t2)) = ___________
```

depth

```
(* depth : tree -> int
   REQUIRES: true
   ENSURES: depth returns the depth of t
            with depth(Empty) being 0
*)

fun depth (Empty : tree) : int = 0
  | depth (Node(t1, x, t2)) =
        1 + Int.max (depth(t1), depth(t2))
```

`depth` is total

`depth` is total

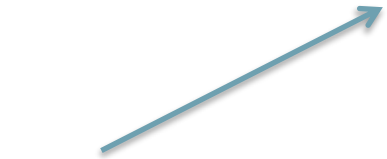
Theorem: For all values `t:tree`, `depth(t)` reduces to a value.

Theorem: For all values `t : tree`, `depth(t)` reduces to a value.

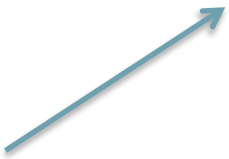
Proof: By structural induction on `t`.

Recursive datatype declaration

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



base case



recursive cases

Principle of Induction for trees

Theorem: For all `t: tree, P(t)`.

Proof: By structural induction on `t`.

Base case: `t = Empty`

Show `P(Empty)`

Inductive step: `t = Node (t1,x,t2)`

I.H. `P(t1)` **and** `P(t2)`

Show `P(Node (t1,x,t2))`

```
fun depth (Empty : tree) : int = 0
  | depth (Node(t1, x, t2)) =
        1 + Int.max (depth(t1), depth(t2))
```

Theorem: For all values `t:tree`, `depth(t)` reduces to a value.

Proof: By structural induction on `t`.

Base case: `t = Empty`

Need to show: `depth(Empty)` reduces to a value.

Showing:

```
fun depth (Empty : tree) : int = 0
  | depth (Node(t1, x, t2)) =
        1 + Int.max (depth(t1), depth(t2))
```

Theorem: For all values `t:tree`, `depth(t)` reduces to a value.

Proof: By structural induction on `t`.

Base case: `t = Empty`

Need to show: `depth(Empty)` reduces to a value.

Showing: `depth(Empty) => 0` [1st clause of depth]

```
fun depth (Empty : tree) : int = 0
  | depth (Node(t1, x, t2)) =
        1 + Int.max (depth(t1), depth(t2))
```

Theorem: For all values `t:tree`, `depth(t)` reduces to a value.

Proof: By structural induction on `t`.

Inductive case: `t = Node(t1,x,t2)` for some values `t1: tree, x: int, t2:tree`

IH: `depth(t1)` reduces to a value `v1` and `depth(t2)` reduces to a value `v2` .

Need to show: `depth(Node(t1,x,t2))` reduces to a value.

Showing: `depth(Node(t1,x,t2)) =>`
`1 + Int.max (depth(t1), depth(t2))`
[2nd clause of depth]

Theorem: For all values `t:tree`, `depth(t)` reduces to a value.

Proof: By structural induction on `t`.

Inductive case: `t = Node(t1,x,t2)` for some values `t1: tree, x: int, t2:tree`

IH: `depth(t1)` reduces to a value `v1:int` and `depth(t2)` reduces to a value `v2:int`.

Need to show: `depth(Node(t1,x,t2))` reduces to a value.

Showing: `depth(Node(t1,x,t2)) =>`

```
1 + Int.max (depth(t1), depth(t2))
```

[2nd clause of depth]

```
=> 1 + Int.max(v1 , depth(t2))
```

[IH for `t1`]

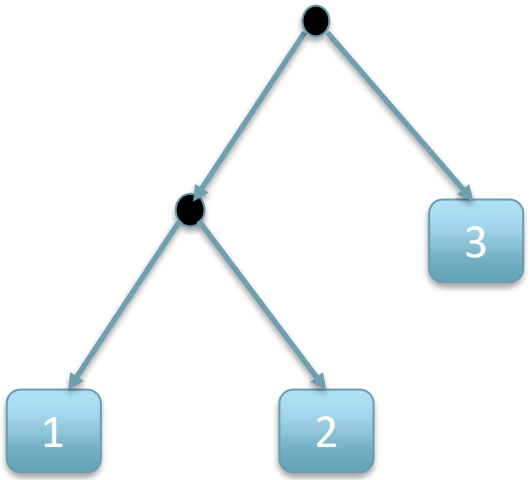
```
=> 1 + Int.max(v1 , v2)
```

[IH for `t2`]

ANOTHER KIND OF TREE

A new `datatype` for trees

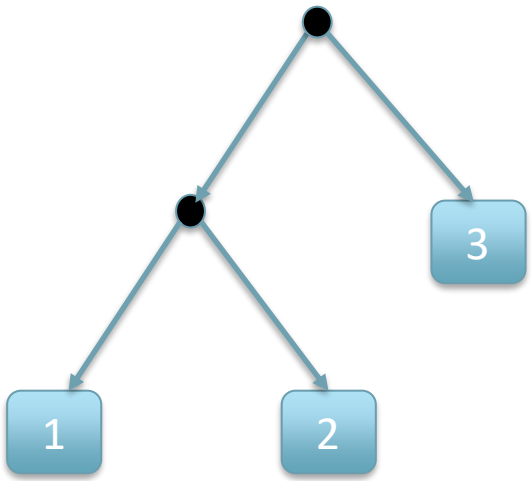
```
datatype tree = Leaf of int | Node of tree * tree
```



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 = ____________
```

