Programming Paradigms

Lecture 5. Introduction to Haskell

Outline

- What is Haskell?
- A program in Haskell
- Numeric types and common operations
- Defining functions with equations
- Boolean types
- Lists and tuples
- User-defined types
- Code.World platform

What is Haskell?

What is Haskell?

Simple and solid foundation
Easy* to reason about
Easy* to maintain safe and correct

What is Haskell?

Very expressive and practical

A program in Haskell

```
x = 2 + 2
```

A program in Haskell

```
x = 2 + 2

y = 3 * x - 1
```

A program (module) is a collection of declarations

A program in Haskell

```
x = 2 + 2

y = 3 * x - 1

z = length "Hello, world!" + 1
```

```
identifier = expression
```

A program (module) is a collection of declarations

A program in Haskell

```
x = 2 + 2

y = 3 * x - 1

z = length "Hello, world!" + 1

isGreater = (x > y)
```

```
identifier = expression
```

A program (module) is a collection of declarations

A program in Haskell

```
x = 2 + 2

y = 3 * x - 1

z = length "Hello, world!" + 1

isGreater = (x > y)

fivePrimes = [2, 3, 5, 7, 11]
```

```
identifier = expression
```

A program (module) is a collection of declarations

A program in Haskell

```
x = 2 + 2

y = 3 * x - 1

z = length "Hello, world!" + 1

isGreater = (x > y)

fivePrimes = [2, 3, 5, 7, 11]

myProgram = print 10 >> print 20
```

```
identifier = expression
```

A program (module) is a collection of declarations

Declarations

```
z = 2
```

This is declaration, not assignment!

Declarations

```
z = 2
z = 3
```

This is declaration, not assignment!

Declarations

```
z = 2
z = 3
```

This is declaration, not assignment!

```
<interactive>:3:1: error:
    Multiple declarations of ‘z’
    Declared at: <interactive>:2:1
                 <interactive>:3:1
```

Order of declarations

```
someMean = (a + b) / 2
a = 5.0
b = 7.0
```

Order of declarations does not matter

Local declarations

```
someMean = (a + b) / 2
  where
    a = 5.0
    b = 7.0
```

Haskell is indentation-sensitive!
Local declarations must be aligned.

Type signatures

```
x = 2 + 2
```

Type signatures

```
x :: Int
x = 2 + 2
```

Type signatures

```
x :: Int
x = 2 + 2
```

`x :: Int`
reads
“x has type `Int`”

Type signatures

```
x :: Int
x = 2 + 2
```

```
identifier :: type
identifier = expression
```

`x :: Int`
reads
“x has type `Int`”

Type signatures

```
x :: Int
x = 2 + 2

y :: Int
y = 3 * x - 1
```

```
identifier :: type
identifier = expression
```

Type signatures

```
someMean = (a + b) / 2
  where
    a = 5.0
    b = 7.0
```

Type signatures

```
someMean :: Double
someMean = (a + b) / 2
  where
    a = 5.0
    b = 7.0
```

Type signatures

```
someMean :: Double
someMean = (a + b) / 2
  where
    a :: Double
    a = 5.0

    b :: Double
    b = 7.0
```

Type signatures

```
someMean :: Double
someMean = (a + b) / 2
  where
    a :: Double
    a = 5.0

    b :: Double
    b = 7.0
```

```
identifier :: type
identifier = expression
  where
    <local declarations>
```

Numeric types: `Int`

```
n :: Int
n = 42

minInt, maxInt :: Int
minInt = -9223372036854775808
maxInt =  9223372036854775807
```

`Int` is a machine-sized integer.
Very efficient, big enough for most tasks.

Numeric types: `Integer`

```
bigNumber :: Integer
bigNumber = 3^(4^5)
```

```
3733918487410200435329597541848665882254097767837340077506369317220790406172652512299936889388039772204687650654314751581087270545921608585813513369828091873141917485942625809388070199519564042855718180410466812887974029255176680123406172983965747316191523867230462351259348960585905882846547935405059362023765478074427305821445270589887562514528177934133521419207446230275187291854328623757370639854853194764169262638199728870069070138992565242971985276987492741962768110607023337103564 81
```

`Integer` is a bignum integer (long arithmetic).
Also efficient and often optimized.
Bounded only by available memory.

