

# Real World Haskell

## Table of Contents

### Getting started

- `hugs`: interpreter primarily used for teaching
- `ghc`: Glasgow Haskell Compiler, used for real work
- `ghci`: REPL for Haskell
- `runghc`: program for running Haskell programs as scripts without compilation
- `PreLude`: standard library of useful functions
- Haskell requires type names to start with an uppercase letter, and variable names to start with a lowercase letter

### Types and Functions

- Haskell types are: strong, static, and can be automatically inferred, making it safer than popular statically typed languages, and more expressive than dynamically typed languages. Much of the debugging gets moved to compile time
- strength refers to how permissive a type system is, with a stronger type system accepting fewer expressions as valid than a weaker one
- **strong**: type system guarantees a program cannot errors from trying to write expressions that don't make sense
- **well typed expressions** obey the languages type rules
- Haskell doesn't perform **automatic coercion**
- **static**: compiler knows the type of every value and expression at compile time before any code is executed
  - compiler detects when you try to use expressions whose types don't match
  - makes type errors at runtime impossible
- **type inference**: compiler can automatically deduce the types of most expressions
- **type signature**: `:: Type`
- function application is left-associative: `a b c d` is equivalent to `((a b)c)d`
- **side effect**: dependency between global state of the system and the behaviour of a function
  - invisible inputs to/outputs from functions
- **pure function**: has no side effects, the default in Haskell
- **impure function**: has side effects

- can be identified by type signature: the result begins with `IO`
- variables in Haskell allow you to bind a name to an expression. This permits substitution of the variable for the expression
- **lazy evaluation**: aka non-strict evaluation. Track unevaluated expressions as **thunks** and defer evaluation until when it is really needed
- **parametric polymorphism**: most visible polymorphism supported by Haskell, that has influenced the generics and templates of Java/C++. This is the ability to specify behaviour without knowing the type.
- Haskell doesn't support **subtype polymorphism** as it isn't object oriented, nor does it support **coercion polymorphism** as a deliberate design choice to avoid automatic coercion
- in `ghci` you can list the type of an expression using `:t` or `:type`

### Comment on Purity

- makes understanding code easier: you know things the function cannot do (e.g. talk to the network), what valid behaviours could be, and it is inherently modular, because each function is self-contained with a well-defined interface
- pure code makes working with impure code simpler: code that must have side effects can be separated from code that doesn't need side effects. Impure code is kept simple, with heavy lifting in pure code.
- minimises attack surface

### Type constructors

`[]` and `(,)` are **type constructors**: they take types as input and build new types from them

### `String` `s`

`String` in Haskell is a **type synonym** with `[Char]`

### Defining type synonyms

- similar to C's `typedef`

```
1 type Pair = (Int, Int)
```

## Type Classes

- use **type classes** to restrict applicable types in a function with parametric polymorphism
- type classes are like interfaces in Java: if you have implementation of functions `+`, `-`, and other numerical operations, it can be considered a `Num`

e.g. for `sum`:

```
1 Num a => [a] -> a`
```

- `Num`: collection of types for which addition, multiplication, and other numerical operations make sense
- `Ord`: collection of types for which comparison operations (e.g. `<`, `>`, `==`) are defined

## Type Definitions

Define a new type with the `data` keyword. Possible values are separated by `|`

```
1 -- e.g. our own implementation of Bool
2 data MyBool = MyTrue | My Falsek
3
4 -- e.g. point to store 2D Cartesian coordinates
5 data Point = Pt Float Float
```

Here we have defined a `Point`, which can be a `Pt` which also carries two `Floats`

Typically, you use the same name for the type and data constructor:

```
1 data Point = Point Float Float
```

## Recursive Data Types

e.g. implementation of linked list: here's a type `List`, which can be a `ListNode` carrying with it an `Int`, and another `List` value. Otherwise, it can be a `ListEnd` (just a constant)

```
1 data List = ListNode Int List | ListEnd
```

- `ListNode 20 (ListNode 10 ListEnd): List` containing 20 and 10

To make this polymorphic with respect to type, introduce type parameter `a`:

```
1 data List a = ListNode a (List a) | ListEnd
```

Now `List` is a type constructor, rather than a type. To get a type, you need to provide `List` with the type to use, e.g. `List Int`.

