Haskell

Functional programming

Michele Tomaiuolo
Ingegneria dell'Informazione, UniPR

Algebraic data types

- We've run into a lot of data types: `Bool, Int, Char`...
- How do we make our own?
- One way is to use the `data` keyword to define a *type*
  - *Type name* and *value constructors*: capital cased
  - *Algebra* of *sums* (alternations) and *products* (combinations)

```haskell
data TrafficLight = Red | Yellow | Green
```

HASKELL

```haskell
data Shape = Circle Float Float Float
           | Rectangle Float Float Float Float
```

HASKELL

Value constructors

- Value constructors are f.s
  - They return a value of a data type
  - Fields are actually params

HASKELL

```haskell
Prelude> :t Circle
Circle :: Float -> Float -> Float -> Shape
Prelude> :t Rectangle
Rectangle :: Float -> Float -> Float -> Float -> Shape
```

Functions on datatypes

- F. that takes a shape and returns its surface
  - `Circle` is not a type, Shape is
  - We can pattern match against constructors

HASKELL

```
surface :: Shape -> Float
surface (Circle _ _ r) = pi * r ^ 2
surface (Rectangle x1 y1 x2 y2)
        = (abs $ x2 - x1) * (abs $ y2 - y1)
```

HASKELL

```
Prelude> surface $ Circle 10 20 10
314.15927
Prelude> surface $ Rectangle 0 0 100 100
10000.0
```

Show typeclass

- Error if we try to just print out `Circle 10 20 5`
  - Haskell doesn't know how to display our data type as a string (yet)
  - Make our Shape type part of the Show typeclass

HASKELL
```
data Shape = Circle Float Float Float
           | Rectangle Float Float Float Float
           deriving (Show)
```

HASKELL
```
Prelude> Circle 10 20 5
Circle 10.0 20.0 5.0
Prelude> Rectangle 50 230 60 90
Rectangle 50.0 230.0 60.0 90.0
```

Point datatype

HASKELL

```
data Point = Point Float Float deriving (Show)
data Shape = Circle Point Float |
             Rectangle Point Point deriving (Show)
```

- Same name for the data type and the value constructor
    - Idiomatic if there's only one value constructor

HASKELL

```
surface :: Shape -> Float
surface (Circle _ r) = pi * r ^ 2
surface (Rectangle (Point x1 y1) (Point x2 y2))
        = (abs $ x2 - x1) * (abs $ y2 - y1)
```

HASKELL

```
Prelude> surface (Rectangle (Point 0 0) (Point 100 100))
10000.0
Prelude> surface (Circle (Point 0 0) 24)
1809.5574
```

Nudging a shape

- F. that takes shape, dx, dy…
- Returns a *new shape*, located somewhere

HASKELL

```
nudge :: Shape -> Float -> Float -> Shape
nudge (Circle (Point x y) r) a b = Circle (Point (x+a) (y+b)) r
nudge (Rectangle (Point x1 y1) (Point x2 y2)) a b
      = Rectangle (Point (x1+a) (y1+b)) (Point (x2+a) (y2+b))
```

HASKELL

```
Prelude> nudge (Circle (Point 34 34) 10) 5 10
Circle (Point 39.0 44.0) 10.0
```

Shapes at the origin

HASKELL

```
baseCircle :: Float -> Shape
baseCircle r = Circle (Point 0 0) r

baseRect :: Float -> Float -> Shape
baseRect width height = Rectangle (Point 0 0) (Point width height)
```

HASKELL

```
Prelude> nudge (baseRect 40 100) 60 23
Rectangle (Point 60.0 23.0) (Point 100.0 123.0)
```

Record syntax

- Create a data type that describes a person
  - First name, last name, age, height, phone number, and favorite ice-cream flavor

HASKELL

```
data Person = Person String String Int Float
                     String String deriving (Show)
```

HASKELL

```
Prelude> let guy = Person "Buddy" "Finklestein" 43 184.2
                          "526-2928" "Chocolate"
Prelude> guy
Person "Buddy" "Finklestein" 43 184.2 "526-2928" "Chocolate"
```

Accessing fields

HASKELL

```haskell
firstName   (Person firstname _ _ _ _ _) = firstname
lastName    (Person _ lastname  _ _ _ _) = lastname
age         (Person _ _ age       _ _ _) = age
height      (Person _ _ _ height    _ _) = height
phoneNumber (Person _ _ _ _ number    _) = number
flavor      (Person _ _ _ _ _ flavor   ) = flavor
```