Numeric types: `Double`

```
almostPi :: Double
almostPi = 3.14
```

Numeric types: `Double`

```
almostPi :: Double
almostPi = 3.14

wholeNumber :: Double
wholeNumber = 5
```

Integer literals can be interpreted with any numeric type.

Numeric operations: (+), (*), (-)

```
exampleInt :: Int
exampleInt = 20 + 3 * 47 - 10

exampleInteger :: Integer
exampleInteger = 20 + 3 * 47 - 10

exampleDouble :: Double
exampleDouble = 2 + 3 * 4.7 - 1
-- 15.100000000000001
```

Operations that work for all numeric types.

Numeric operations: `(/)`, `sin`, `cos`, `sqrt`, `log`, `...`

```
almostOne :: Double
almostOne = sin x * sin x + cos x * cos x
  where
    x = log (sqrt 10)

-- 1.0000000000000002
```

Operations that work for floating point numbers.

Numeric operations: `div, mod`

```
six :: Int
six = 55 `mod` 7

ten :: Integer
ten = 1 + (123456 `div` 12346)
```

Operations that work for integers.

Numeric operations: `div`, `mod`

```
six :: Int
six = 55 `mod` 7

ten :: Integer
ten = 1 + (123456 `div` 12346)

six :: Int
six = mod 55 7

ten :: Integer
ten = 1 + (div 123456 12346)
```

Put backticks around an identifier
to turn it into an infix operator.

Type conversion

```
n :: Int
n = 42

almostPi :: Double
almostPi = 3.14

bad = n + almostPi
```

Type conversion

```
n :: Int
n = 42

almostPi :: Double
almostPi = 3.14

bad = n + almostPi
```

```
<interactive>:20:11: error:
    • Couldn't match expected type ‘Int’ with actual type ‘Double’
    • In the second argument of ‘(+)’, namely ‘almostPi’
      In the expression: n + almostPi
      In an equation for ‘bad’: bad = n + almostPi
```

Explicit type conversion

```
n :: Int
n = 42

almostPi :: Double
almostPi = 3.14

good1 = (fromIntegral n) + almostPi
```

`fromIntegral` converts
from any integer type to any numeric type

Explicit type conversion

```
n :: Int
n = 42

almostPi :: Double
almostPi = 3.14

good1 = (fromIntegral n) + almostPi

good2 = n + (floor   almostPi) -- round down
good3 = n + (ceiling almostPi) -- round up
good4 = n + (round   almostPi) -- round to the closest int
```

Defining functions

```
square x = x * x
```

Defining functions

```
square :: Double -> Double
square x = x * x
```

Defining functions

```
square :: Double -> Double
square x = x * x

distance :: Double -> Double -> Double
distance x y = abs (x - y)
```

Defining functions

```
square :: Double -> Double
square x = x * x

distance :: Double -> Double -> Double
distance x y = abs (x - y)

factorial :: Integer -> Integer
factorial 0 = 1
factorial n = n * factorial (n - 1)
```

Boolean type

```
otherwise :: Bool
otherwise = True

lie :: Bool
lie = False
```

Boolean type

```
otherwise :: Bool
otherwise = True

lie :: Bool
lie = False

someMax :: Int
someMax


  where
    x = 5
    y = 3
```

Boolean type

```
otherwise :: Bool
otherwise = True

lie :: Bool
lie = False

someMax :: Int
someMax
  | x > y     = x

  where
    x = 5
    y = 3
```

Boolean type

```
otherwise :: Bool
otherwise = True

lie :: Bool
lie = False

someMax :: Int
someMax
  | x > y     = x
  | otherwise = y
  where
    x = 5
    y = 3
```

Boolean type

```
otherwise :: Bool
otherwise = True

lie :: Bool
lie = False

someMax :: Int
someMax
  | x > y     = x
  | otherwise = y
  where
    x = 5
    y = 3
```

```
identifier :: type
identifier arg1 ... argN
  | cond1 = expression1
  | cond2 = expression2
  | ...
  where
    <local declarations>
```

