

Functional Programming in Haskell for A level teachers

About this document

Functional Programming is now part of the A level curriculum. This document is intended to get those who already have some programming experience in an imperative language (such as Python or Java) up and running in Haskell, one of the functional languages recommended by AQA specification 7516.

Important!

Please note, if you have no experience of programming, then this document is not for you. Try learning an imperative language such as Python first.

This is far from being a full course in Haskell, it's purely intended to get you through the A level syllabus. Many important features of Haskell are omitted, the most important being how Haskell handles types – you'll need to learn about these if you want to learn Haskell properly.

A good place to start is <http://learnyouahaskell.com/> by Miran Lipovaca

Getting Started

Installation

Before you start, you'll need to install the **Haskell Platform**. This contains, amongst other things, the Haskell compiler **GHC** and the Haskell interpreter **GHCI**. If you install the Windows version you'll also get WinGHCI, a simple GUI for the interpreter.

<https://www.haskell.org/platform/index.html>

Using the Haskell Interpreter

Once everything's installed, open the Haskell interpreter by either running `ghci` in a terminal or opening WinGHCI in Windows.

The Haskell interpreter will load, showing the `Prelude>` prompt. `Prelude` refers to the standard module imported by default into all Haskell modules: it contains all the basic functions and types you need.

Try out some arithmetic operators as follows:

```
Prelude> 3+4
7
Prelude> (5*6)+7
37
Prelude> 2^12
4096
```

Saving Your Work

Haskell programs aren't intended to be written as line after line of code, but rather as a collection of functions. Haskell programs are a little like a spreadsheet: you write lots of functions that call each other.

Here's an example of how to save your functions and load them into the interpreter.

1. Open your favourite text editor (Notepad, Notepad++, TextEdit, Emacs, Vim, ...) and type in some Haskell functions. I've given two simple examples below.

```
areaCircle x = 3.14 * x
areaSquare x = x * x
```

1. Save the file. I've saved my file as **c:/haskell/example.hs**
2. Run the Haskell interpreter
3. Type **:l c:/haskell/example.hs** to load your functions into the interpreter. You should see that the prompt now says **Main>**. **Main** refers to the module you just loaded. Note, you still have access to the **Prelude** functions
4. Experiment using your functions in the Haskell interpreter. If you make changes to your functions, hit **:r** to reload the file.

```
areaCircle 3
9.42
areaSquare 4
16
```

Exercise

Write a functions to work out the following

1. The perimeter of circle
2. The perimeter of a square
3. The perimeter of a rectangle

Lists

AQA Quick Reference

The following is taken from the AQA syllabus:

Be familiar with representing a list as a concatenation of a head and a tail. Know that the head is an element of a list and the tail is a list.

Know that a list can be empty.

Describe and apply the following operations:

- return head of list
- return tail of list
- test for empty list
- return length of list
- construct an empty list
- prepend an item to a list
- append an item to a list.

Have experience writing programs for the list operations mentioned above in a functional programming language or in a language with support for the functional paradigm

SYLLABUS	HASKELL EXAMPLE
Create a list	let xs = [1,2,3,4,5]
return head of list	head xs
return tail of list	tail xs
test for empty list	null xs

SYLLABUS	HASKELL EXAMPLE
return length of list	length xs
construct an empty list	xs = []
prepend an item to a list	element : xs
append an item to a list	xs ++ [element]

Going beyond the specification, there are many more list examples here:

https://wiki.haskell.org/How_to_work_on_lists

Using Lists

Haskell lists are homogenous: all the elements must be the same type.

- [1,2,3] OK
- ['a','b','c'] OK
- [1,'b',2] X Not allowed

A string is simply a list of characters

“This” = ['T', 'h', 'i', 's']

Haskell lists have a **head** and **tail**, they also have an **init** and a **last** (see below for examples of these). You can prepend an element to a list (add it to the front) using the **:** operator, and concatenate (join) two lists using the **++** operator.

