

The Frightened Freshers Guide To CO326 - Functional Programming (Haskell)

Thomas Paul Carlson

April 11, 2006

1 Introduction

This document serves the purpose of being a digitalised form of my notes for Functional Programming. Please note that this document does not cover the logic section of the course and concentrates on haskell and its basic syntax.

If you have come to CO326 after taking CO320 then the change from Java to Haskell will be quite a shocker. Haskell has no variables that you can store things into, it is case sensitive, True is spelled True and not true like in Java and the compiler error messages suck. However, Haskell is a powerful language to be writing in because of the way you can just write equasions and be given output. Although it does not have a huge audience outside of an academic field of vision, Haskell does have several applications like AI, Processing of XML, Compilers, Bio informatics and several other topics.

2 What is a function?

A lot of the time when programming in haskell you will come across the wonderous word function. A function is something that takes one or more inputs, does some processsing and then outputs something else. In haskell, when defining a function you need to tell the compiler what your inputs and outputs are and of what type they are.

Here is a simple example for a function called addTwoNumbers that shows the function declaration.

```
addTwoNumbers :: Int->Int->Int
```

You can see from this that the function is called addTwoNumbers and that it two Int inputs and a single Int output. In a more general form, the functions are declared as `functionName :: type1parameter-¿type2parameter-¿returnType`, note that you may have as many inputs as you like but there is only ever a single thing returned.

All functions that we know of will be declared in this way with a method name, a pair of colons and then the inputs and output of the method.

Your function declarations should include a number of things. Firstly, a type declaration to name the function and decide on what inputs and outputs your function will take and return. A comment describing what the function does is also very important. Examples showing what happens when certain things are inputted, what is outputted. Then the actual function definition itself (Which we will see more of in a second) and finally testing the function to see if the function agrees with the examples.

3 Doing something with our function

Taking the example of `doubleNumber` above, we can extend it so that it actually does something. The function `addTwoNumbers` will take two integers and add them together, returning the result. Addition in Haskell is (Thankfully) the same as in Java, just using the `+` sign between two parameters. The best way to see this is in an example.

```
addTwoNumbers :: Int->Int->Int
— This function takes two numbers and adds them together
addTwoNumbers a b = a + b
```

The section below the function declaration is a comment. Code should always include comments! Without comments, if you come back to a function at some time later, then you may find that you need to re-read the code to figure out what it actually does. The third line in this little section of code is important. Firstly, we have the function name repeated and then the letters `a` and `b`. These letters are the inputs from the function and without them you will get an ugly looking error message. This is what happens when you try and remove `b` from the example above. Note that the function is completely useless!

```
addTwoNumbers :: Int->Int->Int
— This function takes two numbers and adds them together
addTwoNumbers a = a
```

```
Hugs.Base> :l t.hs
ERROR "t.hs":3 - Type error in explicitly typed binding
*** Term           : addTwoNumbers
*** Type           : Int -> Int
*** Does not match : Int -> Int -> Int
```

This is one of the few error messages in Haskell that I think is useful. It is telling me that in the function `addTwoNumbers` it was expecting to find `Int Int Int`, but instead it found `Int Int`. This shows that you need to add

b back in because when you define a function that wants two inputs and an output you do need to use all the inputs!

Once that is fixed, you can load it up into hugs (If you do not know how to load hugs and do stuff with it, see the section at the end of the document...) and type in "addTwoNumbers 1 2" and Hugs will come back with an answer of Three.

Note that you do not need to have brackets and commas around your arguments for a function, just using whitespace to break up the function name and its parameters is good enough for Haskell.

4 Importing other modules

If your program gets to be very, very long then you will probably want to break it down into several files. You can import modules into your program using import. The example below shows loading a module called Pictures.

```
import Pictures
```

5 Step-by-step evaluation

If we define a function called myMax which returns the max number from a pair of inputs:

```
myMax x y = if x >= y then x else y
```

It is possible to evaluate this step-by-step:

```
— myMax 7 3
— if 7 >= 3 then 7 else 3
— if True then 7 else 3
— 7
```

Although this is a very simple example, all functions no matter how complex can be broken down in this way into their component parts and evaluated in a simple way. With a very large function this may take quite some time, and one of the major disadvantages of the hugs haskell interpreter is that you can not tell it to step through the evaluation. Debugging may involve a manual walkthrough of the code!