Char and String

```
name :: String
name = "Anna Karenina"

initial :: Char
initial = 'A'
```

Tuples

```
tuple1 = (123, "Hello", False)
```

Tuples

```
tuple1 :: (Int, String, Bool)
tuple1 = (123, "Hello", False)
```

Tuples

```
tuple1 :: (Int, String, Bool)
tuple1 = (123, "Hello", False)

tuple2 :: (Int, String, Bool)
tuple2 = (678, "Bye", True)
```

Tuples

```
tuple1 :: (Int, String, Bool)
tuple1 = (123, "Hello", False)

tuple2 :: (Int, String, Bool)
tuple2 = (678, "Bye", True)


student1 = ("Peter White", 4)
student2 = ("Jane Black", 5)
```

Tuples

```
tuple1 :: (Int, String, Bool)
tuple1 = (123, "Hello", False)

tuple2 :: (Int, String, Bool)
tuple2 = (678, "Bye", True)


student1 = ("Peter White", 4)
student2 = ("Jane Black", 5)

point3D = (1.0, -2.2, 5.3)
vector3D = (3.0, 0.0, 4.0)
```

A function of a tuple

```
len :: (Double, Double, Double) -> Double
len (x, y, z) = sqrt (x^2 + y^2 + z^2)
```

A function of a tuple

```
len :: (Double, Double, Double) -> Double
len (x, y, z) = sqrt (x^2 + y^2 + z^2)

vector3D = (3.0, 0.0, 4.0)

example = len vector3D
```

A function of a tuple

```
len :: (Double, Double, Double) -> Double
len (x, y, z) = sqrt (x^2 + y^2 + z^2)

vector3D = (3.0, 0.0, 4.0)

example = len vector3D
-- example
-- = len vector3D
```

A function of a tuple

```
len :: (Double, Double, Double) -> Double
len (x, y, z) = sqrt (x^2 + y^2 + z^2)

vector3D = (3.0, 0.0, 4.0)

example = len vector3D
-- example
-- = len vector3D
-- = len (3.0, 0.0, 4.0)
```

A function of a tuple

```
len :: (Double, Double, Double) -> Double
len (x, y, z) = sqrt (x^2 + y^2 + z^2)

vector3D = (3.0, 0.0, 4.0)

example = len vector3D
-- example
-- = len vector3D
-- = len (3.0, 0.0, 4.0)
-- = sqrt (x^2 + y^2 + z^2)
--     where x = 3.0, y = 0.0, z = 4.0
```

A function of a tuple

```
len :: (Double, Double, Double) -> Double
len (x, y, z) = sqrt (x^2 + y^2 + z^2)

vector3D = (3.0, 0.0, 4.0)

example = len vector3D
-- example
-- = len vector3D
-- = len (3.0, 0.0, 4.0)
-- = sqrt (x^2 + y^2 + z^2)
--     where x = 3.0, y = 0.0, z = 4.0
-- = sqrt (3.0^2 + 0.0^2 + 4.0^2)
```

A function of a tuple

```
len :: (Double, Double, Double) -> Double
len (x, y, z) = sqrt (x^2 + y^2 + z^2)

vector3D = (3.0, 0.0, 4.0)

example = len vector3D
-- example
-- = len vector3D
-- = len (3.0, 0.0, 4.0)
-- = sqrt (x^2 + y^2 + z^2)
--     where x = 3.0, y = 0.0, z = 4.0
-- = sqrt (3.0^2 + 0.0^2 + 4.0^2)
-- = ...
-- = 5.0
```

Empty tuple

```
emptyTuple = ()
```

Empty tuple

```
emptyTuple :: ()
emptyTuple = ()
```

Lists

```
list1 = [1, 2, 3]
```

Lists

```
list1 :: [Int]
list1 = [1, 2, 3]
```

Lists

```
list1 :: [Int]
list1 = [1, 2, 3]

list2 :: [String]
list2 = ["Hello", "world"]
```

Lists

```
list1 :: [Int]
list1 = [1, 2, 3]

list2 :: [String]
list2 = ["Hello", "world"]

list3 :: [Bool]
list3 = [False, True]
```

