Advanced Programming (I00032)
Generic programming by overloading for different kinds and Clean generics

Assignment 3

iTasks test

This is a preparation for the next assignment, to make sure that everyone has iTasks working and can work on the assignment next week. To setup iTasks follow the instruction in <Clean Directory>/lib/iTasks-SDK/Installation/<OS version>/INSTALL.txt. Windows user should use the 32-bit version! To test the installation copy iTasksTest.icl and iTasksTest.prj to the same directory and compile either using the IDE or on the console: cpm iTasksTest.cpm. This should generate the executable test.exe and running it should give:

```
*** test HTTP server ***

Running at http://localhost:8080/
```

Open this URL with a web browser. You should now get a message that you successfully setup iTasks. If you cannot get it running, please send me a mail (s.michels@science.ru.nl).

Preparation

The skeletons use the module StdMaybe, which is not part of StdEnv, and skeleton3b.icl also requires the generic extensions. To include it in the IDE select the environment Everything. On the console on Linux or Mac you can build with:

```
clm -IL ../lib/StdLib -IL ../lib/Generics skeleton3b
```

1 Generic printing and parsing

Use skeleton3a for part 1 and 2 of this assignment. In this assignment we implement the classes show and parse from the previous assignment based on generics with different classes for each kind. Here we use the standard type Maybe,

```
:: Maybe a :== Nothing | Just a
```

instead of a tailor-made data type Result. The new result is:

```
:: Result a :== Maybe (a, [String])
```

In the new implementation you should use kinds as explained in the lecture. This implies that you need a show_ and a parse for the kinds $*$, $* \to *$ and $* \to * \to *$. For kind $*$ we have

```
class show_0 a :: a [String] → [String]
class parse0 a ::   [String] → Result a
```

1.1 With tags

Construct classes for `show_` and `parse` for the other kinds that you need in generic programming and define instances for the generic types (`UNIT`, `PAIR`, `EITHER`, and `CONS`). Implement a version that uses *tags*: a tag is a string that indicates what (generic) constructor is printed. This makes parsing easy: there is always a tag that tells what generic constructor to expect. The expression [1] is shown as:

```
["RIGHT","CONS","Cons","PAIR","Int","1","LEFT","CONS","Nil","UNIT"]
```

Implement `show_` and `parse` at least for the types `Int`, `Bool`, `T`, `Color`, `Tree a`, `[a]`, and `(a,b)` using generic programming.

```
:: T       = C
:: Color   = Red | Yellow | Blue
:: Tree a  = Tip | Bin a (Tree a) (Tree a)
```

Test the correct behavior of your functions by evaluating

```
test :: t → Bool | eq0, show_0, parse0 t
test x
   = case parse0 (show x) of
       Just (y,[]) = eq0 x y
       _           = False
```

for some relevant expressions (e.g. `Start = test [1..3]` should yield `True`). Here we use the new equality classes from the lecture. These classes are listed in the provided prelude.

1.2 Without tags

For the implementation of the parser it is quite convenient if all generic information is available. An ordinary user of the generic system is not interested in the generic information at all. Change the implementation of part 1.1 in such a way that it works correctly without the generic information. The expression [1] should now be shown as `["Cons","1","Nil"]`.

The only tricky instance is the parser for `EITHER`. Just try the parser for the left branch. If it fails you should backtrack and use the parser for the right branch. It is sufficient to hand in only the solution for the parser without tags.

2 Generic map

In the lecture slides you will find a generic `map` for various kinds. These definitions are also listed in the skeleton.

- Use these definitions to map the factorial function over the expressions `[1..10]`, `Bin 2 Tip (Bin 4 Tip Tip)` and `([1..10], Bin 2 Tip (Bin 4 Tip Tip))`.
- Use the generic map to apply `λi.(i,fac i)` to all elements in the list `[1..10]`.

3 Generic printing and parsing using Clean generics

Use `skeleton3b` for part 3 and 4. In this part of the assignment you use the built-in generics of Clean to implement `show_` and `parse`. Do not forget to set the environment of your project to Everything. Just like in part 1.2, this version should not show generic constructs (like `LEFT` and `UNIT`), and hence, parsing expects strings without such tags. Constructor with arguments should be surrounded by parenthesis. Constructors without arguments should not have parenthesis. The necessary information can be found in the `GenericConsDescriptor` from StdGeneric. You can use it in a function definition after the **of** keyword within a generic specializer:

show_{|CONS **of** {gcd_name, gcd_arity}|} ...= ...

Your parser should use the information provided by the generic system in the instance for CONS to check whether the input contains the correct constructor name.

Test this implementation in a similar way as the previous show_ and parse.

4 Extra challenge

If you like some additional challenge you can implement a special case for of show_ and parse for tuples. When you derive everything the expression (1,True) will be shown as ["(","_Tuple2","1","True",")"]. Change the generic show_ and parse such that this is shown as ["(","1",",","True",")"] and that this list of strings is parsed correctly.

Deadline

The deadline for this exercise is September 28, 13:30h.