Haskell – An Introduction

What is Haskell?

- General purpose
- Purely functional
  - No function can have side-effects
  - IO is done using special types
- Lazy
- Strongly typed
  - Polymorphic types
- Concise and elegant

A First Look

- Provides a REPL
  - **`ghci`** is the reference implementation
- But there's compiler – unfortunately we won't see the compiler in this talk!

Functions: 101

- Functions are called thus:

  func arg1 [argN+]

- Examples

```
id 5
succ 'a'
even 7
odd 3
```

Functions: The Basics

- A function that doubles its argument

```
doubleArg x = 2 * x
```

- A function that doubles odd arguments and returns even ones

```
doubleOddArg x = if odd x then
(2*x) else x
```

- Let's define and use these in the REPL.

Lists

- Lists in Haskell are *homogenous*
  - Store several elements of the identical type.
- Here's a list of integers

```
[1,2,3,4,5]
```

- Concatenating two lists

```
[1,2,3,4,5] ++ [6,7,8,9,10]
```

- Prepending an element

```
1:[2,3]
```

More on Lists

- Head

```
head [1,2,3,4,5] == 1
```

- Tail

```
tail [1,2,3,4,5] == [2,3,4,5]
```

- Last

```
last [1,2,3,4,5] == 5
```

- Init

```
init [1,2,3,4,5] = [1,2,3,4]
```

Yet more on Lists

- Get an element by its index (indexing starts at 0)

```
[1,2,3,4,5] !! 2
```

- Does a thing exist in a list

```
4 `elem` [1,2,3,4,5]
```

- Length of a list

```
length [1,2,3,4,5]
```

- Taking values

```
take 3 [1,2,3,4,5]
```

Last List

- Reverse

```
reverse [1,2,3,4,5]
```

- Drop elements from beginning of list

```
drop 3 [1,2,3,4,5]
```

- Sum elements

```
sum [1,2,3,4,5]
```

- Product of elements

```
product [1,2,3,4,5]
```

Ranges

- Can create a list with a sequence of values

  `[1..20]` is a list containing numbers 1 to 20.

  `['a'..'z']` is a list containing lowercase letters.

- Creating a range with a step

  `[2,4..20]` is a list of even numbers between 2 and 20

  `['a','c'..'z']` is a list of the letters a, c, e, g, I, k , m, o, q, s, u, w, and y.

  `[20,19..1]` is a list of numbers from 20 to 1.

Infinite Lists

- Infinite list with a range

  `[1..]` an infinite list of numbers starting at 1.

- The cycle function

  `cycle [1,2,3]` generates an infinite list [1,2,3,1,2,3,1,2,3...]

- The repeat function

  `repeat 7` generates an infinite list of 7s.

List Comprehensions

- Apply a function to each element in a list

```
[x*2 | x <- [1,2,3,4,5]]
```

- For each number in the range `[1,2,2,3,5]`
  - **`x`** is bound to the current number
  - The function **`x*2`** is applied to **`x`**
- We can filter the list(s)

```
[x*2 | x <- [1,2,3,4,5], odd x]

[x*y | x <- [1..4], y <- [1..4],
       x /= 3, y /= 2]
```

Tuples

- Store several values of different type
- Useful for when you know exactly how many values you'll combine
- Tuples type depends on how many components it has and the types of the components
- E.g. A list of tuples is type safe:

```
(3,'c',9):[(1,'a'),(4,'d'),(7,'g')]
```

is illegal!