Lists

```
list1 :: [Int]
list1 = [1, 2, 3]

list2 :: [String]
list2 = ["Hello", "world"]

list3 :: [Bool]
list3 = [False, True]

list4 :: [(String, Int)]
list4 = [("Peter White", 4), ("Jane Black", 5)]
```

Lists

```
list1 :: [Int]
list1 = [1, 2, 3]

list2 :: [String]
list2 = ["Hello", "world"]

list3 :: [Bool]
list3 = [False, True]

list4 :: [(String, Int)]
list4 = [("Peter White", 4), ("Jane Black", 5)]

list5 :: [[Int]]
list5 = [[1], [2, 3], []]
```

Lists are homogeneous

```
bad = [1, "two"]
```

Lists are homogeneous

```
bad = [1, "two"]
```

```
<interactive>:21:8: error:
    • No instance for (Num [Char]) arising from the literal
‘1’
    • In the expression: 1
      In the expression: [1, "two"]
      In an equation for ‘bad’: bad = [1, "two"]
```

Lists are homogeneous

```
bad = [1, "two"]
```

One of the items in the list is a number
Another one is a `String` (which is list of `Char`s).
But `String`s are not `Num`bers!

```
<interactive>:21:8: error:
    • No instance for (Num [Char]) arising from the literal ‘1’
    • In the expression: 1
      In the expression: [1, "two"]
      In an equation for ‘bad’: bad = [1, "two"]
```

Lists are homogeneous

```
bad = [False, "two"]
```

```
<interactive>:22:15: error:
    • Couldn't match expected type ‘Bool’ with actual type ‘[Char]
    • In the expression: "two"
      In the expression: [False, "two"]
      In an equation for ‘bad’: bad = [False, "two"]
```

Lists are homogeneous

```
bad = [False, "two"]
```

The list started with a `Bool`.
But now we see a `String` (which is a list of `Char`s).
But `String` is not the same as `Bool`.

```
<interactive>:22:15: error:
    • Couldn't match expected type ‘Bool’ with actual type ‘[Char]
    • In the expression: "two"
      In the expression: [False, "two"]
      In an equation for ‘bad’: bad = [False, "two"]
```

Constructing lists

```
list0 = []
```

Constructing lists

```
list0 = []
list1 = 1 : list0     -- [1]
```

Constructing lists

```
list0 = []
list1 = 1 : list0      -- [1]
list2 = 2 : list1      -- [2, 1]
```

Constructing lists

```
list0 = []
list1 = 1 : list0      -- [1]
list2 = 2 : list1      -- [2, 1]
list3 = 3 : list2      -- [3, 2, 1]
```

Constructing lists

```
list0 = []
list1 = 1 : list0      -- [1]
list2 = 2 : list1      -- [2, 1]
list3 = 3 : list2      -- [3, 2, 1]

list4 = 4 : (3 : (2 : (1 : [])))
-- [4, 3, 2, 1]
```

Pattern matching lists

```
f []     =
f (x:xs) =
```

Pattern matching lists

```
f []     = something
f (x:xs) =
```

Pattern matching lists

```
f []     = something
f (x:xs) = somethingElse x xs
```

Pattern matching lists

```
f []     = something
f (x:xs) = somethingElse x xs

list5 = [1, 3, 5]

example = f list5
```

Pattern matching lists

```
f []     = something
f (x:xs) = somethingElse x xs

list5 = [1, 3, 5]

example = f list5
-- example
-- = f list5
```

Pattern matching lists

```
f []     = something
f (x:xs) = somethingElse x xs

list5 = [1, 3, 5]

example = f list5
-- example
-- = f list5
-- = f [1, 3, 5]
```

Pattern matching lists

```
f []     = something
f (x:xs) = somethingElse x xs

list5 = [1, 3, 5]

example = f list5
-- example
-- = f list5
-- = f [1, 3, 5]
-- = f (1 : [3, 5])
```

Pattern matching lists

```
f []     = something
f (x:xs) = somethingElse x xs

list5 = [1, 3, 5]

example = f list5
-- example
-- = f list5
-- = f [1, 3, 5]
-- = f (1 : [3, 5])
-- = somethingElse 1 [3, 5]
```