HASKELL

```haskell
Prelude> :t flavor
flavor :: Person -> String
```

Record syntax

- Haskell automatically creates accessor f.s
- Deriving Show, output is more complete

HASKELL

```
data Person = Person { firstName :: String
                     , lastName :: String
                     , age :: Int
                     , height :: Float
                     , phoneNumber :: String
                     , flavor :: String
                     } deriving (Show)
```

HASKELL

```
Prelude> :t flavor
flavor :: Person -> String
Prelude> :t firstName
firstName :: Person -> String
```

Type constructors

- Type constructors take types as params to produce new types
  - Similar to templates in C++
  - Ex.: Maybe is defined with a *type parameter* (a)
  - Ex.: list type takes a param to produce a concrete type

HASKELL

```haskell
data Maybe a = Nothing | Just a
```

HASKELL

```haskell
Prelude> import Data.Maybe
Prelude Data.Maybe> isJust Nothing
False
Prelude Data.Maybe> fromJust (Just 5)
5
```

Maybe for reading and finding

HASKELL

```
Prelude> import Text.Read
Prelude Text.Read> readMaybe "5" :: Maybe Int
Just 5
Prelude Text.Read> readMaybe "??" :: Maybe Int
Nothing
```

HASKELL

```
Prelude> import Data.List
Prelude Data.List> elemIndex 0 [1,4,0,3,2]
Just 2
Prelude Data.List> elemIndex 0 [1,4,3,2]
Nothing
```

Maybe an int

- Without the *type parameter* (a)...
- `Maybe'` defined for a precise content type, e.g. `Int`
- For containing a `String`, different definition needed

HASKELL

```haskell
data Maybe' = Nothing' | Just' Int
```

HASKELL

```haskell
Prelude> :t Just' 84
Just' 84 :: Maybe'
Prelude> :t Nothing'
Nothing' :: Maybe'
Prelude> Just' "Hello"
...
    Couldn't match expected type ‘Int’ with actual type ‘[Char]’
```

Derived instances

- *Typeclass*: interface that defines some behavior
  - *Type* as instance, if it supports that behavior
  - Ex.: == and /= act as interface for Eq

HASKELL

```
data Person = Person { firstName :: String
                     , lastName :: String
                     , age :: Int
                     } deriving (Eq)
```

- Haskell can *automatically* make our type an instance of:
  - `Eq, Ord, Enum, Bounded, Show, Read`
- Haskell will see if
  - The value constructors match (only one here)
  - Each pair of fields match, using == (fields are Eq)

Show and Read types

HASKELL

```
data Person = Person { firstName :: String
                     , lastName :: String
                     , age :: Int
                     } deriving (Eq, Show, Read)
```

HASKELL

```
Prelude> let mikeD = Person {firstName = "Michael",
                     lastName = "Diamond", age = 43}
Prelude> "mikeD is: " ++ show mikeD
"mikeD is: Person {firstName = \"Michael\",
                   lastName = \"Diamond\", age = 43}"
Prelude> read "Person {firstName =\"Michael\",
               lastName =\"Diamond\", age = 43}" :: Person
Person {firstName = "Michael", lastName = "Diamond", age = 43}
Prelude> read "Person {firstName =\"Michael\",
               lastName =\"Diamond\", age = 43}" == mikeD
True
```

Enum and Bound types

- Use algebraic data types to make enumerations

HASKELL

```haskell
data Day = Monday | Tuesday | Wednesday | Thursday
           | Friday | Saturday | Sunday
           deriving (Eq, Ord, Show, Read, Bounded, Enum)
```

HASKELL

```haskell
Prelude> succ Friday
Saturday
Prelude> Friday >= Wednesday
True
```

Type synonyms

- Giving some types different names

HASKELL

```haskell
type String = [Char]   -- equivalent and interchangeable
```

- To convey more information about data

HASKELL

```haskell
type Name = String
type PhoneNumber = String
type PhoneBook = [(Name,PhoneNumber)]

-- inPhoneBook :: String -> String -> [(String,String)] -> Bool
inPhoneBook :: Name -> PhoneNumber -> PhoneBook -> Bool
inPhoneBook name pnumber pbook = (name,pnumber) `elem` pbook
```

Ex.: Search in phone book

- Implement a f. for `PhoneBook`
- `getPhoneNumber :: Name -> PhoneBook -> PhoneNumber`
- Different patterns
  - `x:xs`
  - `((k,v):xs)`
- Result if name not found:
  - `""`
  - `error "No phone number for " ++ name`
  - Change signature to return `Maybe PhoneNumber`