Tuples continued

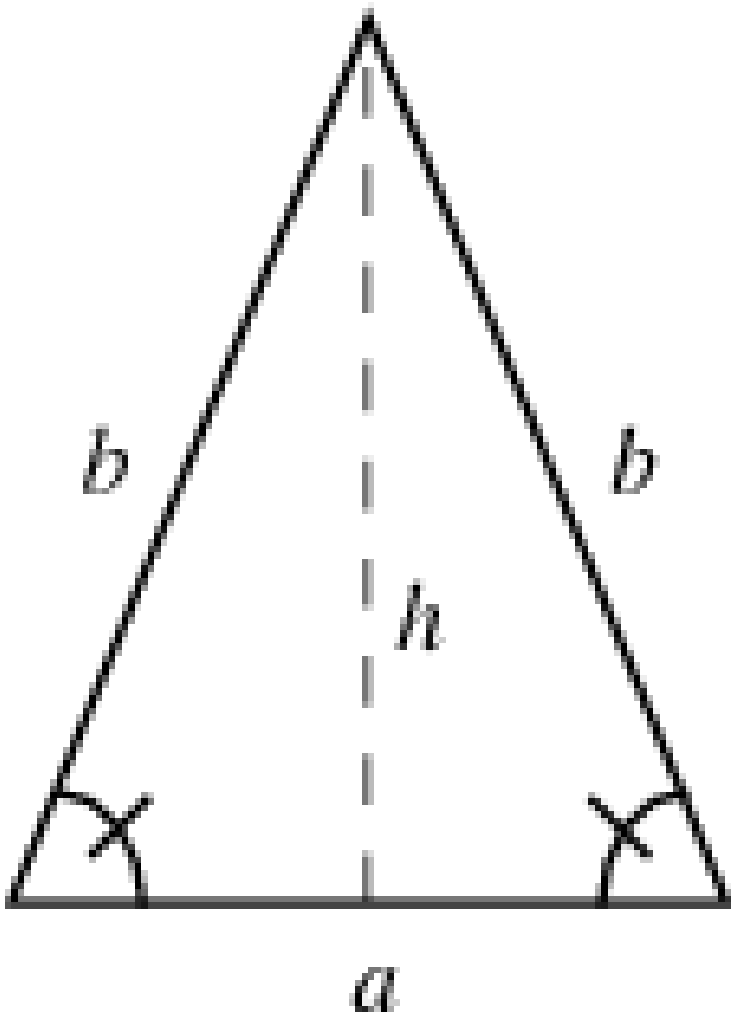
- Singleton tuples cannot exist
  - It's just a value!
- Pairs, though, have their own functions
  - **`fst`** – returns the first element a 2-tuple
  - **`snd`** – returns the second element of a 2-tuple
- Lists of pairs can be generated from two lists using the **`zip`** function

```
zip [1,2,3] ['a','b','c']
```

creates the list **`[(1,'a'),(2'b'),(3,'c')]`**

A Problem

- Find Isosceles triangles, that have integer length sides, whose perimeter is less than 6 units in length. Using 1<=a<=5 and 1<=b<=10.



Types

- Haskell is statically and strongly typed
- Uses type inference
  - Hindley-Milner type system
  - The programmer doesn't need to inform the compiler of a value's type.
- We can use the `:t` command to interrogate Haskell as to the type of a value
  - Scalar types: Bool, Int, Integer, Char
  - Lists: [ ], [Char]
  - Tuples: (Int, Bool), (Bool, [Char])

Common Types

- **`Int`** – Bounded integer type. On 32-bit platforms the range is [-2147483648 , 2147483647]
- **`Integer`** – Unbounded integer type
- **`Float`** – Single precision floating point
- **`Double`** – Double precision floating point
- **`Bool`** – Boolean type, True and False values
- **`Char`** – Character type, single quotes used, e.g. 'a'

Function Types

- A function, say, **`addThreeInts`**

```
addThreeInts :: Int -> Int -> Int -> Int

addThreeInts x y z = x + y + z
```

  - `::` is read as “has type of”
- This function take three **`Int`** types and returns an **`Int`** type
- The last type is the return type
- **`:t addThreeInts`** returns

```
addThreeInts :: Int -> Int -> Int -> Int
```

More Function Types

- **`:t removeUppercase`** returns

  `removeUppercase :: [Char] -> [Char]`

- This function takes a list of characters, a string, and returns a list of characters.
  - The **`String`** type is usually used
  - It is type synonym for **`[Char]`**

Typeclasses

- A typeclass is an interface that defines some behaviour
  - They are similar to Java interfaces
- `:t (==)` returns

```
(==) :: Eq a => a -> a -> Bool
```

- The equality function takes two values of the same type, **`a`**.
  - The type **`a`** must a member of the **`Eq`** typeclass
  - It is a *class constraint*
- The equality function returns a boolean value

`Ord` typeclass

- **`:t (>=)`** returns

```
(>=) :: Ord a => a -> a -> Bool
```

- **`Ord`** is a typeclass that defines the comparison functions >, <, >=, <=
- Compare with the **`compare`** function!
  - **`:t compare`** returns

```
compare :: Ord a => a -> a -> Ordering
```

- The **`Ordering`** type can hold the values **`GT`**, **`LT`** or **`EQ`**.

