Declarative Programming and (Co)Induction
Haskell exercises

Davide Ancona and Elena Zucca

PhD Course, DIBRIS, Univ. Genova, June 23-27, 2014

User-defined types
Given the type of binary trees (`deriving Show` makes them printable)

```
data BTree a = Empty | Node  (a,  (BTree a), (BTree a)) deriving Show
```

define the following functions:

`frontier t` the frontier of t (list of the leaves)
`inorder f a t` inorder visit with accumulation parameter `a`, at each node `b` the new value of the accumulation parameter is `f a b`
`inorder_list` (instance of `inorder`) list of the nodes with inorder visit `sum_tree` (instance of `inorder`) sum of the nodes of a tree with numeric labels
`node_num` (istanza of `inorder`) number of nodes

Laziness

- Define a function `iterate::(a -> a) -> a -> [a]` such that `iterate f x` is the infinite list `x, f x, f(f x), f(f(fx)), ...`
  For instance:

```
Main> iterate (*2) 1
[1,2,4,8,16,32,64,128,256,512,1024,...
```

- Define the function `repeat::a -> [a]` such that `repeat x` is the infinite list `x, x, x, ...` (see in the lecture) as an instance of `iterate`.
- Define a function `cycle:: [a] -> [a]` such that `cycle xs` is the infinite list `xs++xs++xs++...`.
- Define `cycle` using `repeat`.
- Define the (predefined) function `takeWhile` mentioned in the lecture, which, applied to a predicate `p` and a list `xs`, returns the longest prefix (possibly empty) of xs of elements that satisfy p:

Interpreter for the $\mathcal{E}$ calculus Implement the $\mathcal{E}$ calculus. Notably:

- Define a type `Exp` modeling language terms (be careful to avoid name conflicts with predefined constructors such as `True` e `False`).
- Define a function `isNum` which checks whether a term is a numeral, that is of shape

  $n \ ::= \ 0 \mid \mathtt{succ}\ n.$

- Define a function `isVal` which checks whether a term is a value.
- Define a function `reduce :: Exp -> Maybe Exp` which models the reduction relation $\rightarrow$. (`data Maybe a = Nothin` is a predefined type for optional values).
- Define a function `reduceStar :: Exp -> Exp` which models the relation $\rightarrow^\star$.
- Implement big-step semantics as a function `big_reduce :: Exp -> Maybe Exp`.