Use the **:** operator for preference in Haskell: it's much faster than **++**

Here are some examples to illustrate the above

```
Prelude> let xs = [1,2,3,4,5]

Prelude> head xs

1

Prelude> tail xs
```

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

```
Prelude> init xs
```

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

```
Prelude> last xs
```

```
5
```

```
Prelude> tail xs ++ init xs
```

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

```
Prelude> head xs ++ last xs
```

```
<interactive>:15:1:
```

```
    No instance for (Num [a0]) arising from a use of `@it`^
```

```
    In a stmt of an interactive GHCi command: print it
```

```
Prelude> [head xs] ++ [last xs]
```

```
[1,5]
```

```
Prelude> xs!!2
```

```
3
```

```
Prelude> xs!!6
```

```
*** Exception: Prelude.(!!): index too large
```

```
Prelude> xs!!0
```

```
1
```

```
Prelude> 0:xs
```

```
[0,1,2,3,4,5]
```

```
Prelude> xs ++ 6
```

```
<interactive>:25:1:
```

```
No instance for (Num [a0]) arising from a use of it
```

```
In a stmt of an interactive GHCi command: print it
```

```
Prelude> xs ++ [6]
```

```
[1,2,3,4,5,6]
```

List Exercise

1. Write a list containing the days of the week
2. Find the head of the list
3. Find the tail of the list
4. Find the last element of the list
5. Find the last but one element of the list
6. A new day is invented: `HaskellDay`. Prepend this to the list

Remember that a string is a list of characters. Let `name = <your name>`

1. Find the first character in your name
2. Find the last character
3. Find the length of your name.
4. Find all the characters but the last.
5. What output will the following produce?

```
let ls = [1,2,3,4,5]
```

```
last ls: init ls ++ tail ls ++ [head ls] ++ [ls!!3]
```

- Why is `[head ls]` written as a list, and not as an element, eg `head ls`?

A Brief Diversion: List Comprehensions and Ranges

List Comprehensions aren't mentioned on the AQA syllabus, but they're too powerful a feature not to take a look at. It's worth understanding how they work: similar functionality has been introduced into languages such as Java.

Let's start with some ranges

```
Prelude> [1..5]
```

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

```
Prelude> [1,3..10]
```

```
[1,3,5,7,9]
```

```
Prelude> [10,9..1]
[10,9,8,7,6,5,4,3,2,1]
Prelude> [-5,-3..5]
[-5,-3,-1,1,3,5]
```

Now for some list comprehensions. The following examples show how to **draw down** from **ranges** in a list comprehension

```
Prelude> [x*2 | x <- [1..5]]
[2,4,6,8,10]
Prelude> [x*x | x <- [1..10]]
[1,4,9,16,25,36,49,64,81,100]
```

You can add predicates to restrict the values of a list comprehension as follows

```
Prelude> [x | x <- [1..10], odd x]
[1,3,5,7,9]
```

You can add more than one predicate. What are the even numbers between 1 and 50 that are divisible by 3?

```
Prelude> [x|x<-[1..50], x `mod` 3==0, even x]
[6,12,18,24,30,36,42,48]
```

You can **draw down** from two variables as follows. Watch out for the order! Note that $x^{**}y$ means x to the power of y

```
Prelude> [x**y | x <- [1..5], y <- [2,3,4]]
[1.0,1.0,1.0,4.0,8.0,16.0,9.0,27.0,81.0,16.0,64.0,256.0,25.0,125.0,625.0]
Prelude> [x**y | y <- [2,3,4], x <- [1..5]]
[1.0,4.0,9.0,16.0,25.0,1.0,8.0,27.0,64.0,125.0,1.0,16.0,81.0,256.0,625.0]
```

List Comprehension Exercise

Use list comprehensions to produce the following lists:

1. [5,10,15,20,25,30,35,40,45,50,55,60]
2. [0.5,0.4,0.3,0.2,0.1,0]
3. [3,2,1,0,-1,-2,-3]
4. [1,8,27,64,125,216,343,512,729,1000]
5. [1,3,5,7,9]
6. [100,102,104,106,108,110,112,114,116,118,120]

Haskell and Lazy Evaluation

Haskell doesn't work things out until it has to – this is called lazy evaluation.

This means you can write down things that might not lead to errors in imperative languages.

