Computer Science 312

Haskell Data Types and Functions

Basic Haskell Data Types

Data Type	Example Values	What It Is
`Bool`	`True`, `False`	The two truth values.
`Char`	`'a'`, `'!'`	The set of characters (on the Latin keyboard and others as well).
`Double`	3.14, 0.001	The set of double-precision floating-point numbers.
`Float`	3.14, 0.001	The set of single-precision floating-point numbers.
`Int`	0, 67, 1000000	The set of integers ranging from -2^{31} to $2^{31} - 1$.
`Integer`	Includes the `Int`s as well as greater magnitudes	A range of integers limited only by computer memory.

Check ‘em out with `:type`

```
Prelude> :type 'a'
'a' :: Char

Prelude> :type "Hi there"
"Hi there" :: [Char]

Prelude> :type 45
45 :: Num a => a

Prelude> :type odd
odd :: Integral a => a -> Bool

:type 3.14
3.14 :: Fractional a => a

Prelude> :type sqrt
sqrt :: Floating a => a -> a
```

`Num`, **`Integral`**, **`Fractional`**, and **`Floating`** are ***type classes***

What Is a Type Class?

- A type class allows you to group together several types (also called ***type instances***) into a single category
- For example, the types **`Int`**, **`Integer`**, **`Float`**, and **`Double`** are instances of the type class **`Num`**
- Type classes allow you to ***overload*** operators for different types of data
- Type classes are organized in a ***type hierarchy***

Variables and Assignment

```
Prelude> pi
3.141592653589793

Prelude> area = pi * 6 ^ 2       -- New in GHCI 8.2.2

Prelude> area
113.09733552923255
```

Variables are not capitalized.

Single assignment only in a module, but not in the GHCI.

To **`let`** or Not to **`let`**

```
Prelude> pi
3.141592653589793

Prelude> let area = pi * 6 ^ 2

Prelude> area
113.09733552923255
```

Same semantics as simple assignment, but allowed in the GHCI only.

Defining Simple Functions

```
Prelude> square n = n * n

Prelude> square 3
9

Prelude> cube n = n * (square n)

Prelude> cube 3
27

Prelude> between n low high = low <= n && n <= high

Prelude> between 5 1 10
True
```

Look like simple equations

Type Inference

```
-- Function to square an integer
square n = n * n

-- Function to cube an integer
cube n = n * square n

-- Function to test for inclusion in a range
between :: Integer -> Integer -> Integer -> Bool
between n low high = low <= n && n <= high
```

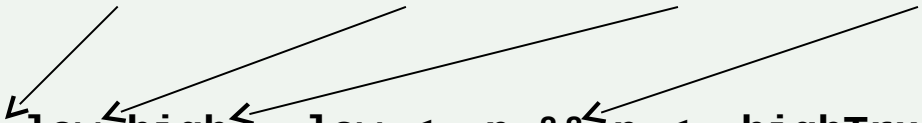
Haskell infers the types of the parameters and the function's return value if the type signature is not included

Including type signatures is standard practice, however

Type Signatures

```
-- Function to test for inclusion in a range
between :: Integer -> Integer -> Integer -> Bool


between n low high = low <= n && n <= high
```



Specify the types of parameters and the return type of the function, at compile time

Include Type Signatures

```
{-
File: NewMath.hs
Author: Ken Lambert
Purpose: provides some simple math functions
-}

module NewMath where

-- Function to square an integer
square :: Integer -> Integer
square n = n * n

-- Function to cube an integer
cube :: Integer -> Integer
cube n = n * square n

-- Function to test for inclusion in a range
between :: Integer -> Integer -> Integer -> Bool
between n low high = low <= n && n <= high
```

Generalize the Types

```
{-
File: NewMath.hs
Author: Ken Lambert
Purpose: provides some simple math functions
-}

module NewMath where

-- Function to square an integer
square :: Num a => a -> a
square n = n * n

-- Function to cube an integer
cube :: Num a => a -> a
cube n = n * square n

-- Function to test for inclusion in a range
between :: Ord a => a -> a -> a -> Bool
between n low high = low <= n && n <= high
```

Simple Recursive Functions

n! = 1, when n = 1
n! = n * (n – 1)!, when n > 1

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

A recursive function has at least two equations, or ***clauses***

One equation states the base case

The other equation states the recursive step

For next time

Making choices

Dealing with errors

Local contexts with **`where`**