6 Conditional statements (If-then-else)

Like a great deal of other languages, there are conditional statements that will control the flow of logic in a program. The way in which an if statement is constructed is slightly different to what you may be used to though as Haskell relies on whitespace to figure out just what is part of the if and

what is not. Below is a short example of using an if statement in a haskell function.

```
myMax :: Int->Int->Int
— Finds which one of the parameters is the highest
myMax x y =
    if x >= y
        then x
        else y
```

Hopefully (if LaTeX or Latex2html do not eat my formatting) you can see that an if statement needs its then and else parts to be indented at least as far as the if. This also means that the function above can be re-written in a slightly shorter form:

```
myMax :: Int->Int->Int
— Finds which one of the parameters is the highest
myMax x y = if x >= y then x else y
```

This is still legal because the then and else come after the if. Haskell relies on layout to do a lot of its processing. In Java it would be common to wear out your semicolon key on your keyboard. Haskell does not enforce this on you and relies on you to set out programs neatly. You can also do slightly more interesting if statements that have more than one condition to be true.

```
if ((x == y) && (y == z)) then x else y
```

Although that little bit of code did not really have a point to it it is worth noting the double ampersand signs to represent logical AND. If we were to feed this little bit of code with values for x y and z of 1 2 and 2 then (x==y) would evaluate to False, (y==z) to True and (False) && (True) equals False. Therefore the else section of code would be executed and y would be the output.

It is worth noting that haskell uses lazy evaluation! If the first bit of an if statement with an && happens to turn out to be False then the second bit will be skipped over by hugs.

7 Types in Haskell

Although we have seen a few datatypes so far (Int and Boolean) there are a number of other types! Heres a nice table to show you some of the types that can be used.

Type	Description
Int	Stores whole numbers
Float	Stores decimals
Bool	Stores True or False
Char	Stores single characters
String	Stores lots of chars

Note that all the datatypes start with a capital letter. We can define as many new types as we like, for example we may want to define a new datatype that contains the Months of the year, days in a week or abstract things like a directory of names and passwords, filesystem layout and so on. There are two type definitions, Enumerated types and alternatives. Bool is an example of an enumerated type as it has two elements which are True and False. Alternatives could contain other types. For example, if we had two types defined called Circle and Square then a Shapes datatype could have Circle or Square contained inside of it.

8 Enumerated Types

The season datatype shown below has four elements, each separated by a pipe symbol. This declaration is known as a ‘Data Constructor’.

```
data Season = Spring | Summer | Autumn | Winter
```

And another example:

```
data Temp = Hot | Cold
```

Now, using these two user-defined datatypes we can make a function that will, when fed a season, will return if that season is generally hot or cold.

```
Weather :: Season -> Temp
Weather Spring = Cold
Weather Summer = Hot
Weather Autumn = Cold
Weather Winter = Cold
```

This is known as Pattern Matching and is very, very useful indeed! What if we want to return a String from a Season? Well, we could do:

```
show Spring = "Spring"
```

but that is bad because the data constructor itself already has something built in to handle this kind of thing.

```
data Season = Spring | Summer | Autumn | Winter
           deriving (Show, Eq, Ord)
```

Wow, what on earth does that mean? Well, we know that `Season` is a datatype that has four elements. The deriving part has three distinct little bits. `Show` converts whatever `Season` to a `String` when required. `Eq` stands for Equality and allows testing if two seasons are the same. `Ord` means this datatype is ordered and two seasons can be compared to see which is bigger. The code below shows examples of this.

```
Winter == Winter -- would return True
Winter >= Spring -- would return True
```

Here is another example using the `Shape` datatype defined in lectures and classes.

```
data Shape = Circle Float | Rectangle Float Float
           deriving (Show, Eq, Ord)
```

— *So now we can define Circles and Rectangles*
— *as Shapes*

```
Circle 1.3
Rectangle 14.1 114.2
```

— *And from this we can define a function to*
— *check if any given shape is round or not*

```
isRound :: Shape -> Bool
isRound (Circle r) = True
isRound (Rectangle h w) = False
```

Interestingly, the `Bool` (`True` or `False`) is just another datatype. You can ask `hugs` about the type of these and they are in fact ordered!

```
Hugs.Base> :t True
True :: Bool
Hugs.Base> :t False
False :: Bool
Hugs.Base> True > False
True
Hugs.Base> False > True
False
Hugs.Base> False == False
True
```

From this we can say that the datatype `Bool` is probably defined something like this (Although in reality it is probably something different.)

```
data Bool = False | True
           deriving (Show, Eq, Ord)
```

9 Recursion

Recursion is the practice of including a function in its own definition. First of all, we really need to know how to define a recursive datatype for use in such a recursive function. The example below is a list of integers.

```
data ListInt = Nil | Cons Int ListInt
           deriving (Show, Eq, Ord)
```

Although this does seem very odd at first, it makes perfect sense in reality! The two options for things of type ListInt are either Nil (So the list will end here) or Int ListInt in which there will be a single Int and then another ListInt inside of the current ListInt. This can go on for as long as you like. Valid values for this datatype include:

```
Nil
Cons 42 Nil
Cons 42 (Cons 6 Nil)
```

You can also do pattern matching on datatypes such as the ListInt datatype that was defined earlier. The functions below return the first element in a list and the rest of a list respectively.

```
headList :: ListInt -> Int
headList Nil = error "Whoops, nothing in this list"
headList (Cons x xs) = x
```

```
tailList :: ListInt -> ListInt
tailList Nil = error "Argh, this list is empty!"
tailList (Cons x xs) = xs
```

In each of these definitions you can check how it works based on the recursive definition of ListInt. The first one, headList, will assign the variables x and xs to the first (an Int) and last (A ListInt) parts of the data fed to it and thus spit out the first Integer in the ListInt. The tailList has an almost identical structure to it and will instead return the ListInt minus the first Integer.