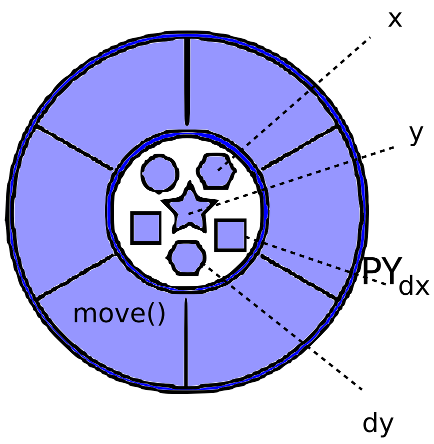
Ex.: Bouncing ball

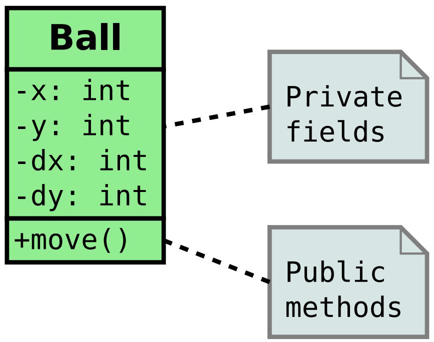
- Mimic the following Python datatype, in Haskell *functional* style
- Implement a `move` f., for advancing a step and bouncing at borders

```
ARENA_W, ARENA_H = 320, 240
BALL_W, BALL_H = 20, 20

class Ball:
    def __init__(self, x: int, y: int):
        self._x = x
        self._y = y
        self._dx = 5
        self._dy = 5
    # ...
```

http://www.ce.unipr.it/brython/?p2_oop_ball.py





Class diagram UML

Randomness

- The `System.Random` module has all needed f.s, including `random`

HASKELL

```
random :: (RandomGen g, Random a) => g -> (a, g)
```

- It takes a random generator
- It returns a random value and a new random generator
  - `RandomGen`: types acting as sources of randomness
  - `Random`: types representing random values
- Why does it also return a new generator?
  - *Idempotence*: calling a f. with same params twice, produces same result

```
sudo apt install libghc-random-dev
```

Rnd generators

- `StdGen`: instance of `RandomGen`
- `mkStdGen` f., to manually make a random generator

HASKELL

```
mkStdGen :: Int -> StdGen
```

HASKELL

```
Prelude> random (mkStdGen 100) :: (Int, StdGen)
(-1352021624,651872571 1655838864)
Prelude> random (mkStdGen 100) :: (Int, StdGen)
(-1352021624,651872571 1655838864)
```

- Same parameters → Same result

Tossing a coin

- Represent a coin with a simple `Bool`: `True` is tails, `False` is heads
- Call `random` with a generator, get a coin and a new generator

HASKELL
```haskell
threeCoins :: StdGen -> (Bool, Bool, Bool)
threeCoins gen =
    let (firstCoin, newGen) = random gen
        (secondCoin, newGen') = random newGen
        (thirdCoin, newGen'') = random newGen'
    in  (firstCoin, secondCoin, thirdCoin)
```

HASKELL
```haskell
Prelude> threeCoins (mkStdGen 21)
(True,True,True)
Prelude> threeCoins (mkStdGen 943)
(True,False,True)
```

Multiple random values

- `randoms` f. takes a generator and returns an infinite sequence of values
- Doesn't give the new random generator back

HASKELL
```
randoms' :: (RandomGen g, Random a) => g -> [a]
randoms' gen = let (value, newGen) = random gen
               in value:randoms' newGen
```

HASKELL
```
Prelude> take 5 $ randoms (mkStdGen 11) :: [Int]
[-1807975507,545074951,-1015194702,-1622477312,-502893664]
Prelude> take 5 $ randoms (mkStdGen 11) :: [Bool]
[True,True,True,True,False]
```

Random in a range

- `randomR`: single random value within a defined range

HASKELL

```
Prelude> randomR (1,6) (mkStdGen 123456)
(4,645041272 40692)
Prelude> randomR (1,6) (mkStdGen 654321)
(6,412237752 40692)
```

- `randomRs`: stream of random values within a defined range

HASKELL

```
Prelude> take 10 $ randomRs ('a','z') (mkStdGen 3) :: [Char]
"ndkxbvmomg"
```

The impure

Input and output

- Imperative languages: series of steps to execute
- Functional programming: defining what stuff is
- Haskell is a purely functional language
    - A f. can't change some state, or produce side-effects
    - Result based only on the params
    - Called twice with same params: same result
- I/O ops require changing some state
    - Haskell separates the pure part of the program...
    - from the impure, which does all the dirty work...
    - like talking to the keyboard and the screen

Hello, world!