Pattern matching lists: sum

```
f []     = something
f (x:xs) = somethingElse x xs
```

```
sum :: [Double] -> Double
sum []     =
sum (x:xs) =
```

Pattern matching lists: sum

```
f []     = something
f (x:xs) = somethingElse x xs
```

```
sum :: [Double] -> Double
sum []     = 0
sum (x:xs) =
```

Pattern matching lists: sum

```
f []     = something
f (x:xs) = somethingElse x xs
```

```
sum :: [Double] -> Double
sum []     = 0
sum (x:xs) = x + sum xs
```

Type aliases

```haskell
-- | A 2D point.
type Point = (Double, Double)
```

Type aliases

```haskell
-- | A 2D point.
type Point = (Double, Double)

-- | A distance in kilometers.
type Kilometers = Double

-- | A distance in miles.
type Miles = Double

milesToKm :: Miles -> Kilometers
milesToKm miles = 1.609344 * miles
```

User defined types: product types

```haskell
-- | A 2D vector.
data Vector = MkVector Double Double
```

User defined types: product types

```haskell
-- | A 2D vector.
data Vector = MkVector Double Double
```

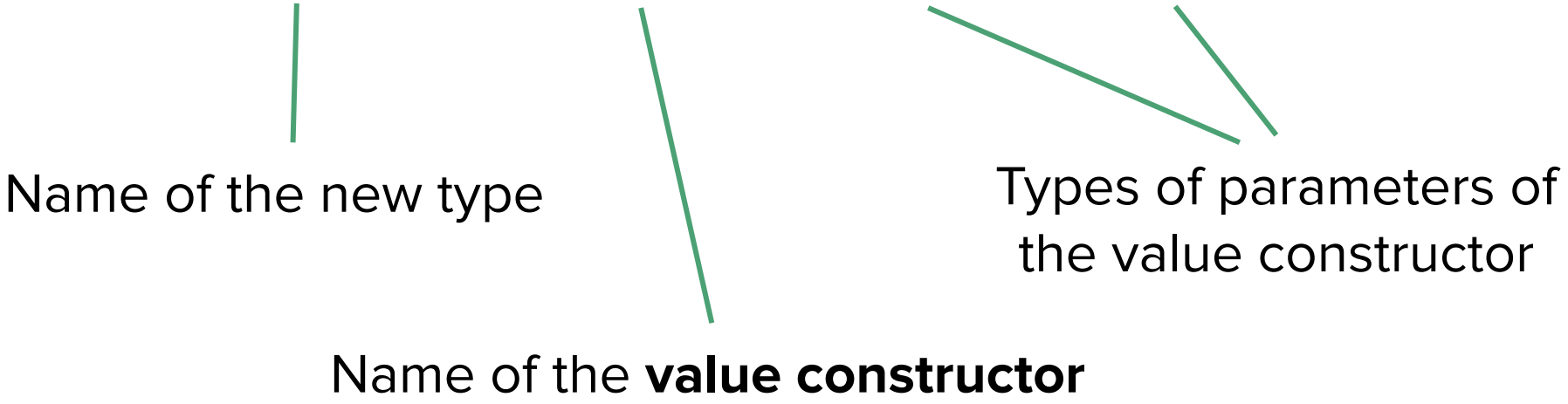

Name of the new type

User defined types: product types

```haskell
-- | A 2D vector.
data Vector = MkVector Double Double
```

User defined types: product types

```haskell
-- | A 2D vector.
data Vector = MkVector Double Double
```



User defined types: product types

```haskell
-- | A 2D vector.
data Vector = MkVector Double Double

someVector :: Vector
someVector = MkVector 3 4
```

User defined types: product types

```haskell
-- | A 2D vector.
data Vector = MkVector Double Double

someVector :: Vector
someVector = MkVector 3 4

vectorLen :: Vector -> Double
vectorLen (MkVector x y) = sqrt (x^2 + y^2)
```

User defined types: product types

```haskell
-- | A 2D vector.
data Vector = Vector Double Double

someVector :: Vector
someVector = Vector 3 4

vectorLen :: Vector -> Double
vectorLen (Vector x y) = sqrt (x^2 + y^2)
```

