Datatypes,
Trees and Structural Induction

15-150 Fall 2025

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

```
<        LESS
>        GREATER
=        EQUAL
```

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

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

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

Example: comparing integers

```
datatype order = EQUAL | LESS | GREATER

    Int.compare (3,4) ==> LESS
```

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

value constructors

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

Pattern matching

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

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

fun minL ([]:int list): _________ = ___________
```

```
(* minL : int list -> int
   REQUIRES: length(L) > 0
   ENSURES: minL(L) returns some x, where s is the smallest
            integer in L
*)

  fun minL ([]:int list): int = raise Fail "REQUIRES violated"
    | minL ([x]) = x
    | minL (x::xs) = Int.min(x, minL xs)
```

Can we re-define it so that it is total?

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

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

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

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

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

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

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

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

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

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

fun minL ([]:int list): extint = PosInf
  | minL (x::xs) = (case minL (xs) of
                        PosInf => __________
                      | Finite (y) => ______
                      | NegInf=> ___________ )
```

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

fun minL ([]:int list): extint = PosInf
  | minL (x::xs) = (case minL (xs) of
                        PosInf => Finite(x)
                      | Finite (y) =>_______
                      | NegInf=> ___________)
```

```
(* 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)
```

```
(* 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)
```

Recall how we defined lists

A list of integers is either

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

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

full truth in a few weeks

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

infixr ::
```

```
(* 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)
```

We are using the constructors `[]` and `::`, and `PosInf`, `NegInf`, and `Finite` in pattern matching.

Type `bool`?

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

```
datatype bool = true | false
```

Options

Type	`t option` for any type `t`
Values	`NONE` or `SOME v` for value `v : t`
Expressions	`SOME e` for with `e : t`

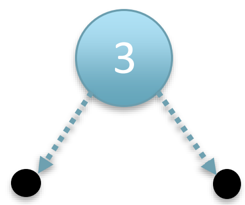
```
(* minL : int list -> int option
   REQUIRES: length (L) > 0
   ENSURES: minL(L) returns SOME x, where s is the smallest
            integer in L
*)
fun minL ([]:int list): int option = NONE
  | minL (x::xs) = case minL (xs) of
                     NONE => SOME x
                    |SOME y => SOME (Int.min(x,y))
```

Representing trees with datatypes

BINARY TREES

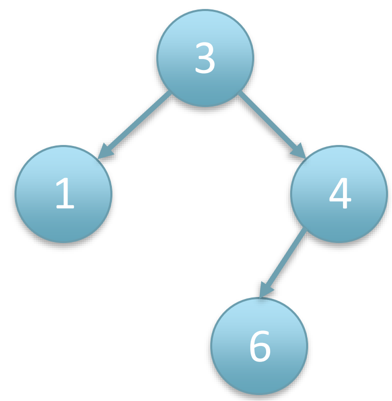
Examples

empty



Examples

empty



We sometimes don’t draw the empty tree explicitly.

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
```

constant constructor

constructor that takes an argument

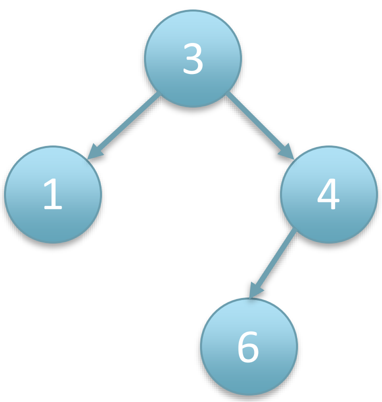
A `tree` is

- 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.

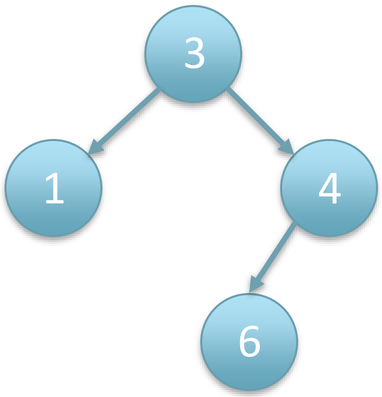
Recursive `datatype` declaration

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



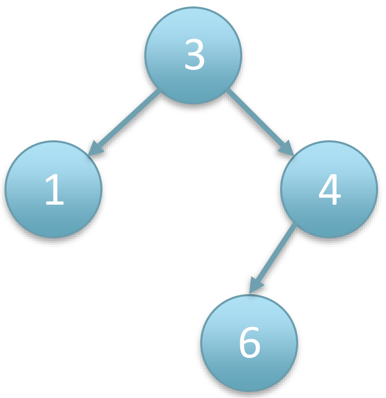

```
Node
  (Node(Empty,1,Empty), 3,
   Node(Node(Empty,6,Empty),4,Empty))
```

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 case

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

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: $t = \texttt{Node}\ (t_1, x, t_2)$

I.H. $P(t_1)$ and $P(t_2)$

Show $P(\texttt{Node}\ (t_1, x, t_2))$

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

```
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:

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

```
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]

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

```
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`.