- Up until now, we've always loaded our functions into GHCI to test them
- Let's write our first Haskell program (`helloworld.hs`)

```
main = print "hello, world"
```

- And now let's build and run it

```
$ ghc --make helloworld
[1 of 1] Compiling Main             ( helloworld.hs, helloworld.o )
Linking helloworld ...
$ ./helloworld
"hello, world"
```

ELL

SHELL

I/O actions

HASKELL

```
Prelude> :t print
print :: Show a => a -> IO ()
Prelude> :t print "hello, world"
print "hello, world" :: IO ()
```

- *I/O action*: action with side-effects
  - E.g., reading input or writing to screen
  - And may also contain some result value
- `print` takes a value and returns an *I/O action*
  - Result type `()` -- empty tuple, aka *unit*
- I/O action performed when named as **`main`**
  - And the program is run

Sequence of actions

- One I/O action seems limiting...
- Use **do** syntax to glue together several I/O actions into one

HASKELL

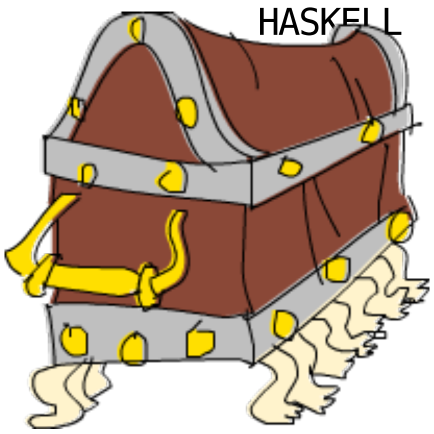
```
main = do
    print "Hello, what's your name?"
    name <- getLine
    print ("Hey " ++ name ++ ", you rock!")
```

- This reads like an imperative program
  - We laid out a series of steps into a single do
  - Each step is an I/O action
  - The whole do has type `IO ()`, same as last I/O action inside

Getting data

```
Prelude> :t getLine
getLine :: IO String
```

- What does "`name <- getLine`" mean?
    - Perform the I/O action `getLine` (get a line from *stdin*)
    - Then bind its result value to `name`
- I/O action: ~ *box* to send into the real, impure world
    - Do something there
    - Maybe bring back some data
- Arrow (`<-`) to open box and get data
    - In particular, `getLine` contains a `String`
    - This can be done only inside another I/O action



I/O results

- Take a look at this piece of code. Is it valid?

HASKELL

```haskell
nameTag = "Hello, my name is " ++ getLine
```

- `++` requires both its params to be lists over the same type
  - The left parameter has a type of `String` (or `[Char]`)
  - `getLine` has a type of `IO String`
  - You can't concatenate a string and an I/O action

Binding

HASKELL

```haskell
name = getLine
```

- This code doesn't read text from the input and bind it to a name
  - It gives the getLine I/O action a different name
- To get the value out of an I/O action
  - Bind it to a name with <-, inside another I/O action
  - Deal with impure data, in impure env
  - *Keep the I/O parts of your code as small as possible!*

Lines with reversed words

- Continuously read a line and print it out with the words reversed, until reading a blank line

HASKELL

```
main = do
    line <- getLine
    if null line
        then return ()
        else do
            print $ reverseWords line
            main
reverseWords :: String -> String
reverseWords = unwords . map reverse . words
```

- Protip: `runhaskell` runs a program on the fly
  - `runhaskell helloworld.hs`
- The `words`, `unwords` f.s are in the stdlib

The return action

- `return` in Haskell is *different* from other languages
  - It doesn't stop the execution of the I/O do block
  - It just makes an I/O action out of a pure value
- Mostly use return to create an I/O action that either:
  - Doesn't do anything, or
  - Always contains the desired result (we put it at the end)
- We can use `return` in combination with `<-`
  - In fact, they're sort of *opposite*

HASKELL

```haskell
main = do
    a <- return "hell"      -- hey, just use let!
    b <- return "yeah!"     -- hey, just use let!
    print $ a ++ " " ++ b
```