User defined types: product types

```haskell
-- | A 2D vector.
data Vector = Vector Double Double

data Student = Student Year Name Grade

data MenuItem = MenuItem Title Price
```

User defined types: wrapper types

```
data Kilometers = Kilometers Double

data Miles = Miles Double

milesToKm :: Miles -> Kilometers
milesToKm miles
  = 1.609344 * miles
```

User defined types: wrapper types

```
data Kilometers = Kilometers Double

data Miles = Miles Double

milesToKm :: Miles -> Kilometers
milesToKm miles
  = 1.609344 * miles
```

```
<interactive>:30:5: error:
    • Couldn't match expected type ‘Kilometers’
                  with actual type ‘Miles’
    • In the expression: 1.609344 * miles
      In an equation for ‘milesToKm’: milesToKm miles = 1.609344 * miles
```

User defined types: wrapper types

```
data Kilometers = Kilometers Double

data Miles = Miles Double

milesToKm :: Miles -> Kilometers
milesToKm miles
  = Kilometers (1.609344 * miles)
```

User defined types: wrapper types

```
data Kilometers = Kilometers Double

data Miles = Miles Double

milesToKm :: Miles -> Kilometers
milesToKm miles
  = Kilometers (1.609344 * miles)
```

```
<interactive>:39:17: error:
    • Couldn't match expected type ‘Double’ with actual type ‘Miles’
    • In the first argument of ‘Kilometers’, namely
        ‘(1.609344 * miles)’
      In the expression: Kilometers (1.609344 * miles)
      In an equation for ‘milesToKm’:
          milesToKm miles = Kilometers (1.609344 * miles)
```

User defined types: wrapper types

```
data Kilometers = Kilometers Double

data Miles = Miles Double

milesToKm :: Miles -> Kilometers
milesToKm (Miles miles)
  = Kilometers (1.609344 * miles)
```

Code.World platform

```haskell
import CodeWorld

myPicture :: Picture
myPicture = leftShape <> rightShape
  where
    leftShape = solidCircle 1
    rightShape =
      translated 3 0 (solidCircle 2)

main :: IO ()
main = drawingOf myPicture
```

https://code.world/haskell

Code.World Pictures

```
import CodeWorld

myPicture :: Picture
myPicture = leftShape <> rightShape
  where
    leftShape = solidCircle 1
    rightShape =
      translated 3 0 (solidCircle 2)

main :: IO ()
main = drawingOf myPicture
```

Code.World Pictures

```
import CodeWorld

myPicture :: Picture
myPicture = leftShape <> rightShape
  where
    leftShape = solidCircle 1
    rightShape =
      translated 3 0 (solidCircle 2)

main :: IO ()
main = drawingOf myPicture
```

```
(<>) :: Picture -> Picture -> Picture
```

Code.World Pictures

```
import CodeWorld

myPicture :: Picture
myPicture = leftShape <> rightShape
  where
    leftShape = solidCircle 1
    rightShape =
      translated 3 0 (solidCircle 2)

main :: IO ()
main = drawingOf myPicture
```

```
solidCircle :: Double -> Picture
```

Code.World Pictures

```
import CodeWorld

myPicture :: Picture
myPicture = leftShape <> rightShape
  where
    leftShape = solidCircle 1
    rightShape =
      translated 3 0 (solidCircle 2)

main :: IO ()
main = drawingOf myPicture
```

```
translated :: Double -> Double -> Picture -> Picture
```

Code.World Pictures

```
import CodeWorld

myPicture :: Picture
myPicture = leftShape <> rightShape
  where
    leftShape = solidCircle 1
    rightShape =
      translated 3 0 (solidCircle 2)

main :: IO ()
main = drawingOf myPicture


drawingOf :: ???
```

What what the most unclear part of the lecture for you?

See Moodle

Homework (self-study)

1. Install **Haskell** https://www.haskell.org/downloads/
2. Read **Learn you a Haskell for Great Good** Chapters 2, 4, and 5 http://learnyouahaskell.com/chapters
3. Test yourself by implementing a program that renders a Koch snowflake of a given rank in Haskell on Code.World platform (https://code.world/haskell):