For example

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

The above means take the first 5 elements from the infinite list that counts up from 1. Haskell only creates as much of the list as it needs.

Combining lazy evaluation with functions that produce infinite lists means you can do such things as the following

```
Prelude> take 10 (cycle [1,2])
[1,2,1,2,1,2,1,2,1,2]

Prelude> take 5 (repeat "Brubeck")
["Brubeck","Brubeck","Brubeck","Brubeck","Brubeck"]
```

Summary

- Haskell allows you to use list comprehensions to work out lists of numbers.
- A list comprehension draws down from a range (e.g. $x \leftarrow [1..10]$) or a number of ranges.
- You can apply predicates (e.g. odd or even) to your list comprehension to decide what goes in it.
- List comprehensions allow you to solve problems in a completely different way to imperative programming languages. For example, here's how you'd find all the pythagorean triples (numbers such that $a^2 = b^2 + c^2$) for $a, b, c < x$.

```
pythagTriples x = [(a, b, c) | a <- [1..x], b <- [1..x], c <- [1..x],
c^2 == a^2 + b^2]
```


More on Functions

More than one parameter

Here are the two Haskell functions used in the Getting Started section:

```
areaCircle x = 3.14 * x
areaSquare x = x * x
```

Note that Haskell functions must start with lower case letters.

Remember, whilst you are learning you can type up functions in your favourite editor and then load them into the editor using `:l path/to/file`. Use `:r` to reload the functions when you've made changes to them.

Here's how to write functions with two parameters:

```
areaRectangle l w = l*w
perimeterRectangle l w = 2*l + 2*w
```

Partial Application

AQA defines partial application as the process of applying a function by creating an intermediate function by fixing some of the arguments to the function

As an example, let's consider the `areaRectangle` function above. Suppose you want to work out the areas of all rectangles where one side is fixed at 3. You can write a function, `area3Rect`, as follows

```
area3Rect = areaRectangle 3
```

You can now work out the areas of different rectangles as follows

```
*Main> area3Rect 4
12
*Main> area3Rect 5
15
*Main> area3Rect 6
18
```

```
*Main>
```

area3Rect is a partially applied function – a function where some of the parameters have been fixed.

Try creating a partially applied function based on perimeterRectangle.

This leads us nicely on to Higher Order Functions...

Higher Order Functions

AQA Quick Reference

A function is higher-order if it takes a function as an argument or returns a function as a result, or does both.

Have experience of using the following in a functional programming language:

- map
- filter
- reduce or fold.

map is the name of a higher-order function that applies a given function to each element of a list, returning a list of results.

filter is the name of a higher-order function that processes a data structure, typically a list, in some order to produce a new data structure containing exactly those elements of the original data structure that match a given condition.

reduce or fold is the name of a higher-order function which reduces a list of values to a single value by repeatedly applying a combining function to the list values.

SYLLABUS	EXAMPLE
map	map (+3) [1,2,3,4,5] -> [4,5,6,7,8]
filter	filter (>3) [1,2,3,4,5] -> [4,5]
fold	foldl (+) 0 [1..10] -> 55

Map

The Map function applies a function to every element of a list. In other words, it's a function that takes a function as a parameter, in other words a higher order function.

Here we map the function (+3) to the list

```
*Main> map (+3) [1,2,3,4,5]
[4,5,6,7,8]
```

Here we map the odd function...

```
*Main> map odd [1..10]
[True,False,True,False,True,False,True,False,True,False]
```

Filter

The filter function filters a list according to a *predicate* – a function that returns true or false. Filter is therefore a higher order function, a function that takes a (predicate) function as a parameter.

Here we filter the numbers >3 from the list.

```
*Main> filter (>3) [1,2,3,4,5]
[4,5]
```

Here we filter out the odd numbers in a list.

```
*Main> filter (odd) [1..20]
[1,3,5,7,9,11,13,15,17,19]
```

A string in Haskell is treated as a list of letters. ``elem`` returns true if the letter is an element of the list so...

```
*Main> filter (`elem` ['A'..'Z']) "What Are The Capitals In This Sentence?"
"WATCITS"
```

Fold

A fold has three parts. A function, and accumulator and a list to work on.