Split or join text

- *Newline* as separator
  - `lines :: String -> [String]`
  - `unlines :: [String] -> String`
- *Spaces* as separator
  - `words :: String -> [String]`
  - `unwords :: [String] -> String`
- Split or join with a *given separator*
  - `splitOn :: String -> String -> [String]`
  - `intercalate :: String -> [String] -> String`
  - In modules `Data.List.Split` and `Data.List`

sudo apt install libghc-split-dev

putChar and putStr

- `putChar` takes a char and returns an I/O action to print it
- `putStr` is much like `putStrLn`, without a new line
  - Defined recursively with the help of `putChar`

HASKELL

```
putStr :: String -> IO ()
putStr [] = return ()
putStr (x:xs) = do
    putChar x
    putStr xs
```

- `print` prints an instance of `Show`
- It's basically `putStrLn . show`
- `getChar` reads a `Char` from the input (with buffering)

The when action

- Like a control flow statement, but actually a normal f.
- It takes a boolean value and an I/O action
  - If value is `True`, it returns the same I/O action
  - If it's `False`, it returns `return ()` -- void action
- Encapsulats `if ... else return ()` pattern

HASKELL

```
import Control.Monad

main = do
    c <- getChar
    when (c /= ' ') $ do
        putChar c
        main
```

The sequence action

- It takes a list of I/O actions
- It returns an I/O action to perform them in sequence
- Action result: list of the results

HASKELL
```
sequence :: [IO a] -> IO [a]
```

HASKELL
```
main = do
    rs <- sequence [getLine, getLine, getLine]
    print rs
```

HASKELL
```
main = do
    a <- getLine
    b <- getLine
    c <- getLine
    print [a,b,c]
```

The sequence action

- Useful when mapping f.s like `print` or `putStrLn` over lists
- `map print [1,2,3,4]` creates a list of I/O actions
- `sequence` transforms that list into an I/O action

HASKELL

```
Prelude> sequence (map print [1,2,3,4,5])
1
2
3
4
5
[(),(),(),(),()]
```

- What's with the `[(),(),(),(),()]` at the end?
- When we evaluate an I/O action *in GHCI*, it's performed, and...
- Then its result is printed out, unless it's `()`

The mapM action

- Mapping a f. that returns an I/O action over a list and then sequencing: very common
- `mapM` takes a f. and a list, maps the function over the list, then sequences it
- `mapM_` does the same, only it throws away the result

HASKELL

```
Prelude> mapM print [1,2,3]
1
2
3
[(),(),()]
Prelude> mapM_ print [1,2,3]
1
2
3
```

The forever action

- `forever` takes an I/O action `act` and...
- Returns an I/O action that just repeats `act` forever

HASKELL

```
import Control.Monad
import Data.Char

main = forever $ do
    print "Give me some input:"
    l <- getLine
    print $ map toUpper l
```

The forM action

- `forM` is like `mapM`, with switched params
- Useful in combination with lambdas and do notation

HASKELL

```
import Control.Monad

main = do
    colors <- forM [1,2,3,4] (\a -> do
        print $ "Which color do you associate with the number "
                ++ show a ++ "?"
        color <- getLine
        return color)
    print "The colors that you associate with 1, 2, 3 and 4 are: "
    mapM print colors
```

- Simply `getLine`, which already contains same data
    - `color <- getLine; return color;` is just unpacking and repackaging the result

Interact

- `getContents`: whole stdin as a `String` (*lazy*)

HASKELL

```
main = do
    contents <- getContents
    putStr (shortLinesOnly contents)

shortLinesOnly :: String -> String
shortLinesOnly input =
    let allLines = lines input
        shortLines = filter (\line -> length line < 10) allLines
    in  unlines shortLines
```

- `interact`: applies a `String -> String` f. between stdin and stdout (*lazy*)

HASKELL

```
main = interact $ unlines . filter ((<10) . length) . lines
```

Basic operations on files

- Basic operations on file:
  - Open/close: `openFile`, `hClose`, `withFile`
  - Mode: `ReadMode | WriteMode | AppendMode | ReadWriteMode`
  - Read: `hGetContents`, `hGetLine`, `hGetChar`
  - Write: `hPrint`, `hPutStr`, `hPutStrLn`

HASKELL

```
import System.IO

main = do
    withFile "something.txt" ReadMode (\handle -> do
        contents <- hGetContents handle
        putStr contents)
```

Read and write on files

- simpler f.s: `readFile`, `writeFile`, `appendFile`

HASKELL

```
import System.IO
import Data.Char

main = do
    contents <- readFile "girlfriend.txt"
    writeFile "girlfriendcaps.txt" (map toUpper contents)
```