`Show` and **`Read`** typeclass

- Members of the **`Show`** typeclass can be represented as strings
  - Use the **`show`** function
- Members of the **`Read`** typeclass can take strings and a type that is a member of **`Read`**
  - Use the **`read`** function

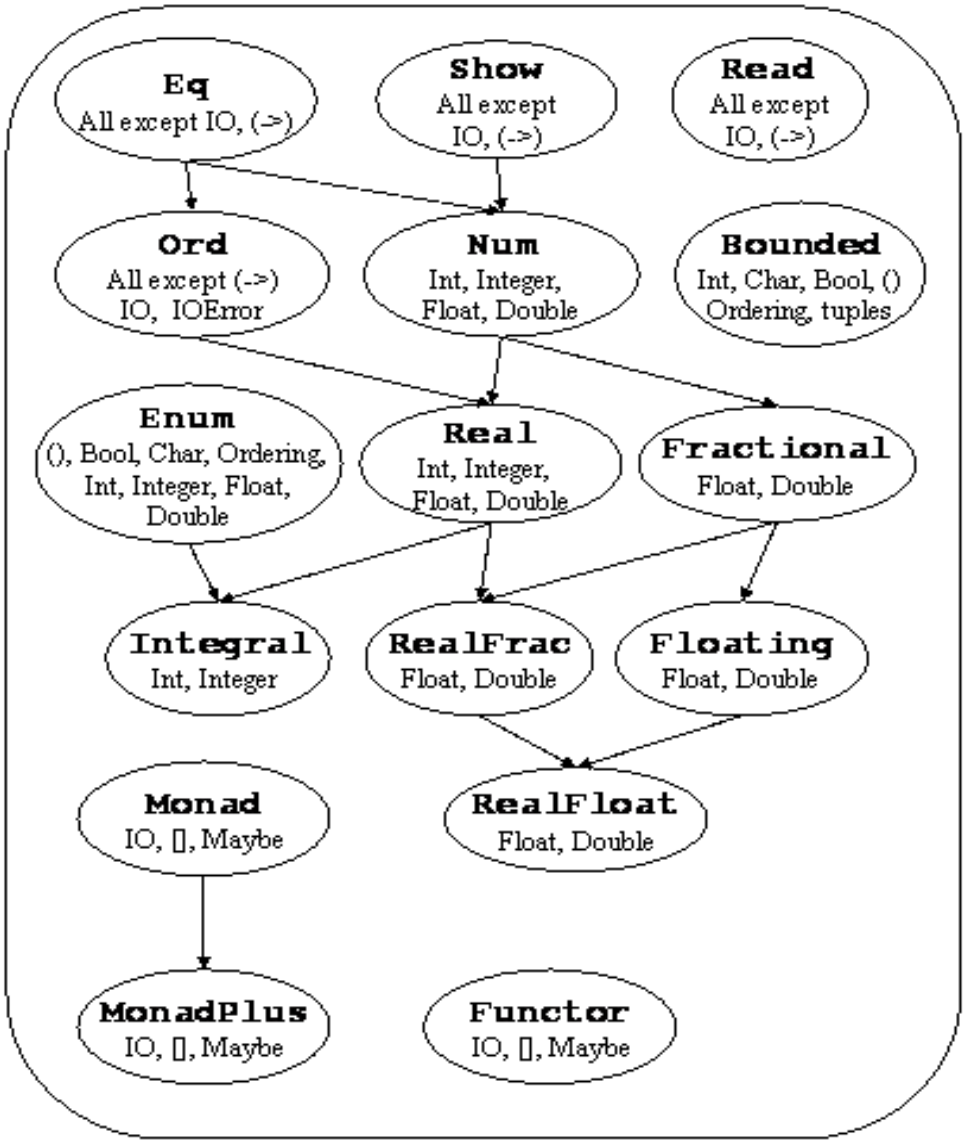
`Enum` typeclass

- `Enum` members can be enumerated
  - They are sequentially ordered
- The `pred` and `succ` functions can be used on these members
  - `succ 2`
  - `pred 'b'`
- Can be used in ranges
  - `['a'..'z']`
  - `[LT .. GT]`

Numeric typeclasses

- **`Num`** is a numeric typeclass
  - Members (**`Int`**, **`Integer`**, **`Float`**, **`Double`**) act like numbers
- **`Integral`** is a typeclass for integer numbers
  - Members are **`Int`** and **`Integer`**
- **`Floating`** is a typeclass for real numbers
  - pi, exp, log, sqrt, sin, cos, tan etc...
  - Members are **`Float`** and **`Double`**
- **`Fractional`** is a type class for number that can be used in division

