Advanced Programming (I00032)
Type constructor classes and kinds

Assignment 2

1 Type Constructor Classes

On Blackboard, Bb, you find a skeleton that provides useful definitions for this assignment. In the skeleton a type constructor class `Container` is defined:

```
class Container t where
    Cinsert   :: a (t a) → t a      | <        a
    Ccontains :: a (t a) → Bool     | <, Eq    a
    Cshow     ::   (t a) → [String] | toString a
    Cnew      :: t a
```

A container has elements of type `a`. There can be several implementations of `Container` with one uniform interface. A simple implementation of the container is just an unsorted list. A more advanced implementation is a binary search tree.

Give implementations of this container type constructor class using list `[]` and binary search `Tree`.

Test the correctness of you implementations by evaluating expressions such as:

```
Start = (Ccontains 3 c, Cshow c) where c = ..
```

2 Kinds

Given the following type definitions:

```
:: IntList   = Empty  | ConsInt Int IntList
:: List a    = Nil    | Cons a (List a)
:: Tree a b  = Leaf a | Node (Tree a b) b (Tree a b)
:: T1 a b    = C11 (a b) | C12 b
:: T2 a b c  = C2 (a (T1 b c))
:: T3 a b c  = C3 (a b c)
:: T4 a b c  = C4 (a (b c))
```

What is the *kind* of the following types: `IntList`, `List`, `List IntList`, `Tree`, `T1`, `T2`, `T3`, and `T4`?

3 Generic Printing

A generic based `show` function with continuations takes a value and a continuation, a list of strings, and produces a list of strings showing the generic representation of that value.

```
show :: a → [String] | show_ a
show a = show_ a []
```

```
class show_ a where show_ :: a [String] → [String]
```

```
instance show_ Int  where show_ i c = ["Int"  : toString i : c]
instance show_ Bool where show_ b c = ["Bool" : toString b : c]
instance show_ UNIT where show_ _ c = ["UNIT" : c]
```

The lecture discussed that it is a smart idea to add constructor names to the generic representation.

```
:: CONS  a  = CONS String a
:: ListG a :== EITHER (CONS UNIT) (CONS (PAIR a [a])) // generic type for list

fromList :: [a] → ListG a
fromList []     = LEFT  (CONS "Nil" UNIT)
fromList [a:as] = RIGHT (CONS "Cons" (PAIR a as))
```

Give the necessary instances of the class show_ in order to show the generic representation of lists – [a] –, trees – Tree a –, and tuples – (a,b) –.

4   Generic Parsing

Define a generic parser that transforms the list of strings generated by show to the original data type. The result of parsing is either Fail, or a Match with the result of parsing and the remaining input.

```
:: Result a = Fail | Match a [String]
class parse a :: [String] → Result a

instance parse UNIT where
    parse ["UNIT" : r]   = Match UNIT r
    parse _              = Fail
instance parse Int where
    parse ["Int", i : r] = Match (toInt i) r
    parse _              = Fail
```

Complete the class parse such that you are able to use them for elements of the types Int, Bool, (a,b), [a], and (Tree a). Do this via the generic representation, e.g. the show for a list transforms the list to type ListG using fromList. The corresponding parse function parse the generic representation, ListG, and transforms the result to a list by toList.

Deadline

The deadline for this exercise is September 16, 23:59h.