Getting a rnd generator

- `getStdGen`, get the global rnd generator (`:: IO StdGen`)
  - Performed twice: get same generator
- `newStdGen`, get a new generator, update the global one

HASKELL

```
import System.Random

main = do
    gen <- getStdGen
    putStr $ take 20 (randomRs ('a','z') gen)

$ runhaskell random_string.hs
pybphhzzhuepknbykxhe
$ runhaskell random_string.hs
eiqgcxykivpudlsvvjpg
```

Guess the number

HASKELL

```
main = do
    gen <- getStdGen
    askForNumber gen

askForNumber :: StdGen -> IO ()
askForNumber gen = do
    let (secret, newGen) = randomR (1,10) gen :: (Int, StdGen)
    print "Which number (1-10) am I thinking of?"
    guess <- getLine
    when (not $ null guess) $ do
        if secret == (read guess)
            then print "You are correct!"
            else print $ "Sorry, it was " ++ show secret
        askForNumber newGen
```

Guess, purer

HASKELL

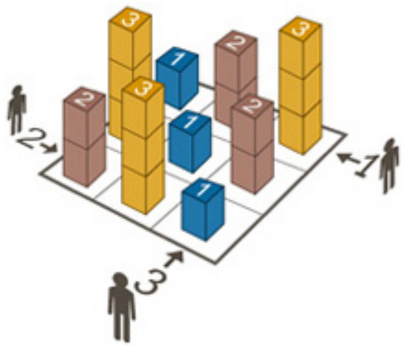
```haskell
process :: StdGen -> [String] -> [String]
process gen guesses =
    "Which number (1-10) am I thinking of?":
        check newGen (show secret) guesses
    where
        (secret, newGen) = randomR (1,10) gen :: (Int, StdGen)

check :: StdGen -> String -> [String] -> [String]
check _ _ ("":_) = []
check gen secret (guess:guesses)
    | guess == secret = "You are correct!":process gen guesses
    | otherwise = ("Sorry, it was " ++ secret):process gen guesses

main = do
    gen <- getStdGen
    interact $ unlines . (process gen) . lines
```

Ex.: Skyscrapers

- Open the following files in Haskell and read the content as a matrix
  - http://sowide.ce.unipr.it/sites/default/files/files/games.zip
  - The numbers on the borders represent constraints to satisfy
- Check if data complies with the following rules
  - https://www.brainbashers.com/skyscrapershelp.asp
  - Check also unicity and range of values
- Possibly, use `reverse` and `transpose`
  - From module `Data.List`



More on types

Either

- Encapsulate a value of one type or another
- Two value constructors
  - If `Left` is used, then its contents are of type a
  - If `Right` is used, then its contents are of type `b`

HASKELL

```haskell
data Either a b = Left a | Right b
                  deriving (Eq, Ord, Read, Show)
```

HASKELL

```haskell
Prelude> Right 20
Right 20
Prelude> :t Right 'a'
Right 'a' :: Either a Char
Prelude> :t Left True
Left True :: Either Bool b
```

Use of Either

- `Maybe` can represent a result that could have either failed or not
  - `Nothing` doesn't convey details about failure
- `Either a b`, when interested in how some function failed or why
  - Errors use the `Left` value constructor
  - While results use `Right`
  - a is a type that tells something about failure
  - `b` type of a successful computation

Recursive data structures

- One value of some type contains values of that type...
  - We can make types whose constructors have fields...
  - that are of the same type
- List `[4,5]` same as `4:(5:[])`
  - First `:` has an element on its left side...
  - and a list (`5:[]`) on its right side
- A list can be:
  - An empty list, or
  - An element joined together with a `:` with another list

Generic list

HASKELL

```
data List a = Empty
              | Cons a (List a)
              deriving (Show, Read, Eq, Ord)
```

HASKELL

```
data List a = Empty
              | Cons { listHead :: a, listTail :: List a}
              deriving (Show, Read, Eq, Ord)
```

- `Cons` constructor represents `:`
  - `:` is a constructor for lists (params: value, list)

HASKELL

```
Prelude> Empty
Empty
Prelude> 4 `Cons` (5 `Cons` Empty)
Cons 4 (Cons 5 Empty)
```

List of ints

- Without the *type parameter* (a)…
- A `List'` should be defined for a precise content type, e.g. `Int`
- For containing a `String`, for example, a different definition of `List'` would be needed

HASKELL

```
data List' = Empty'
           | Cons' Int (List')
           deriving (Show, Read, Eq, Ord)
```