Standard Haskell Classes



Functions, again

- Pattern matching
  - Specifies a pattern which some data should conform
  - If the data matches the pattern then that data is deconstructed

```
magicNumber :: (Integral a) => a -> String

magicNumber 13 = “You won!”

magicNumber x = “You lose.”
```

More Pattern Matching

- Implementation of factorial

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

Pattern Matching, again

- Adding pairs

```
addPairs :: (Num a) => (a, a) -> (a,a) -> (a,a)
addPairs (a1, a2) (b1, b2) = (a1+b1, a2+b2)
```

- Ignoring values

```
second :: (Num a) => (a, a, a)
second (_, b, _) = b
```

Pattern Matching Lists

- Sum the elements in a list

```
sum' :: (Num a) => [a] -> a
sum' [] = 0
sum' (x:xs) = x + sum'(xs)
```

- Head of a list

```
head' :: (Num a) => [a] -> a
head' [] = error “Invalid list”
head' (x:_) = x
```

- Length of a list

```
length' :: (Num a) => [a] -> a
length'[] = 0
length' (_:xs) = 1 + length' xs
```

As Patterns

- As patterns match data whilst keeping a reference to the whole thing
- Report the first letter

```
first' :: String -> String

first' "" = "Empty string, whoops!"

first' all@(x:xs) = "The first letter of "++all++" is "++[x]
```

Guards

- Guards are used to test the values of inputs to functions

```
councilTaxBand :: (Num a) => a -> Char

councilTaxBand value
    | value <= 40000 = 'A'
    | value <= 52000 = 'B'
    | value <= 68000 = 'C'
    | value <= 88000 = 'D'
    | value <= 120000 = 'E'
    | value <= 160000 = 'F'
    | value <= 320000 = 'G'
    | otherwise 'H'
```

Where

- Where bindings a visible everywhere
- BMI calculator

```
bmiTell :: (RealFloat a) => a -> a -> String
bmiTell weight height
    | bmi <= underweight = "Underweight"
    | bmi <= normal = "Normal"
    | bmi <= overweight = "Overweight"
    | otherwise = "Obese"
    where bmi = weight / height ^ 2
          (underweight, normal, overweight) = (18.5, 25.0, 30.0)
```

Let

- Let bindings are local
- Volume of a cone

```
volCone :: (Num a) => a -> a -> a
volCone radius height =
   let thirdPi = 1/3 * pi
       rh = height * radius ^ 2
   in thirdPi * rh
```

Currying

- Every Haskell function takes only 1 parameter!
- These two expressions are equivalent

```
(+) 7 3
((+) 7) 3
```

- (+) function is defined as

```
(+) :: Num a => a -> a -> a
(+) :: Num a => a -> (a -> a)
```

- Applying too few parameters will return a *partially applied* function.

More Currying

- Consider the following function

```
addThree :: (Num a) => a -> a -> a -> a
addThree x y z = x + y + z
```

- Evaluate addThree 6 3 9
  - 6 is applied and a partially applied function is returned
  - 3 is applied to the partially applied function and returns another partially applied function
  - 9 is applied to this new partially applied function and a value is returned

Currying Example

- Multiply by 4

  multFour :: (Num a) => a → a

  multFour = (* 4)

Map and Filter

- **`map`** is a function that takes a function and applied it to every element in the list

```
map (+7) [1,2,3,4,5] == [8,9,10,11,12]
```

- **`filter`** is a function that takes a predicate function and returns a list whose elements satisfy the predicate

```
filter (< 9) [4,6,9,10,45,3] == [4,6,3]
```

Lambdas

- Useful for when you only need a function once
- Anonymous functions using \ character

```
map (\x -> 7 + x) [1,2,3,4,5] == [8,9,10,11,12]
filter (\x -> x < 9) [4,6,9,10,45,3] == [4,6,3]
```

What I Didn't Tell You

- How to define your own typeclasses and types
- Functors, Applicative Functors, Monoids and Monads
- Haskell wraps up IO in an IO Monad
- Haskell can implement code in modules
- But all that will be in a future talk

Haskell Resources

- Haskell.org
  - One stop shop for everything Haskell
  - http://www.haskell.org/
- Learn You a Haskell for Great Good! by Miran Lipovača.
  - http://learnyouahaskell.com/
- A Gentle Introduction to Haskell by P. Hudak, J. Peterson, and J. Fasel
  - http://www.haskell.org/tutorial/

More Haskel Resources

- Try Haskell
  - http://tryhaskell.org/
- The Haskell 2010 report
  - http://www.haskell.org/onlinereport/haskell2010/