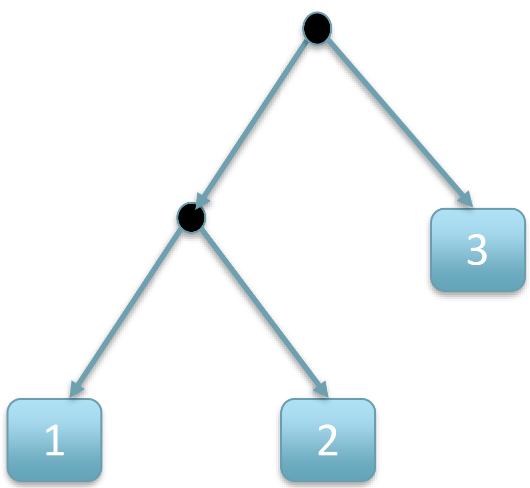
Need to show: `depth(Node(t1,x,t2))` reduces to a value.

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

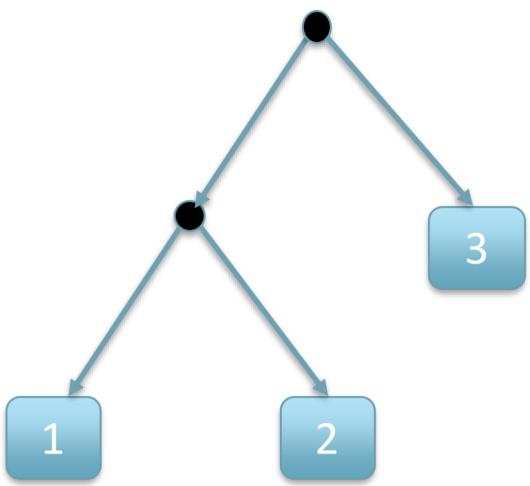
```
datatype tree = Leaf of int | Node of tree * tree
```



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
  | 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)))
```

Is `flatten2` tail recursive?

`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,[])
```

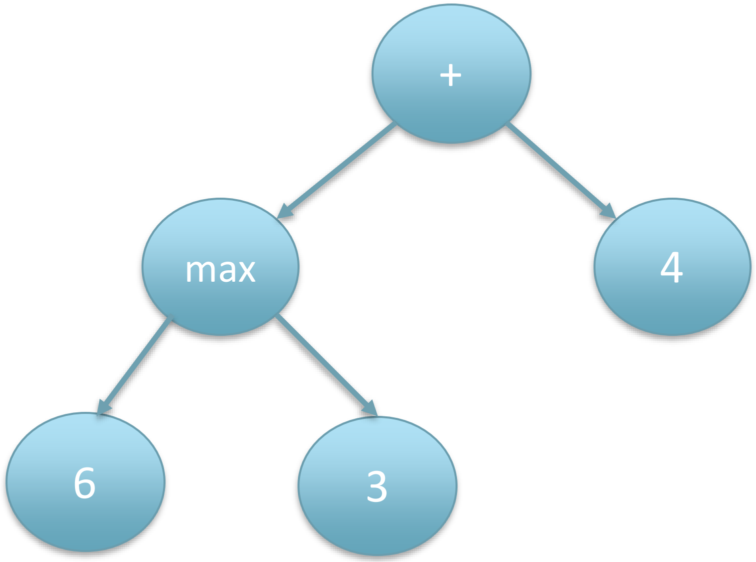
Correctness of `flatten2`

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

PLEASE READ THE NOTES

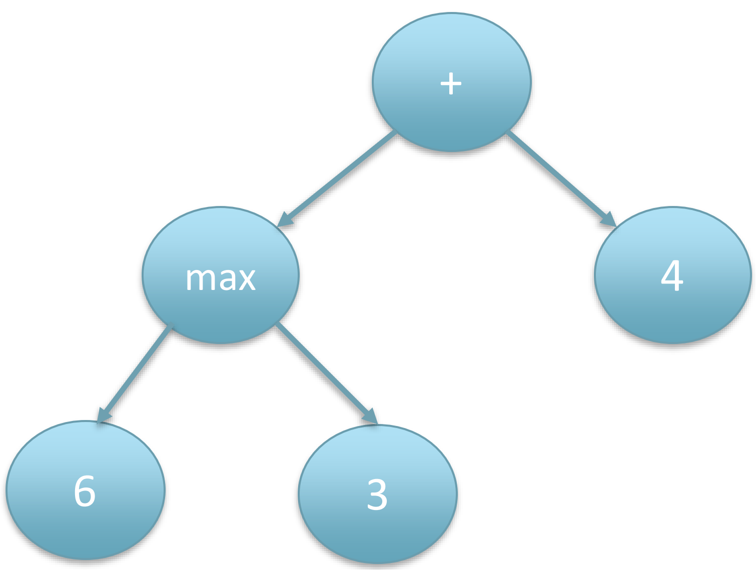
Another kind of tree

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



Operator/operand tree

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




```
Oper(Oper(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)
```