

Introduction to Functional Programming

Zsók Viktória, Ph.D.

Department of Programming Languages and Compilers
Faculty of Informatics
Eötvös Loránd University
Budapest, Hungary
zsv@elte.hu



Overview

- 1 Classes and instances
- 2 Instances
 - \mathbb{Q} rational numbers
 - \mathbb{C} complex numbers
- 3 Type classes



Instances

```
instance + String
```

```
where
```

```
(+) s1 s2 = s1 +++ s2
```

```
Start = "Hello" + " world!" // "Hello world!"
```

```
instance + (a,b) | + a & + b
```

```
where
```

```
(+) (x1,y1) (x2,y2) = (x1+x2,y1+y2)
```

```
Start = (1,2) + (3,4) // (4,6)
```



Instances

```
//in StdTuple.dcl
```

```
instance = (a,b) | Eq a & Eq b
```

```
instance = (a,b,c) | Eq a & Eq b & Eq c
```

```
//in StdTuple.icl
```

```
instance = (a,b) | Eq a & Eq b
```

```
where
```

```
(=) :: !(a,b) !(a,b) → Bool | Eq a & Eq b
```

```
(=) (x1,y1) (x2,y2) = x1==x2 && y1==y2
```

```
instance = (a,b,c) | Eq a & Eq b & Eq c
```

```
where
```

```
(=) :: !(a,b,c) !(a,b,c) → Bool | Eq a & Eq b & Eq c
```

```
(=) (x1,y1,z1) (x2,y2,z2) = x1==x2 && y1==y2 && z1==z2
```

```
Start = (1,2) = (3,4) // False == overloading
```



Instances

```
increment n = n+1
```

```
Start = increment 4
```

```
double :: a → a | + a  
double x = x + x
```

```
Start = double 3
```

```
Start = double 3.3
```



Classes

```
delta :: a a a → a | *,-,fromInt a
delta a b c = b*b - (fromInt 4)*a*c
```

```
Start = delta 1.0 2.0 1.0
```

```
class Delta a | *,-,fromInt a
```

```
delta1 :: a a a → a | Delta a
delta1 a b c = b*b - (fromInt 4)*a*c
```

```
Start = delta1 1.0 2.0 1.0
```



Classes

```
class PlusMinx a
  where
    (+-) infixl 6 :: !a !a → a
    (--) infixl 6 :: !a !a → a
    zerox          :: a
```

```
instance PlusMinx Char
  where
    (+-) :: !Char !Char → Char
    (+-) x y = toChar (toInt(x) + toInt(y))
    (--) x y = toChar (toInt(x) - toInt(y))
    zerox = toChar 0
```

Start = 'a' +- 'e'



Classes

```
Start :: Char  
Start = zerox
```

```
double1 :: a → a | PlusMin a  
double1 x = x + x
```

```
Start = double1 2 // 4
```



Rational nr

```
:: Q = { nom :: Int
        , den :: Int
        }
```

```
instance + Q
```

```
where
```

```
(+) x y = mkQ (x.nom*y.den+y.nom*x.den) (x.den*y.den)
```

```
Start = mkQ 2 4 + mkQ 5 6 // (Q 4 3)
```

```
instance - Q
```

```
where
```

```
(-) x y = mkQ (x.nom*y.den-y.nom*x.den) (x.den*y.den)
```

```
Start = mkQ 2 4 - mkQ 5 6 // (Q -1 3)
```



Rational nr

```
instance fromInt Q
where
    fromInt i = mkQ i 1
```

```
Start :: Q
Start = fromInt 3 // (Q 3 1)
```

```
instance zero Q
where
    zero = fromInt 0
```

```
Start :: Q
Start = zero // (Q 0 1)
```



Rational nr

```
instance one Q
```

```
where
```

```
    one = fromInt 1 //
```

```
Start :: Q
```

```
Start = one // (Q 1 1)
```

```
instance toString Q
```

```
where
```

```
    toString q
```

```
        | xq.den == 1 = toString xq.nom
```

```
        | otherwise = toString xq.nom ++ "/" ++ toString xq.den
```

```
    where xq = simplify q
```

```
Start = toString (mkQ 3 4) // "3/4"
```



Rational nr

```
instance < Q
```

```
where
```

```
  (<) x y = x.nom*y.den < y.nom*x.den
```

```
Start = mkQ 1 2 < mkQ 3 4 // True
```

```
ls = [toString q \\ q ← [zero, mkQ 1 3 .. mkQ 3 2]]
```

```
Start :: [String]
```

```
Start = ls // ["0", "1/3", "2/3", "1", "4/3"]
```



Rational nr

//overloading can not be solved

```
Start = toString zero+zero
```

```
Start :: String
```

```
Start = toString sum // "0"
```

```
where sum :: Q
```

```
      sum = zero + zero
```



Complex nr

```
:: C = { re :: Real  
        , im :: Real  
        }
```

```
mkC n d = { re = n, im = d }
```

```
Start = mkC 1.0 10.0 // (C 1 10)
```

```
instance + C
```

```
where
```

```
(+) x y = mkC (x.re+y.re) (x.im+y.im)
```

```
Start = mkC 2.2 4.1 + mkC 1.5 6.4 // (C 3.7 10.5)
```

```
instance - C
```

```
where
```

```
(-) x y = mkC (x.re-y.re) (x.im-y.im)
```

```
Start = mkC 2.2 4.1 - mkC 1.5 6.4 // (C 0.7 -2.3)
```



