

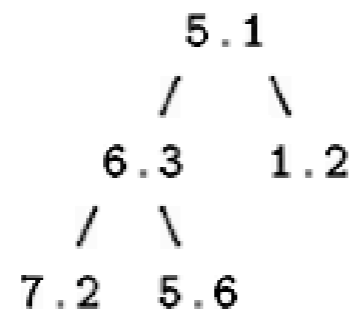
# *Midterm Review*

Echo Recitation • 21 February 2007

The following datatype represents binary trees whose nodes are either leaf nodes containing a single real value or nodes containing a real value along with two subtrees.

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

For instance, the following picture of a tree



would be represented by the following value of type `tree`:

```
Node(Node(Leaf 7.2, 6.3, Leaf 5.6), 5.1, Leaf 1.2);
```

```
(* val weight : tree -> real
   weight(t) ==> sum of all real values appearing in the nodes/leaves of t.
   invariants: none
   effects: none
*)
```

```
(* val weight : tree -> real
   weight(t) ==> sum of all real values appearing in the nodes/leaves of t.
   invariants: none
   effects: none
*)
```

## Now, tail recursively:

```
(* val weight2 : tree * real -> real
   weight2(t, acc) ==> weight(t) + acc
   invariants: none
   effects: none
*)
```

2. (15 points) Using structural induction prove that your implementation of `weight2` correctly returns the value stated in the specification.

Be sure to clearly state the theorem you are trying to prove, the base case, the inductive hypothesis, what you need to show in the induction step, and where you use the inductive hypothesis.

**Theorem:**

**Proof by structural induction on** \_\_\_\_\_.

**Base Case:**

**Induction Step:**

**Inductive Hypothesis:**

**Need to show:**

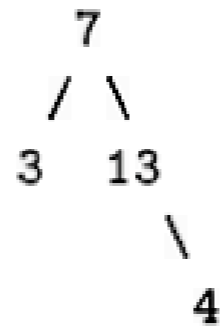
(and now show it)

```
datatype 'a tree = Empty | Node of 'a tree * 'a * 'a tree
```

For example, the value

```
Node(Node(Empty, 3, Empty), 7, Node(Empty, 13, Node(Empty, 4, Empty)))
```

represents the tree



### 3. Mapping Trees [5 Points]

Implement a function `tmap` : `('a -> 'b) -> 'a tree -> 'b tree` for trees that is analogous to the function `map` for lists. Specifically:

Suppose `f` : `'a -> 'b` and `t` : `'a tree`. Then `(tmap f t)` should return a new tree structurally equivalent to `t` but in which every node element `x` has been replaced by `f(x)`.

For example,

```
tmap (fn x => 2*x)
  (Node(Node(Empty, 3, Empty), 7, Node(Empty, 13, Node(Empty, 4, Empty))))
==> Node(Node(Empty, 6, Empty), 14, Node(Empty, 26, Node(Empty, 8, Empty)))
```

## 4. Folding Trees [3 Points]

Implement a function `tfold` : `('b * 'a * 'b -> 'b) -> 'b -> 'a tree -> 'b` that folds a given function over a tree. Specifically:

Suppose `f` : `'b * 'a * 'b -> 'b`, `b` : `'b`, and `t = Node(lt, x, rt)` : `'a tree`. Then `(tfold f b t)` should return `f(lv, x, rv)`, where `lv` and `rv` are the results of folding `f` over the left and right subtrees, `lt` and `rt`, respectively. Folding `f` over an `Empty` tree should return `b`.

```
signature RELATION =
sig
  type ('a, 'b) rel          (* abstract *)

  (* constant *)
  val empty: ('a, 'b) rel

  (* membership *)
  val contains : ('a, 'b) rel -> ('a * 'b) -> bool

  (* constructors *)
  val insert : ('a * 'b) * ('a, 'b) rel -> ('a, 'b) rel
  val fromList : ('a * 'b) list -> ('a, 'b) rel

  (* extractor *)
  val image : 'a * ('a, 'b) rel -> 'b list

  (* binary operations *)
  val union: ('a, 'b) rel * ('a, 'b) rel -> ('a, 'b) rel
  val compose : ('a, 'b) rel * ('b, 'c) rel -> ('a, 'c) rel
end
```

## 2.2 Functional Implementation [18 Points]

In your second implementation of `RELATION` you will represent relations by functions:

```
type ('a, 'b) rel = 'a -> 'b list
```

### Abstraction Function:

The value `f : 'a -> 'b list` represents the relation consisting of those pairs  $(x, y)$  such that `y` is in the list `f(x)`.

For example, the function `(fn 1 => ["a", "b"] | 2 => ["c"] | _ => nil)` represents the relation consisting of the set of pairs  $\{(1, "a"), (1, "b"), (2, "c")\}$ .

**Representation Invariants:** None

The structure `FunRel` on the next page ascribes opaquely to signature `RELATION`.

Please fill in the missing code.

```
structure FunRel :> RELATION =  
struct  
  
  type ('a, 'b) rel = 'a -> 'b list  
  
  fun empty _ = _____  
  
  (* The helper function inList is identical to the previous version. *)  
  fun inList ls x = [you do not need to re-implement this]  
  
  fun contains f (x, y) = _____  
  
  fun insert((a, b), f) = fn x => _____  
  
  fun fromList pairs =  
    fn x => map _____ _____
```

```
(* This helper function removes duplicates from a list.
   val remdup : 'a list -> 'a list *)

fun remdup nil = _____

  | remdup (y::ls) = if _____ then _____
                    else y::_____

fun image(x, f) = _____

fun union(f1, f2) = fn x => _____

fun compose(f1, f2) =
  fn x => foldr _____ nil _____

end (* structure FunRel *)
```