- `List Char` roughly corresponds to Java's `LinkedList<Character>`

### Defining operations on custom types

- `Eq`: type class for which equality makes sense
- `show`: provides string representation
- `Show`: type class that can be converted to string representation
- to automatically generate default behaviour (i.e. two values are equal when they have the same structure, and show strings that look like the code you use to write the values):

```
1 data List a = ListNode a (List a) | ListEnd
2   deriving (Eq, Show)
```

### Binary Tree

```
1 data Tree a = Node a (Tree a) (Tree a) | Empty
2   deriving Show
3
4 tree :: Tree Int
5 data Tree a = Node a (Tree a) (Tree a) | Empty
6   deriving Show
7
8 -- returns contents of a tree
9 elements :: Tree a -> [a]
10 elements Empty = []
11 elements (Node x l r) = elements l ++ [x] ++ elements r
12
13 -- insert element into binary search tree
14 insert n Empty = (Node n Empty Empty)
15 insert n (Node x l r)
16   | n == x = (Node x l r)
17   | n <= x = (Node x (insert n l) r)
18   | n > x = (Node x l (insert n r))
19
20 -- build a binary search tree from list of values
21 buildtree :: Ord a => [a] -> Tree a
22 buildtree [] = Empty
23 buildtree [x] = insert x Empty
24 buildtree (x:xs) = insert x (buildtree xs)
25
26
27 -- build a BST then return sorted values
```

```

28 treesort :: (Ord a) => [a] -> [a]
29 treesort [] = []
30 treesort x = elements (buildtree x)

```

## List Comprehension

```
[ expression | generator ]
```

- **generator**: pull items from list one at a time to operate on
- **expression**: what to do with each item of the generator

```

1 > [x^2 | x<- [1..6]]
2 [1,4,9,16,25,36]
3
4 > [(-x) | x <- [1..6]]
5 [-1,-2,-3,-4,-5,-6]
6
7 > [even x | x <- [1..6]]
8 [False,True,False,True,False,True]

```

- you can also add **filters** to restrict which input items should be processed

```

1 > [x^2 | x <- [1..12], even x]
2 [4, 16, 36, 64, 100, 144]

```

- multiple generators and filters can be combined:
  - each generator introduces a variable that can be used in filters
  - each filter cuts down the input items which proceed to subsequent filters/generators
  - the further down a generator is on the list, the faster it will cycle through its values
- nested generators and filters are analogous to nested **for** loops and **if** statements as you would use in the procedural approach
- using variables in later generators

```
1 [ x | x <- [1..4], y <- [x..5], even (x+y) ]
```

- Pythagorean triples

```

1 pyth :: [(Integer,Integer,Integer)]
2 pyth = [(a,b,c) | c <- [1..], a <- [1..c], b <- [1..c], a^2 + b^2 == c
           ^2]

```

Output:

```
1 Prelude> take 5 pyth
2 [(3,4,5), (4,3,5), (6,8,10), (8,6,10), (5,12,13)]
```

## zip

`zip xs ys`: returns a list of pairs of elements  $(x, y)$

```
1 > zip [1,2,3,4] [5,6,7,8]
2 [(1,5), (2,6), (3,7), (4,8)]
```

- e.g. compute dot product with list comprehension

```
1 dot xs ys = sum [x*y | (x, y) <- zip xs ys]
```

## Modules

- import using `import Module.Name` keyword
- define a module using

```
1 module Module.Name
2 where
```

## Lazy evaluation

You can think of execution in a pure functional language as evaluation by rewriting through substitution

e.g. with the following definitions:

```
1 f x y = x + 2*y
2 g x = x^2
```

You can evaluate the following by **applicative order/call by value**, rewriting the expression at the innermost level first:

```
1 g (f 1 2)
2 = g (1 + 2*2) -- use f's definition
3 = g (1 + 4)
4 = g 5
5 = 5^2         -- use g's definition
6 = 25
```

Alternatively, you can evaluate it using **normal order/call by name**, where you pass an un-evaluated expression, rewriting the outermost level first:

```
1 g (f 1 2)
2 = (f 1 2)^2    -- use g's definition
3 = (1 + 2*2)^2  -- use f's definition
4 = (1 + 4)^2
5 = 5^2
6 = 25
```

Haskell uses normal order evaluation, unlike most programming languages. This will always produce the same value as applicative order evaluation, but sometimes produces a value when applicative order evaluation would fail to terminate (e.g. asking for a result on an infinite list).