Complex nr

```
instance * C
```

```
where
```

```
(*) x y = mkC (x.re*y.re - x.im*y.im) (x.re*y.im + x.im*y.re)
```

```
Start = mkC 2.0 4.0 * mkC 3.0 2.0 // (C -2 16)
```

```
// for simplicity only division by a real nr is defined
```

```
instance / C
```

```
where
```

```
(/) x y
```

```
| y.im == 0.0 = mkC (x.re/y.re) (x.im/y.re)
```

```
= abort "division not defined"
```

```
Start = (mkC 2.0 4.0) / (mkC 2.0 0.0) // (C 1 2)
```



Complex nr

```
instance fromReal C
```

```
where
```

```
  fromReal r = mkC r 0.0
```

```
Start :: C
```

```
Start = fromReal 3.0 // (C 3 0)
```

```
instance toReal C
```

```
where
```

```
  toReal x
```

```
  | x.im == 0.0 = x.re
```

```
  = abort "x has imaginary part"
```

```
Start = toReal (mkC 3.0 0.0) // 3
```



Complex nr

```
instance zero C
where
    zero = fromReal 0.0
```

```
Start :: C
Start = zero // (C 0 0)
```

```
instance one C
where
    one = fromReal 1.0
```

```
Start :: C
Start = one // (C 1 0)
```



Complex nr

```
instance abs C
```

```
where
```

```
  abs x = fromReal (sqrt (x.re*x.re + x.im*x.im))
```

```
Start = abs (mkC 3.0 4.0) // (C 5 0)
```

```
//conjugate of a complex x+yi is x-yi
```

```
instance ¬ C
```

```
where
```

```
  (¬) x = mkC x.re (-x.im)
```

```
Start = ¬ (mkC 2.0 3.0) // (C 2 -3)
```



Complex nr

```
instance toString C
```

```
where
```

```
  toString x
```

```
    | x.im == 0.0 == toString x.re
```

```
    | otherwise = toString x.re +++ "+"
```

```
                +++ toString x.im +++ "i"
```

```
Start = toString (mkC 3.0 4.0) // "3+4i"
```

```
instance == C
```

```
where
```

```
  (==) x y = x.re == y.re && x.im==y.im
```

```
Start = mkC 1.0 2.0 == mkC 1.0 2.0 // True
```



Complex nr

```
// test whether the complex number represents a real nr
```

```
isRealC :: C → Bool
```

```
isRealC x
```

```
| x.im == 0.0 = True
```

```
= False
```

```
Start = isRealC (mkC 2.0 0.0) // True
```

```
re :: C → Real
```

```
re x = x.re
```

```
Start = re (mkC 1.0 2.0) // 1
```

```
im :: C → Real
```

```
im x = x.im
```

```
Start = im (mkC 1.0 2.0) // 2
```



Map

```
module Map
import StdEnv
```

```
// The (Maybe a) type represents a collection of at most one element
:: Maybe a = Just a
           | Nothing
```

```
// Binary trees
:: Tree a = Leaf | Node a (Tree a) (Tree a)
```

```
// Single tree
:: Tree1 a = Node1 a [Tree1 a]
```



Map

// the type constructor class Map such that the all instances bellow can be created.

```
class Map t :: (a → b) (t a) → t b
```

```
instance Map []
```

```
where Map f xs = map1 f xs
```

```
instance Map Maybe
```

```
where Map f mb = mapMaybe f mb
```

```
instance Map Tree
```

```
where Map f tr = mapTree f tr
```



Map

```
instance Map Tree1
where Map f tr = mapTree1 f tr
```

```
instance Map ((,) a)
where
  Map :: (a → b) (c,a) → (c,b)
  Map f (x,y) = (x,f y)
```



Map

// given function, for lists:

```
map1 :: (a → b) [a] → [b]
map1 f [] = []
map1 f [x:xs] = [f x : map1 f xs]
```

// given function, for Maybe:

```
mapMaybe :: (a → b) (Maybe a) → Maybe b
mapMaybe f Nothing = Nothing
mapMaybe f (Just x) = Just (f x)
```



Map

// given function, for Tree:

```
mapTree          :: (a → b) (Tree a) → Tree b
```

```
mapTree f Leaf  = Leaf
```

```
mapTree f (Node x le ri) = Node (f x) (mapTree f le) (mapTree f ri)
```

// given function, for Tree1:

```
mapTree1        :: (a → b) (Tree1 a) → Tree1 b
```

```
mapTree1 f (Node1 elem ls) = Node1 (f elem) (map (mapTree1 f) ls)
```



Map

```
t1 :: Tree Int
t1 = Node 1 Leaf (Node 2 Leaf (Node 3 Leaf (Node 4 Leaf Leaf)))
```

```
a1 :: Tree1 Int
a1 = Node1 1 [Node1 2 [Node1 3 [], Node1 4 [], Node1 5
    [Node1 6 []]]]
```

```
Start = Map inc [1..10]
```

```
Start :: Maybe Int
```

```
Start = Map inc (Just 4)
```

```
Start = Map inc Nothing
```



Map

```
Start = t1
```

```
Start = Map inc t1
```

```
Start = a1
```

```
Start = Map inc a1
```

```
Start = Map inc (True, 4)
```

```
Start = Map inc (1.5, 2)
```



Conclusions

The goal was:

- to get familiarized with all the key elements of the functional languages
- to practice by examples in order to acquire the functional programming paradigm

For FP mark of the course: please choose 5 tasks from the assignment list (1 numerical, 1 list processing, 1 records - arrays, 1 trees, 1 instances-classes) and send the solution .icl file by email to zsv@elte.hu before 1st of December.

THANK YOU FOR TAKING PART IN THIS COURSE!