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

```
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: …
*)
```

`flatten` with accumulator

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

`flatten` with accumulator

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

fun flatten2 (Leaf(x), acc) = _________
```

`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
```

`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 …
```

`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) = ___________
```

`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,[])
```

Correctness of `flatten2`

Theorem: For all values `T : tree` and `acc : int list`,
`flatten2(t,acc) ≅ flatten(t)@ acc`.

PLEASE READ THE NOTES

strengthening the specification (proving for all `acc`) is what *makes the induction go through*

`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`.

Code:

```
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)
```

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

By structural induction on `T`.

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

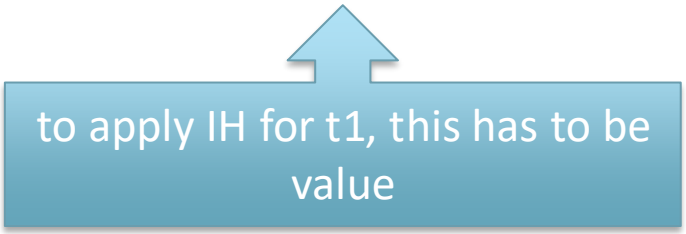
Showing.

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

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 and ref. trans.]
```



Inductive step

Showing.

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

[2nd clause of `flatten2`]

[I.H. for `t2` and ref. trans.]

[I.H. for t1 , Lemmas 2 and 3]

Inductive step

Showing.

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

[2nd clause of `flatten2`]

[I.H. for `t2` and ref. trans.]

[I.H. for t1 , Lemmas 2 and 3]

[Lemma 3]

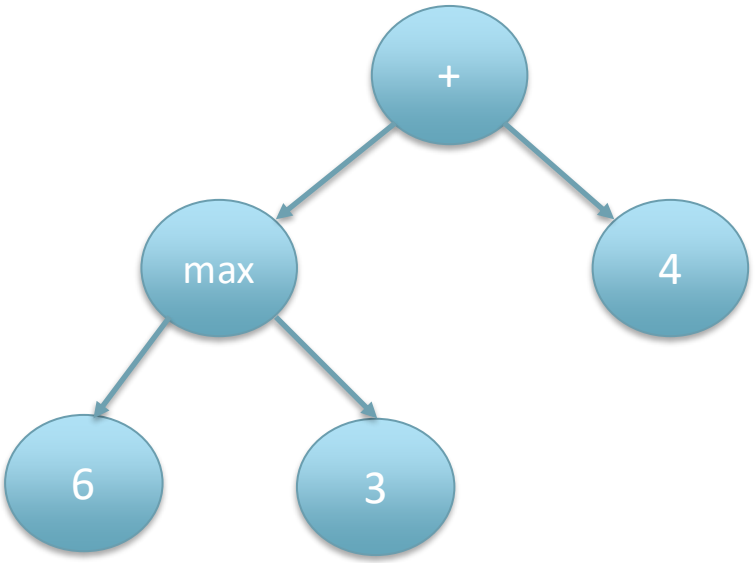
Inductive step

Showing.

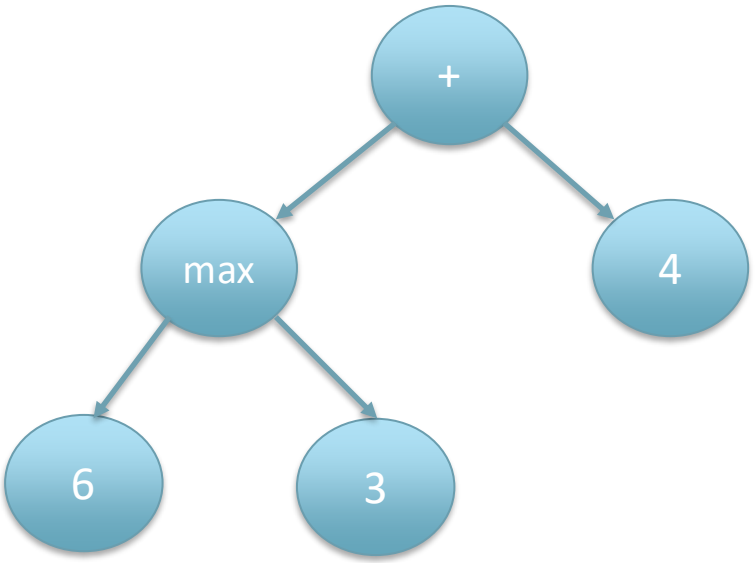
```
  flatten2(Node(t1,t2), acc))
≅ flatten2(t1, flatten2(t2,acc))   [2nd clause of flatten2]
≅ flatten2(t1, flatten(t2) @ acc)  [I.H. for t2 and ref. trans.]
≅ flatten(t1) @ ((flatten(t2) @ acc))
                                   [I.H. for t1 , Lemmas 2 and 3 ]
≅ (flatten(t1) @ flatten(t2)) @ acc 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

Another kind of tree

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



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




```
Op(Op(Val 6,Int.max,Val 3),
     (fn (x,y)=> x+y),
     Val 4)
```

Could also write `(op +)`

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)
```