Haskell uses **call by need**: a function's argument is evaluated at most once if needed, otherwise never. This evaluation isn't all-or-nothing: Haskell is **lazy** in that it only evaluates *on demand*. This allows Haskell to operate on infinite data structures: **data constructors** are simply functions that are also lazy (e.g. `cons` :)

e.g. `take 3 [1..]` evaluates to `[1,2,3]`

## Infinite Lists

Here is an efficient Fibonacci implementation that uses a recursive definition of the infinite sequence as `fibs`:

```
1 fibs :: [Integer]
2 fibs = 1 : 1 : zipWith (+) fibs (tail fibs)
3
4 fib n = fibs !! n
```

Here's another example for an infinite sequence of powers of 2:

```
1 powers :: [Integer]
2 powers = 1 : map (*2) powers
```

## What drives evaluation?

```
1 -- zipWith takes a 2-argument function and 2 lists and applies the
   function element-wise across the lists
2 zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
3 zipWith f [] _ = []
4 zipWith f _ [] = []
5 zipWith f (x:xs) (y:ys) = f x y : zipWith f xs ys
```

Let's consider how Haskell evaluates `zipWidth f e1 e2`:

- Haskell does a pattern match on the 3 cases defined: this requires a small amount of evaluation of `e1` and `e2` to determine that they are non-empty lists (for the 1st and 2nd cases respectively)
- `zipWith` causes the *spines* of `e1/e2` to be evaluated until one of the lists is exhausted
- `zipWith` doesn't cause any of the list elements to be evaluated

## Sieve of Eratosthenes

Algorithm for computing primes:

1. write out list of integers `[2..]`
2. put the list's first number `p` aside
3. remove all multiples of `p` from the list
4. repeat from step 2

This works with infinite lists, removing an infinite number of multiples. Haskell can evaluate using it:

```
1 primes :: [Integer]
2 primes = sieve [2..]
3   where sieve (x:xs) = x : sieve [y | y <- xs, y `mod` x > 0]
```

Now you can generate lists of primes rapidly:

```
1 *Main> take 100 primes
2 [2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,71,73,79,83,89,97,101,103,107,
3  283,293,307,311,313,317,331,337,347,349,353,359,367,373,379,383,389,397,401,409,419,
```

## I/O and Monads

- **monad**: representation or encapsulation of computational action

To preserve purity in computation, Haskell:

- uses typing to distinguish pure functions from effectual functions. While all functions return a result, some functions also have associated actions. The type system distinguishes between the two, helping isolate side effects and enabling the mixing of pure functions with effectful monads in a principled manner
- actions/things of monadic type should be able to work on lists, actions can be elements of lists, etc.



- Haskell describes recipes for action, which can be combined to create more complex recipes
- when its time for the actions to occur, you call `main :: IO ()`
- monads allow sequencing, dereferencing, destructive assignment, I/O etc. to be expressed within a pure functional language. The resulting programs appear imperative but retain the properties of pure functional programs
- I/O is an example of monadic programming

## Input Actions

Input returns a value. Something of type `IO a` is an input/output action which returns a result of type `a` to the caller.

```
1 getChar :: IO Char
```

## Output Actions

Output actions have type `IO ()`, an instance of a monad.

```
1 putChar :: Char -> IO ()
2 print  :: Show a => a -> IO ()
```

- `>>` is used to put actions in a sequence.
- `a >> b` denotes the combined action of `a` followed by `b`

```
1 (>>) :: IO () -> IO () -> IO ()
```

- `()`: the **unit type**, with a single inhabitant `()`. Used for I/O actions that return nothing of interest. Used to represent no value.

## File I/O

```
1 type FilePath = String
2 readFile :: FilePath -> IO String
3 writeFile :: FilePath -> String -> IO ()
4 appendFile :: FilePath -> String -> IO ()
```

## Binding and Sequencing

```

1 -- bind: pass result of first action to the next
2 (>>=) :: IO a -> (a -> IO b) -> IO b
3 -- sequence: second action doesn't care about result of the first
  action
4 (>>)  :: IO a -> IO b -> IO b
5 return :: a -> IO b

```

Lambda abstraction for `f`: `\x -> f x` function that takes an argument `x` and return `f x`

`>>` is defined in terms of `>>=`:

```

1 m >> k = m >>= \_ -> k

```

e.g. read an input file and write to an output file, excluding non-ASCII characters:

```

1 main
2   = readFile "inp" >>= \s ->
3     writeFile "outp" (filter isAscii s) >>
4     putStr "Filtering successful\n"

```

## Do

Syntactic sugar time: write things under **do**

```

1 action1 >>= \x -> action2 using x

```

becomes:

```

1 x <- action1
2 action2 using x

```

```

1 action1 >> action2

```

becomes:

```

1 action1
2 action2

```

e.g. purely functional code that asks user for input, reads/writes a file, writes to standard out

```

1 import Data.Char(isAscii)
2
3 main
4   = do
5     putStr "Input file: "
6     ifile <- getLine
7     putStr "Output flie: "
8     ofile <- getLine

```

```
9      s <- readFile ifile
10     writeFile ofile (filter isAscii s)
11     putStrLn "All done"
```

- Haskell is layout sensitive: you need to indent actions the same or else they won't be considered part of the `do` expression