HASKELL

```
Prelude> Empty'
Empty'
Prelude> 4 `Cons'` (5 `Cons'` Empty')
Cons' 4 (Cons' 5 Empty')
```

Binary search tree

- A tree is either an empty tree, or...
- it's an element that contains some value and two trees
  - Elements at the left sub-tree are smaller than the value
  - Elements in the right sub-tree are bigger

HASKELL

```
data Tree a = EmptyTree
            | Node a (Tree a) (Tree a)
            deriving (Show, Read, Eq)
```

- Instead of manually building a tree...
- Make a f. that takes a tree and an element to insert

Inserting an element

- In *C* etc., we modify the pointers and values inside the tree
- In *Haskell*, the insertion function returns a **new tree**
    - `a -> Tree a - > Tree a`
- It seems inefficient, but most of the structure is shared

HASKELL

```
singleton :: a -> Tree a    -- just a shortcut f.
singleton x = Node x EmptyTree EmptyTree

treeInsert :: (Ord a) => a -> Tree a -> Tree a
treeInsert x EmptyTree = singleton x
treeInsert x (Node a left right)
    | x == a = Node x left right
    | x < a  = Node a (treeInsert x left) right
    | x > a  = Node a left (treeInsert x right)
```

Folding into a tree

- Folding: traversing a list and returning some value
- Use a fold to build up a tree from a list

HASKELL

```
Prelude> let nums = [8,6,4,1,7,3,5]
Prelude> let numsTree = foldr treeInsert EmptyTree nums
Prelude> numsTree
Node 5 (Node 3 ...
```

Checking for membership

HASKELL

```
treeElem :: (Ord a) => a -> Tree a -> Bool
treeElem x EmptyTree = False
treeElem x (Node a left right)
    | x == a = True
    | x < a  = treeElem x left
    | x > a  = treeElem x right
```

HASKELL

```
Prelude> 8 `treeElem` numsTree
True
Prelude> 100 `treeElem` numsTree
False
```

Making typeclasses

Defining a typeclass

HASKELL

```
class Eq a where              -- stdlib
    (==) :: a -> a -> Bool
    (/=) :: a -> a -> Bool
    x == y = not (x /= y)
    x /= y = not (x == y)
```

- Keyword **`class`** for defining a new typeclass
    - a is the *type variable*
- Then, specify some f.s (*type declarations*)
    - It's not mandatory to implement them
- Here, f.s are mutually recursive
    - Two Eq are equal if they are not different
    - They are different if they are not equal

Creating instances

HASKELL
```haskell
data TrafficLight = Red | Yellow | Green
```

- Let's write up an *instance* by hand

HASKELL
```haskell
instance Eq TrafficLight where
    Red == Red = True
    Green == Green = True
    Yellow == Yellow = True
    _ == _ = False
```

- In class declaration, `==` defined in terms of `/=` and vice versa
    - In instance declaration, only overwrite one of them
    - Called *minimal complete definition* for the typeclass

Show instance

- Satisfy the minimal complete definition for Show...
  - Implement its *show* function
  - It takes a value and turns it into a string

HASKELL

```
instance Show TrafficLight where
    show Red = "Red light"
    show Yellow = "Yellow light"
    show Green = "Green light"
```

HASKELL

```
Prelude> Red == Yellow
False
Prelude> Red `elem` [Red, Yellow, Green]
True
Prelude> [Red, Yellow, Green]
[Red light,Yellow light,Green light]
```

Subclasses

- You can also make typeclasses that are *subclasses* of other typeclasses
- Ex.: class declaration for `Num`

HASKELL

```
class (Eq a) => Num a where
   ...
```

- We have to make a type an instance of `Eq`...
- Before we can make it an instance of `Num`

Info about types

- The a from `class Eq a` will be replaced with a real type, when you make an instance
- So try mentally putting your type into the function type declarations as well
- To see what the instances of a typeclass are, just do `:info YourTypeClass` in GHCI
  - `:info` works for types, type constructors, functions

Functor typeclass

- The `Functor` typeclass is basically for things that can be mapped over
    - (Yes, *list* type is part of `Functor`)

HASKELL
```
class Functor f where
    fmap :: (a -> b) -> f a -> f b
```

- It defines one function, `fmap`, no default implementation
- Type: `fmap` takes a f. from one type a to another `b` and a *functor* applied to a and returns the functor applied to `b`
- `f` not a concrete type
    - But a type constructor that takes one type param
    - Ex.: `Maybe Int` concrete type, `Maybe` type constructor