Haskell gives you two types of fold, `foldl` which folds from the left side of a list, and `foldr` which folds from the right.

Fold works its way through a list one item at a time, performing an operation on that item and storing it in the accumulator.

This is probably best demonstrated with a few examples

```
Prelude> foldl (+) 0 [1..10]
55
Prelude> foldr (+) 0 [1..10]
55
Prelude> foldl (-) 0 [1..10]
-55
Prelude> foldr (-) 0 [1..10]
-5
```

The first example is quite straightforward. `foldl` takes the first item from the list (1) adds it to the accumulator (0) and stores the result in the accumulator (1)

It then takes the second item from the list (2) adds it to the accumulator and stores the result (3). Working through the list you get the result 55.

The second and third examples are similar.

The last example is particularly interesting, however. Why does it give the result -5? Try and work through the logic, and remember that if you subtract a -ve number its the same as adding

You can use fold in Haskell like you use loops in imperative languages

Exercises

1. Use the `map` function on a list `[1..5]` to produce a list `[2..6]`
2. Use the `map` function on a list `[1..10]` to produce a list of even numbers `[2..20]`
3. Use the `filter` function to find the odd numbers between 1 and 30
4. Use the `filter` function to find the numbers < 4 in the list `[1..10]`
5. Use the `foldl` function to add up the numbers from 1 to 100
6. Use the `foldr` function to find $4!$ ($4 \times 3 \times 2 \times 1$)

Beyond the AQA specification

The following are not part of the current specification. I've included in case you want to get more of a taste of Haskell...

Pattern Matching

Haskell allows pattern matching. The following function counts one, two or many objects

```
simpleCount 1 = "One"
simpleCount 2 = "Two"
simpleCount x = "Many"
```

You can use pattern matching to set base cases in recursive functions. Here's an example of a factorial function from later on. Note how factorial 0 is defined as being 1 using pattern matching.

```
factorial 0 = 1
factorial n = n * factorial (n-1)
```

ifs and guards

Haskell allows you to use ifs and guards in your functions. A guard is a more readable version of an if.

People can learn to drive from the age of 17 in the UK. The following function uses to an if statement to check if someone is old enough to drive

```
canDrive x = if x <18 then "Too young to drive" else "Old enough to drive"
```

Here it is using guards:

```
canDrive' x
  | x<18 = "Too young to drive"
  | otherwise = "Old enough to drive"
```

The following function uses guards to work out the cost in pence of sending a large letter in the UK

```
letterCost weight
```

```
| weight <= 100 = 96  
  
| weight <= 250 = 127  
  
| weight <= 500 = 171  
  
| otherwise = 2.46
```

Looking at the above `letterCost` function you might reasonably deduce that you could send an elephant via the UK postal service for £2.46. Unfortunately, the prices given are only for large letters which can weigh no more than 1kg.

show

What if you want to mix text and numbers when using ifs and guards?

For example, in the game of fizz, you say “Fizz” if a number is divisible by 3, otherwise you just say the number. Writing a function to implement this can cause a problem in Haskell, as the function will have to return text or a number. One way round this is to convert the number to text using `show`, as follows.

```
fizz n  
  
| n `mod` 3 == 0 = "Fizz"  
  
| otherwise = show n
```

Pattern Matching, Ifs and Guards Exercise

Write the following functions:

1. A function that returns “Zero” if 0 is passed, then “Odd Number” or “Even Number” if an odd or even number is passed.
2. A grade function that calculates students grade as follows: A >50, B >40, C >30 otherwise fail
3. A fizz function, the returns “Fizz” if a number is divisible by three, otherwise it just returns the number
4. A buzz function, the returns “Buzz” if a number is divisible by five, otherwise it just returns the number
5. A FizzBuzz Function that returns “FizzBuzz” if a number is divisible by 3 and 5, Fizz if it’s divisible by three and Buzz if it’s divisible by five. Otherwise, just return the number.

