1/28 - PBT Cont: Generators, Combinators and Polymorphism

```
type bintree =
  | Leaf of int
  | Branch of bintree * bintree
```

Property of binary tree: flipping trees are not always equal. We can check this property using a testing harness.

We are writing a recursive function to generate the tree, using integer bounds on how deep the tree can go and the larges

```
let rec gen_bintree(leaf_bound : int) (n : int) : bintree =
  match n with
  | i when i <= 0 -> Leaf (Random.int leaf_bound)
  | _ -> Branch (gen_bintree leaf_bound (n - 1), gen_bintree leaf_bound (n - 1))
(*OR*)
let rec gen_bintree(leaf_bound : int) (n : int) : unit -> bintree =
  fun () ->
    if n <= 0
    then Leaf (Random.int leaf_bound)
    else Branch (gen_bintree leaf_bound (n - 1), gen_bintree leaf_bound (n - 1) ())
```

This will give us perfectly balanced trees where each branch is the same length.

Now we are designing a general structurally recursive function on binary trees.

```
let rec bintree_fun (bt : bintree) : ??? =
  match bt with
  | Leaf -> ... v ...
  | Branch (l, r) -> ... bintree_fun l ... bintree_fun r
```

This structure can be used to create a sum function:

```
type let rec sum_bintree (bt : bintree) : int =
  match bt with
  | Leaf v -> v
  | Branch l,r -> sum_bintree l + sum_bintree r
```

Entries - We might have trees with entries storing data. We might want to precompute the subsets

```
type entry = {
  tag : string;
  value : int
}
```

We can make a second version of sum where we perform a tree filter to find instances of a tag

```
let rec collect_tagged (q : string) (bt : bintree) : int list =
  match bt with
  | Leaf { tag; value } when tag = q -> [value]
  | Leaf _ -> []
  | Branch (l, r) -> collect_tagged q l @ collect_tagged q r

let rec gen_tag : unit -> string =
fun () -> string_of_it (Random.int 100)

let rec gen_entry : unit -> entry =
```

```
  fun () -> { tag = gen_tag () ; value = Random.int 100}

let rec gen_tagged_bintree (leaf_bound : int) (n : int) : unit -> tagged_bintree =
  fun () ->
    if n <= 0
    then Leaf (Random.int leaf_bound)
    else Branch (gen_bintree leaf_bound (n - (1 + Random.int 2)), gen_bintree (n - (1
+ Random.int 2)) leaf_bound (n - 1) ())
(* OR *)
tagged_bintree =
  fun () ->
    if n <= 0
    then Leaf (Random.int leaf_bound)
    else
      if Random.int n + 1 = 0 then Leaf(gen_entry ())
      else Branch (gen_bintree leaf_bound (n - (1 + Random.int 2)),
                   gen_bintree (n - (1 + Random.int 2)) leaf_bound (n - 1) ())
```

We are making this function higher-order in order to easily feed it into the forall function.

We can use the and keyword in order to put helper functions after the user functions

```
let rec sum (xs : int list) : int =
  match xs with
  | [] -> 0
  | x :: xs' -> x + sum xs'
```

We can combine a custom sum function with collect_tagged

```
let sum_tagged (q : string) (bt : tagged_bintree) : int = sum (collect_tagged q bt)
```