List as a functor

- `map` takes a f. from type `a` to `b`, a list of `a`, returns a list of `b`
  - `map` is just a `fmap` that works only on lists

HASKELL

```
map :: (a -> b) -> [a] -> [b]

instance Functor [] where
    fmap = map
```

HASKELL

```
Prelude> fmap (*2) [1..3]  -- same as map
[2,4,6]
```

- We didn't write `instance Functor [a]`, because `f` has to be a *type constructor* that takes one type
  - `[a]` is already a concrete type
  - `[]` is a type constructor that takes one type

Maybe as a functor

- Types that can act *like a box* can be functors
- Here's how `Maybe` is a functor

HASKELL

```
instance Functor Maybe where
    fmap f (Just x) = Just (f x)
    fmap f Nothing = Nothing
```

- We wrote `Functor Maybe` instead of `Functor (Maybe m)`
- Functor wants a type constructor that takes one type and not a concrete type
- Mentally replace each `f` with `Maybe`, or `Maybe m` (nonsense)
  - `(a -> b) -> Maybe a -> Maybe b`
  - `(a -> b) -> Maybe m a -> Maybe m b`

Mapping over a Maybe

- If it's an empty value of `Nothing`, then just return a `Nothing`
- If it's a single value packed up in a `Just`, then we apply the function on the contents of the `Just`

HASKELL

```
Prelude> fmap (++ " LOOK MA, INSIDE JUST") Nothing
Nothing
Prelude> fmap (++ " LOOK MA, INSIDE JUST") (Just "Stg serious.")
Just "Stg serious. LOOK MA, INSIDE JUST"
Prelude> fmap (*2) (Just 200)
Just 400
Prelude> fmap (*2) Nothing
Nothing
```

Tree as a functor

HASKELL

```
instance Functor Tree where
    fmap f EmptyTree = EmptyTree
    fmap f (Node x leftsub rightsub)
           = Node (f x) (fmap f leftsub) (fmap f rightsub)
```

HASKELL

```
Prelude> fmap (*4) EmptyTree
EmptyTree
Prelude> fmap (*4) (foldr treeInsert EmptyTree [5,7,3,2,1,7])
Node 28 (Node 4 EmptyTree (Node 8 EmptyTree ...
```

Either as a functor

- The `Functor` typeclass wants a *type constructor* that takes only one type parameter but `Either` takes two
- Partial application: `Either a` is a type constructor that takes one parameter

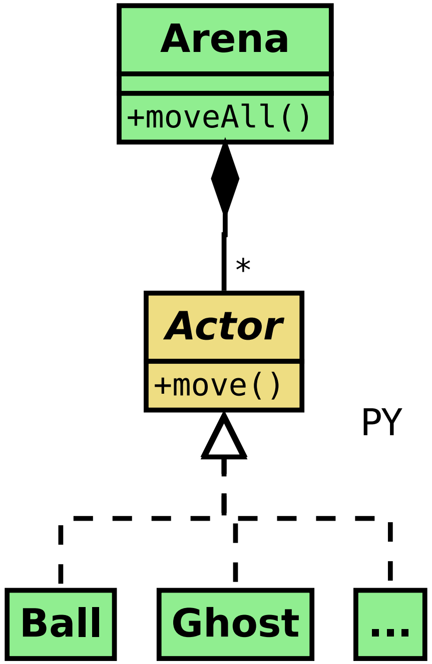
HASKELL

```
instance Functor (Either a) where
    fmap f (Right x) = Right (f x)
    fmap f (Left x) = Left x
```

- We mapped in the case of a `Right` value constructor, but we didn't in the case of a `Left`
  - To map one f. over both of them, a and b same type
  - The first parameter a (for `Left`) has to remain the same
  - Left part: ~ empty box, with an error message written on the side

Ex.: Actor typeclass

- Define a `Actor` typeclass, for things that can be moved
  - `move :: (Actor a) => a -> a`
- Create a container type for generic `Actor` things
  - *In Haskell: compile-time polymorphism!*
  - Cannot mix different *types* in a list, even if they are part of the same *typeclass*




```
class Arena:  # ...
    def __init__(self):
        self._actors = []
    def add(self, a: Actor):
        self._actors.append(a)
    def move_all(self):
        for a in self._actors:
            a.move()
```

<Domande?>

Michele Tomaiuolo
Palazzina 1, int. 5708
Ingegneria dell'Informazione, UniPR
sowide.unipr.it/tomamic