Functions and List Comprehensions

Let’s define a function that uses a list comprehension to find the factors of n

```
factors n = [x | x <- [1..n], n `mod` x == 0]
```

Now, remembering that a number n is prime if and only if it has two factors, $[1,n]$, let's define function to determine if a number is prime

```
prime n = factors n == [1,n]
```

```
*Main> factors 15
```

```
[1,3,5,15]
```

```
*Main> factors 7
```

```
[1,7]
```

```
*Main> prime 7
```

```
True
```

```
*Main> prime 2
```

```
True
```

Oops, 2 is not a prime number. Use pattern matching to fix the prime function...

```
prime 2 = False
```

```
prime n = factors n == [1,n]
```

Check that...

```
*Main> prime 7
```

```
True
```

```
*Main> prime 2
```

```
False
```

Done!

Recursive Functions

Pattern matching is very useful when writing recursive functions. Recursive functions work very well on Haskell: it was designed for them. Here's an example of a recursive function in Haskell

```
factorial 0 = 1
```

```
factorial n = n * factorial (n-1)
```

As you can see, pattern matching makes it very easy to set the base case for a recursive function. Here's another recursive function. This one reverses a list. I've called my function `reverse'` to distinguish it from the existing Haskell `reverse` function,

```
reverse' [] = []  
reverse' (x:xs) = reverse(xs) ++ [x]
```

There are two important things to note here:

First, pattern matching is used to ensure the case of an empty list is handled.

Second, note the use of the `(x:xs)` pattern to identify the head and the tail of the list. Haskell programmers use this pattern a lot.

Recursion Exercise

Here some recursion problems from the Daily Java. See if you can solve them using Haskell

1. The first 6 triangle numbers are 0, 1, 3, 6, 10, 15. The n th triangle number is $1 + 2 + 3 + \dots + n$. Write a recursive method to find the n th triangle number
2. Write a recursive method that returns m to the n th power, e.g. 2 to the power of 3 returns 8.
3. The Harmonic Series begins $1 + 1/2 + 1/3 + 1/4 + 1/5 + \dots$. Write a recursive method that finds the Harmonic Series up to the n th term.
4. The Fibonacci Series begins 1,1,2,3,5,8,13,... The next digit is the sum of the last two digits, so $1 + 1 = 2$, $1 + 2 = 3$ etc. Write a recursive method to print the n th fibonacci number

Lambda Functions

Lambda functions are sometimes called anonymous functions. Quite simply, they are a function without a name. It's often more convenient to not bother naming a function when you're only going to use it once.

Lambda is a Greek letter that looks like this: λ

Haskell uses a `\` as it looks a little like a lambda.

Here's an example of a lambda function that doubles a number

```
*Main> (\x -> x*2) 3  
6
```

Here's another lambda function that multiplies two numbers

```
*Main> (\x y -> x*y) 3 4
```



```
12
```

The above example is an anonymous version of the `areaRectangle` function mentioned earlier.

Here's how to use a lambda function to find the numbers that divide by 3

```
*Main> filter (\x -> x `mod` 3 == 0) [1..20]
[3,6,9,12,15,18]
```

Putting it all together: Perfect Numbers

Here's a Haskell way of solving a problem, using some of the things learned so far.

A perfect number is a number that is the sum of its factors. 6 is a perfect number as its factors are 1, 2 and 3 and $1 + 2 + 3 = 6$.

Find all the perfect numbers < 10000

We know from earlier that we can find the factors of a number using the following function

```
factors n = [x | x <- [1..n], n `mod` x == 0]
```

Here's an example

```
*Main> factors 6
[1,2,3,6]
```

Remember that a number is perfect if it's the sum of its factors, not including the number itself, so we add an extra predicate to eliminate that.

```
factors n = [x | x <- [1..n], n `mod` x == 0, x/=n]
```

Check this

```
*Main> factors 6
[1,2,3]
```

So a number is perfect if $\text{sum}(\text{factors } x) == x$. Lets run a list comprehension to find the perfect numbers <1000.

```
*Main> [x | x <- [1..1000], sum (factors x) == x]
[6,28,496]
```

Or use a filter...

```
*Main> filter (\x -> sum(factors x) == x) [1..1000]
[6,28,496]
```

Try running that on [1..10000] to see how long it takes Haskell to figure out the next perfect number!