Advanced Programming (I00032)
Overloading and simple generics

Assignment 1

Preparation

Make sure to have the latest version of Clean on your machine. The current version of Clean can be found at `http://wiki.clean.cs.ru.nl/`. Study the lecture slides that can be found at Blackboard for the course named Geavanceerd Programmeren (advanced programming in Dutch).

1 Ordering by overloading

On Blackboard (Bb) you find a skeleton file, `skeleton1.icl`, of a program that provides useful definitions for this assignment. In this skeleton you will find a type `Ordering` and an infix operator $\gtrless$ to compare elements of a type. These are defined as:

```
::  Ordering = Smaller | Equal | Bigger

class (><) infix 4 a :: !a !a → Ordering
```

The skeleton also provides instances of this operator for some basic types. In addition, the skeleton defines a number of custom types (note that the `Tree` type definition is slightly different from the one used in the lecture!):

```
:: Color  = Red | Yellow | Blue
:: Tree a = Tip | Bin a (Tree a) (Tree a)
:: Rose a = Rose a [Rose a]
```

Define instances of the $\gtrless$ operator for these types (`Color`, `Tree a`, and `Rose`) as well as the standard Cleantype constructors (`a`,`b`) and [`a`]. Choose a convenient notion of an ordering relation in your definitions. For instance textual ordering for constructors and a generalization of lexicographical ordering for recursive types.

Test your implementation by evaluating expressions of the form:

```
Start = ([1..3] >< [1..2], [1..2] >< [1..5])
```

Include other elements your program to ensure that all instances of $\gtrless$ are tested.

2 Generic representation

The idea of generic programming is that we can save a lot of work by using a uniform representation of types. In this exercise we will use the same representation that is used in lecture 1:

```
:: UNIT         = UNIT
:: PAIR   a b   = PAIR a b
:: EITHER a b   = LEFT a | RIGHT b
```

In this representation the type `Rose a` is represented as:

```
:: RoseG a  :== PAIR a [Rose a]
```

1. Give generic representations for the types `Color` and `[a]` (the standard lists of Clean). Name these generic types `ColorG` and `ListG a` respectively.
2. Define a function `listToGen :: [a] → ListG a` that transforms lists to their generic representation.
3. What is the generic representation of `[1,2,3]`?

   Is this also the value obtained by `listToGen [1,2,3]`?
4. Is it possible to define a general class `toGen` that transforms ordinary Clean values to their generic representation?

   If this is possible define such a class and instances for integers, characters, lists and tuples, otherwise show the problems with defining such a class.

3 Ordering via a generic representation

Instead of defining instances of the operator $\gg\!<$ for each and every type, we can also transform elements of that type to the uniform representation and compare them.

```
instance >< [a] | >< a where (><) l m = listToGen l >< listToGen m
```

1. Define instance of $\gg\!<$ for the types `UNIT`, `PAIR`, and `EITHER`. Use these to implement the ordering on `Color`, `(a,b)`, and `Tree a`.
2. Are these results equal to the results obtained above?
3. What is the advantage of this generic approach (if any)?
4. What is the disadvantage of the generic approach (if any)?

Deadline

The deadline for this exercise is September 9, 23:59h.