User-Defined Datatypes

```
- datatype color = Blue | Red | Yellow;
datatype color = Blue | Red | Yellow
- fun colorToString Blue = "blue"
= | colorToString Red = "red"
= | colorToString Yellow = "yellow";
val colorToString = fn : color -> string
- datatype color = Blue | Red | Yellow | Color of string;
datatype color = Blue | Color of string | Red | Yellow
- fun colorToString Blue = "blue"
= | colorToString Red = "red"
= | colorToString Yellow = "yellow"
= | colorToString (Color c) = c;
val colorToString = fn : color -> string
- colorToString Blue;
val it = "blue" : string
- colorToString (Color "magenta");
val it = "magenta" : string
- datatype 'a Option = None | Some of 'a;
datatype 'a Option = None | Some of 'a
- fun f None = []
= | f (Some x) = [x];
val f = fn : 'a Option -> 'a list
- datatype 'a List = Nil | Cons of 'a * 'a List;
datatype 'a List = Cons of 'a * 'a List | Nil
- fun toList Nil = []
= | toList (Cons(x,t)) = x :: toList t;
val toList = fn : 'a List -> 'a list
- toList (Cons(1, Cons(2, Cons(3, Nil))));
val it = [1,2,3] : int list
```

```
- datatype 'a Option = None | Some of 'a | More of 'a list;
datatype 'a Option = More of 'a list | None | Some of 'a
- fun optList None = []
= | optList (Some x) = [x]
= | optList (More l) = l;
val optList = fn : 'a Option -> 'a list
- datatype ('a, 'b) Or = A of 'a | B of 'b;
datatype ('a,'b) Or = A of 'a | B of 'b
- fun isA (A _) = true
= | isA _ = false;
val isA = fn : ('a,'b) Or -> bool
- fun isB x = not (isA x);
val isB = fn : ('a,'b) Or -> bool
- A 2;
val it = A 2 : (int,'a) Or
- B "foo";
val it = B "foo" : ('a,string) Or
- val l = [A 2, B "foo", A 3, A 4, B "bar"];
val l = [A 2,B "foo",A 3,A 4,B "bar"] : (int,string) Or list
- fun listA [] = []
= | listA (A x :: t) = x :: listA t
= | listA (_ :: t) = listA t;
val listA = fn : ('a,'b) Or list -> 'a list
- listA l;
val it = [2,3,4] : int list
- map (fn (A x) => x) (List.filter isA l);
stdIn:48.6-48.19 Warning: match nonexhaustive
          A x => ...

val it = [2,3,4] : int list
```

```
- datatype 'a binTree = EmptyTree
= | Node of 'a * 'a binTree * 'a binTree;
datatype 'a binTree = EmptyTree
                    | Node of 'a * 'a binTree * 'a binTree
- val t = Node(1, Node(2, EmptyTree, EmptyTree),
=          Node(3, EmptyTree, EmptyTree));
val t = Node (1,Node (2,EmptyTree,EmptyTree),Node (3,EmptyTree,EmptyTree))
  : int binTree
- exception EmptyBinTree;
exception EmptyBinTree
- fun left (Node(_,l,_)) = l
= | left _ = raise EmptyBinTree;
val left = fn : 'a binTree -> 'a binTree
- left t;
val it = Node (2,EmptyTree,EmptyTree) : int binTree
- fun right (Node(_,_,r)) = r
= | right _ = raise EmptyBinTree;
val right = fn : 'a binTree -> 'a binTree
- fun root (Node(x,_,_)) = x
= | root _ = raise EmptyBinTree;
val root = fn : 'a binTree -> 'a
- root t;
val it = 1 : int
- root (right t);
val it = 3 : int
- fun isEmpty EmptyTree = true
= | isEmpty _ = false;
val isEmpty = fn : 'a binTree -> bool
```

```
- fun size tree =
= if isEmpty tree then 0
= else 1 + size (left tree) + size (right tree);
val size = fn : 'a binTree -> int
- size t;
val it = 3 : int
- fun size (Node(_,l,r)) = size l + size r + 1
= | size _ = 0;
val size = fn : 'a binTree -> int
- local
= fun sz (Node (_,l,r)) n = sz l (sz r (n+1))
=   | sz _ n = n
= in
= fun size t = sz t 0
= end;
val size = fn : 'a binTree -> int
```

```
- fun toListPre EmptyTree = []
= | toListPre (Node(x,l,r)) = x :: toListPre l @ toListPre r;
val toListPre = fn : 'a binTree -> 'a list
- local
= fun tlp EmptyTree acc = acc
=   |  tlp (Node(x,l,r)) acc = x :: tlp l (tlp r acc)
= in
= fun toListPre tree = tlp tree []
= end;
val toListPre = fn : 'a binTree -> 'a list
- fun fromListPre [] = EmptyTree
= | fromListPre (x :: t) =
= let val l = length t div 2
=     val a = List.take(t,l)
=     val b = List.drop(t,l)
= in
= Node(x, fromListPre a, fromListPre b)
= end;
val fromListPre = fn : 'a list -> 'a binTree
- fromListPre [1,2,3,4,5];
val it = Node (1,Node (2,EmptyTree,Node #),Node (4,EmptyTree,Node #))
  : int binTree
- toListPre it;
val it = [1,2,3,4,5] : int list
```

```
- val l = natList 1000000;
val l = [1,2,3,4,5,6,7,8,9,10,11,12,...] : int list
- val big = fromListPre l;
val big = Node (1,Node (2,Node #,Node #),Node (500001,Node #,Node #))
  : int binTree
- size big;
val it = 1000000 : int
- size (left big);
val it = 499999 : int
- size (right big);
val it = 500000 : int
- isBalanced big;
val it = true : bool
- depth big;
val it = 20 : int
```

```
datatype 'a binTree = EmptyTree
   | Node of 'a * 'a binTree * 'a binTree;

fun size EmptyTree = 0
   | size (Node(_, l, r)) = 1 + size l + size r
```

```
datatype 'a binTree = EmptyTree
   | Node of 'a * 'a binTree * 'a binTree;

fun size EmptyTree = 0
   | size (Node(_, l, r)) = 1 + size l + size r

fun isBalanced EmptyTree = true
   | isBalanced (Node(_, l, r)) = ( abs(size l - size r) <= 1 )
                                  andalso (isBalanced l)
                                  andalso (isBalanced r);

local
    exception NotBalanced
    fun isB EmptyTree = 0
      | isB (Node(_, l, r)) =
          let
            val ls = isB l and rs = isB r
          in
            if abs (ls - rs) > 1 then raise NotBalanced else ls + rs + 1
          end
in
    fun isBalanced t = (isB t; true) handle NotBalanced => false
end

fun depth EmptyTree = 0
  | depth (Node(_, l, r)) = 1 + Int.max (depth l, depth r)
```

```
- datatype boolean = TRUE | FALSE | Not of boolean | And of boolean * boolean;
datatype boolean = And of boolean * boolean | FALSE | Not of boolean | TRUE
- val e = And (Not (And (TRUE, Not FALSE)), TRUE);
val e = And (Not (And (#,#)),TRUE) : boolean
- Control.Print.printDepth := 20;
val it = () : unit
- e;
val it = And (Not (And (TRUE,Not FALSE)),TRUE) : boolean
- fun eval TRUE = true
=   |  eval FALSE = false
=   |  eval (And (p,q)) = eval p andalso eval q
=   |  eval (Not p) = not (eval p);
val eval = fn : boolean -> bool
- eval e;
val it = false : bool
```