

Informatics 1
Introduction to Computation
Lecture 8

Algebraic Data Types

Philip Wadler
University of Edinburgh

Functional Programming Class Test

11.00–11.50 Tuesday 22 October 2018

Gordon Aikman Lecture Theatre

Plus some other rooms — you will be notified

Closed book written exam, no computers

Will cover everything in FP up to — but not including — higher-order functions

Past exams are available via the course Learn page

Formative vs. Summative

| | |
|-----|---|
| 0% | Week 1: FP tutorial |
| 0% | Week 2: CL and FP tutorials |
| 0% | Week 3: CL and FP tutorials |
| 0% | Week 4: CL and FP tutorials |
| 0% | Week 5: CL and FP tutorials |
| 10% | Week 6: Functional Programming Class Test |
| 0% | Week 6: CL and FP tutorials |
| 0% | Week 7: CL and FP tutorials |
| 0% | Week 8: CL and FP tutorials |
| 0% | Week 9: CL, FP, IP tutorials |
| 0% | Week 10: CL, FP, IP tutorials |
| 0% | Week 11: Mock Functional Programming Exam |
| 90% | December: Final Exam |

Formative vs. Summative

| | |
|--------------------|---|
| 0% | Week 1: FP tutorial |
| 0% | Week 2: CL and FP tutorials |
| 0% | Week 3: CL and FP tutorials |
| 0% | Week 4: CL and FP tutorials |
| 0% | Week 5: CL and FP tutorials |
| 10% 5% | Week 6: Functional Programming Class Test |
| 0% | Week 6: CL and FP tutorials |
| 0% | Week 7: CL and FP tutorials |
| 0% | Week 8: CL and FP tutorials |
| 0% | Week 9: CL, FP, IP tutorials |
| 0% | Week 10: CL, FP, IP tutorials |
| 0% | Week 11: Mock Functional Programming Exam |
| 90% 95% | December: Final Exam |

Part I

Algebraic types

Everything is an algebraic type

```
data Bool = False | True
data Season = Winter | Spring | Summer | Fall
data Shape = Circle Float | Rectangle Float Float
data List a = Nil | Cons a (List a)
data Nat = Zero | Succ Nat
data Exp = Lit Int | Add Exp Exp | Mul Exp Exp
data Tree a = Empty | Leaf a | Branch (Tree a) (Tree a)
data Maybe a = Nothing | Just a
data Pair a b = Pair a b
data Either a b = Left a | Right b
```

Part II

Boolean

Boolean

```
import Prelude
  hiding (Bool(True, False), (&&), (||), not)
```

```
data Bool = False | True
  deriving (Eq, Show)
```

```
not :: Bool -> Bool
not False  =  True
not True   =  False
```

```
(&&) :: Bool -> Bool -> Bool
False && q    =  False
True  && q    =  q
```

```
(||) :: Bool -> Bool -> Bool
False || q   =  q
True  || q   =  True
```


Boolean — eq and show

```
eqBool :: Bool -> Bool -> Bool
eqBool False False  =  True
eqBool False True   =  False
eqBool True  False  =  False
eqBool True  True   =  True
```

```
showBool :: Bool -> String
showBool False  =  "False"
showBool True   =  "True"
```

Part III

Seasons

Seasons

```
data Season = Winter | Spring | Summer | Fall  
  deriving (Eq, Show)
```

```
next :: Season -> Season  
next Winter    = Spring  
next Spring    = Summer  
next Summer    = Fall  
next Fall      = Winter
```

Seasons—eq and show

```
eqSeason :: Season -> Season -> Bool
eqSeason Winter Winter = True
eqSeason Spring Spring = True
eqSeason Summer Summer = True
eqSeason Fall Fall = True
eqSeason x y = False
```

```
showSeason :: Season -> String
showSeason Winter = "Winter"
showSeason Spring = "Spring"
showSeason Summer = "Summer"
showSeason Fall = "Fall"
```

Seasons and integers

```
toInt :: Season -> Int
toInt Winter  = 0
toInt Spring  = 1
toInt Summer  = 2
toInt Fall    = 3
```

```
fromInt :: Int -> Season
fromInt 0  = Winter
fromInt 1  = Spring
fromInt 2  = Summer
fromInt 3  = Fall
```

```
next :: Season -> Season
next x = fromInt ((toInt x + 1) `mod` 4)
```

```
eqSeason :: Season -> Season -> Bool
eqSeason x y = (toInt x == toInt y)
```

Part IV

Shape

Shape

```
type Radius = Float
```

```
type Width = Float
```

```
type Height = Float
```

```
data Shape = Circle Radius  
          | Rect Width Height
```

```
    deriving (Eq, Show)
```

```
area :: Shape -> Float
```

```
area (Circle r) = pi * r^2
```

```
area (Rect w h) = w * h
```

Shape—eq and show

```
eqShape :: Shape -> Shape -> Bool
eqShape (Circle r) (Circle r')    = (r == r')
eqShape (Rect w h) (Rect w' h')   = (w == w') && (h == h')
eqShape x           y              = False

showShape :: Shape -> String
showShape (Circle r)  = "Circle " ++ showF r
showShape (Rect w h)  = "Rect " ++ showF w ++ " " ++ showF h

showF :: Float -> String
showF x | x >= 0       = show x
        | otherwise    = "(" ++ show x ++ ")"
```


Shape—tests and selectors

```
isCircle :: Shape -> Bool
isCircle (Circle r)    = True
isCircle (Rect w h)    = False
```

```
isRect :: Shape -> Bool
isRect (Circle r)      = False
isRect (Rect w h)      = True
```

```
radius :: Shape -> Float
radius (Circle r)      = r
```

```
width :: Shape -> Float
width (Rect w h)       = w
```

```
height :: Shape -> Float
height (Rect w h)      = h
```

Shape—pattern matching

```
area :: Shape -> Float
area (Circle r)  = pi * r^2
area (Rect w h)  = w * h
```

```
area :: Shape -> Float
area s =
  if isCircle s then
    let
      r = radius s
    in
      pi * r^2
  else if isRect s then
    let
      w = width s
      h = height s
    in
      w * h
  else error "impossible"
```

Part V

Lists

Lists

With declarations

```
data List a = Nil
           | Cons a (List a)
deriving (Eq, Show)
```

```
append :: List a -> List a -> List a
append Nil ys          = ys
append (Cons x xs) ys  = Cons x (append xs ys)
```

With built-in notation

```
(++) :: [a] -> [a] -> [a]
[] ++ ys      = ys
(x:xs) ++ ys  = x : (xs ++ ys)
```

Part VI

Natural numbers

Naturals

With names

```
data Nat = Zero
         | Succ Nat
deriving (Eq, Show)
```

```
power :: Float -> Nat -> Float
power x Zero      = 1.0
power x (Succ n)  = x * power x n
```

With built-in notation

```
(^^) :: Float -> Int -> Float
x ^^ 0 = 1.0
x ^^ n = x * (x ^^ (n-1))
```

Naturals

```
add :: Nat -> Nat -> Nat
```

```
add Zero n      = n
```

```
add (Succ m) n  = Succ (add m n)
```

```
mul :: Nat -> Nat -> Nat
```

```
mul Zero n      = Zero
```

```
mul (Succ m) n  = add (mul m n) n
```