Datatypes

Data abstraction   e.g. type indexed = int * real   (7, 2.1) : indexed

e.g. datatype order = LESS | EQUAL | GREATER

constant constructors

values LESS : order
EQUAL : order
GREATER : order

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

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

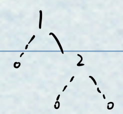
e.g. datatype bool = true | false

```
datatype extint = NegInf | Finite of int | PosInf
(* minlist : int list -> extint *)
fun minlist ([]: int list) : extint = PosInf
  | minlist (x::xs) =
    let
      fun tmin ([]: int list, acc : int) : int = acc
        | tmin (y::ys, acc) = tmin (ys, Int.min (y, acc))
    in
      Finite (tmin (xs, x))
    end
```

Value of type extint
NegInf
Finite 1
Finite ~17
PosInf
----

val Finite(M) = minlist L

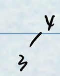
Finite is a constructor that expects values of type int & returns values of type extint.
Finite : int -> extint

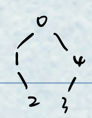
Empty tree 

Node with two subtrees & an integer

datatype tree = Empty | Node of tree * int * tree   Datatype can be recursive!

val t_1 = Node (Empty, 1, Node (Empty, 2, Empty)) -> t_1 is a value of type tree

val t_2 = Node (Node (Empty, 3, Empty), 4, Empty) 

val t = Node (t_1, 0, t_2) 


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

THM depth is total.

By structural on T. (N.T.S. depth (T) returns a value for all T : tree)

Base Case : T = Empty   N.T.S. depth Empty returns a value
=> 0 (clause 1)

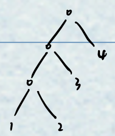
Inductive Step : T = Node (t_1, n, t_2) for some t_1 : tree, n : int, t_2 : tree
N.T.S. depth (T) returns a value

IH : depth (t_1) ⟶ d_1 value
depth (t_2) ⟶ d_2 value

Showing depth (Node (t_1, n, t_2))

⟹ 1 + Int.max (depth t_1, depth t_2) (clause 2)
⟹ 1 + Int.max (d_1, d_2) (IH)
⟹ 1 + v (some value v by totality of Int.max)
⟹ v' (some value v' by totality of +)

datatype tree = leaf of int | Node of tree * tree



```
val tr = Node (Node (Node (Leaf 1, leaf 2),
                     leaf 3),
               Leaf 4)
```

```
(* flatten : tree -> int list *)
fun flatten (leaf x : tree) : int list = [x]
  | flatten (Node (t1, t2)) = flatten (t1) @ flatten (t2)
```

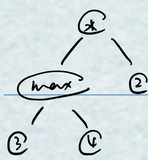
e.g. flatten (tr) ⟶ [1, 2, 3, 4]

O (n^2)

```
(* flatten2 : tree * int list -> int list
   REQUIRES: true
   ENSURES: flatten2 (T, acc) ≅ flatten (T) @ acc
*)
fun flatten2 (leaf x : tree, acc : int list) : int list = x :: acc
  | flatten2 (Node (t1, t2), acc) = flatten2 (t1, flatten2 (t2, acc))
```

→ Not tail recursive !
(call recursion twice)

datatype optree = Val of int | oper of optree * (int * int → int) * optree



oper (oper (Val 3, Int.max, Val 4), $\underbrace{\text{fn (x, y) => x * y}}_{\text{op *}}$, Val 2)

```
(* eval : optree -> int *)
fun eval (Val x) = x
  | eval (Oper (t1, f, t2)) = f (eval t1, eval t2)
```

Asymptotic Cost Analysis

(lemma : length (rev L) ≅ length L)

$W_{rev}(n) = c_1 + W_{rev}(n-1) + \underbrace{W_{@}(n-1, 1)}$, some c_1

$W_{rev}(n) \le c_1 + W_{rev}(n-1) + c_2(n-1)$, some c_2

Unrolling : $W_{rev}(n) \le \cdots$
$\le c_0 + n \cdot c_1 + (n(n+1)/2) \cdot c_2 \Rightarrow O(n^2)$

Analyze trev $W_{trev}(n, m)$ with m, n as sizes of input lists.

