Advanced Programming (I00032)
Overloading and simple generics

Assignment 1

Preliminaries

There is an assignment for each lecture week. The assignments are mandatory and you may only skip one of them. The (strict) deadline is always just before the next lecture. Please, work with a partner, hand in one solution together on Blackboard and put your names and student numbers in the file. You do not get a mark, but assignments are checked and feedback is provided.

Preparation

In this course we will use the functional programming language Clean. Clean is available for Windows, Linux and Max OS X and can be downloaded at `http://wiki.clean.cs.ru.nl/Download_Clean`. Please, use the version with *iTasks*. The IDE is unfortunately available for Windows only, but it is not really necessary for the assignment of this course.

On Linux and Mac you can use the comment line tool `clm`, as for the first assignments you only need a single source file:

```
clm skeleton1 -o skeleton1 (without the extension .icl)
./skeleton1
```

The Clean system comes with a language description in pdf. If you are new to functional programming in Clean you might like `http://www.mbsd.cs.ru.nl/papers/cleanbook/CleanBookI.pdf`. The standard library `StdEnv` is described in `http://www.mbsd.cs.ru.nl/publications/papers/2010/CleanStdEnvAPI.pdf`.

We assume that you are familiar with the basics of functional programming in general, and Clean in particular. It is not impossible to repair a shortage of knowledge during this course, but this requires a considerable effort.

Goals

This exercises review the implementation of instances of a class for various types. This means to implement functions, which have a similar meaning for the different types, and share the same name. After making this exercise you should understand the basics of using the generic approach to replace such classes.

1 Ordering by overloading

On Blackboard 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 >< operator for these types (`Color`, `Tree a`, and `Rose a`) as well as the standard Clean type 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. Being able to define some instance for those types is more important than the actual relation implemented.

Test your implementation by evaluating expressions of the form:

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

Include other expressions in your program to ensure that all instances of >< 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 explain the problems with defining such a class.

3 Ordering via a generic representation

Instead of defining instances of the operator >< 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 $><$ 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 7, 13:30h (just before the next lecture).