BC) $W_{trev}(0, m) = c_0$, for some c_0, all m

IS) $W_{trev}(n, m) = c_1 + W_{trev}(n-1, m+1)$, some c_1, all m

$= \cdots = n \cdot c_1 + c_0$

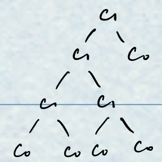
Analyze tree summation

```
fun sum Empty = 0
  | sum (Node(l, x, r)) = sum l + sum r + x
```

$W_{sum}(n)$ with n : # nodes

BC) $W_{sum}(0) = c_0$, for some c_0

Recursive case) $W_{sum}(n) = c_1 + W_{sum}(n_l) + W_{sum}(n_r)$ some c_1, with n_l : # nodes on the left, n_r : ··· right



$W_{sum}(n) = c_1 n + c_0(n+1) \Rightarrow O(n)$

A binary tree has n nodes iff it has n+1 leaves.

Opportunity for parallelism? $S_{sum}(n) = c_1 + \max\{S_{sum}(n_l), S_{sum}(n_r)\}$, some c_1

What if $n_l = n-1$, $n_r = 0$? Suppose the tree is balanced.

$$\begin{aligned} \Rightarrow S_{sum}(n) &= c_1 + \max\{S_{sum}(n/2), S_{sum}(n/2)\} \\ &= c_1 + S_{sum}(n/2) \\ &= c_1 + c_1 + S_{sum}(n/4) \\ &= \cdots = (\log_2 n)\, c_1 \quad \Rightarrow O(\log n) \end{aligned}$$

Express span as $S_{sum}(d)$, d: depth

$S_{sum}(0) = c_0$

$$\begin{aligned} S_{sum}(d) &= c_1 + \max\{S_{sum}(d-1), S_{sum}(d')\} \\ &= c_1 + S_{sum}(d_1) \qquad \Rightarrow O(d) \text{ holds for all trees!} \end{aligned}$$

Balance

$\begin{cases} \text{(i) empty} \\ \text{(ii) a Node whose 2 subtrees are balanced with depth differing} \le 1 \end{cases}$

Work at each level: $c_1, 2c_1, 4c_1, \ldots, 2^{d-2}c_1, 2^{d-1}c_0$

$W(n) = c_1(1 + \cdots + 2^{d-2}) + c_0 \cdot 2^{d-1} \le c2^d \Rightarrow O(n)$

$S(n) = c_1(1+1+\cdots+1) + c_0 \le cd \Rightarrow O(\log n)$ $\quad c = \max\{c_1, c_0\}$

Sorting

compare : $t * t \to$ order for some type t

e.g. Int.compare, String.compare

For lists: L is sorted iff compare$(x, y) \Rightarrow$ LESS / EQUAL whenever x appears to the left of y

```
(* ins : int * int list -> int list
     REQUIRES: L is sorted
     ENSURES: ins(x, L) => a sorted permutation of x::L
*)
```

don't delete / add any elements

```
fun ins (x, []) = [x]
  | ins (x, y::ys) = (case compare(x,y) of
                        GREATER => y :: ins(x, ys)
                      | _       => x :: y :: ys)
```

$W_{ins}(n)$ with n : list length

$\begin{cases} W_{ins}(0) = c_0 \\ W_{ins}(n) = c_1 + W_{ins}(n-1) & (\text{case 1}) \\ W_{ins}(n) = c_2 & (\text{case 2}) \end{cases}$

$\Rightarrow O(n)$

```
(* ins : int list -> int list
     REQUIRES: true
     ENSURES: isort(L) => a sorted permutation of L
*)
```

```
fun isort ([]) = []
  | isort (x::xs) = ins(x, isort xs)
```

$W_{isort}(0) = c_0$

$W_{isort}(n) = c_1 + W_{isort}(n-1) + W_{ins}(n-1)$

$W_{isort}(n) \le c_1 + c_2 \cdot n + W_{isort}(n-1) \quad \Rightarrow O(n^2)$