Another example given in lectures for the ListInt datatype is the ability to sum all of the items in a ListInt. Its function definition is as follows:

```
sumList :: ListInt -> Int
sumList Nil = 0
sumList (Cons x ys) = x + sumList ys
```

Once again, this is just a function that will keep on going around, and around, and around until it reaches a Nil. Evaluation for this function is fairly similar to the one for tailList and headList.

```

sumList Cons 42 (Cons 6 Nil)
  42 + sumList (Cons 6 Nil)
    42 + 6 + sumList Nil
      48 + 0
        48

```

It is quite easy to see what is going on when everything is broken down, but if you try and do it all at once then I almost guarantee you will get confused. So break it down into its smallest components for the best results!

10 More fun (sic!) with lists

So, you know how to define recursive datatypes and do a few interesting things with them. You have seen how to get the first element in a list, the last element and summing a list. You may or may not have noticed but operating on lists always take a similar general form.

— *With the datatype:*

```

data ListInt = Nil | Cons Int ListInt
                deriving (Show, Eq, Ord)

```

— *You can say this:*

```

someFunction :: ListInt -> ...
someFunction Nil = ...
someFunction (Cons x ys) = ... someFunction x ys ...

```

All the sections with ... in are there to be filled in by you when defining this function. This general form works nicely, you can do anything with your recursively defined datatype and valid inputs from it. An example of something like this is checking if there is a specific element inside of a list.

— *Keeping the same ListInt definition as above*

```

elemList :: Int -> ListInt -> Bool
elemList x Nil = False
elemList x (Cons y ys) = (x==y) || elemList x ys

```

While at first this really does not look obvious, its not too bad. If the input list is empty then, of course, the function will return False. If not, then first of all the function will check if x is equal to y and return True if it is so and otherwise will call elemList once again with the same input (x) and the remains of the list (ys). The `||` operator is the logical OR operator, so if either the first or the second part of it is True then the whole thing will be True. The easiest way to understand this is by using once again step by step evaluation.

— *This example the input (6)*

— *is in the list (6, 42 and Nil)*

```

elemList 6 Cons 42 (Cons 6 Nil)

```

```

~(6==42) || elemList 6 (Cons 6 Nil)
~(False) || elemList 6 (Cons 6 Nil)
~(False) || ((6==6) || (elemList 6 Nil))
~(False) || ((True) || (elemList 6 Nil))
~(False) || (True)
~True

```

— *So 6 is in the list 6, 42 and Nil!*

It should be noted that once haskell has found a `True` in the `or` bit of code then it will evaluate the whole statement to `True`. This is another example of lazy evaluation in haskell and saves processing time (haskell does not need to compute the result of `elemList 6 Nil` in this case because no matter what the outcome, `((True) — (elemList 6 Nil))` will return `True`).

11 Trees

No, I am not making it up. This is a section about trees. A Tree structured datatype is great for...um...well actually I do not know of any examples it is good for other than something like a family tree, so here it goes.

Defining a Tree of Integers is done as below.

— *Definition for the TreeInt datatype*

```
data TreeInt = Leaf | Node TreeInt Int TreeInt
```

— *Example of the TreeInt datatype*

```
Node (Node (Node Leaf 1959 Leaf) 1986 (Node Leaf 1953 Leaf))
```

Once again you can see that this datatype uses recursion in its definition. The example of the `TreeNode` datatype represented graphically would have 1986 at the top with two nodes coming off it (1959 and 1953) which both would have leaves coming off them. Not too clear, but clearer when you draw it out for yourself!

A nice example of something neat you can do with Trees is finding its depth.

— *Definition of the depth function*

— *Uses the TreeInt datatype*

```
depth :: TreeInt -> Int
```

```
depth Leaf = 0
```

```
depth (Node t1 n t2) = 1 + max (depth t1) (depth t2)
```

You can also do other funky things like finding the sum of all the elements in a tree using a similar layout.

```
sumTree :: TreeInt -> Int
```

```
sumTree Leaf = 0
```

```
sumTree (Node t1 n t2) =
```

```
n + (sumTree t1) + (sumTree t2)
```

Once again, this is another function that uses recursion to achieve its goal! `sumTree` calls `sumTree` and so on until `sumTree` hits a `Leaf` when it returns 0 instead.

Lists can be a real pain to debug and hugs, as was noted earlier, does not allow you as a programmer to step through and see where things are going wrong.

12 Polymorphic Functions

While you may wish to define lots of different functions to do many things, there are some eventualities that require you to have functions that will do things in a more general sense. An example is summing up all elements in a list. The big problem with this example is that if you tried to feed it `Strings` then it would probably fall over fairly spectacularly.

If this datatype is defined:

```
data List a = Nil | Cons a (List a)
```

Then it is possible to do the following:

```
append :: List a -> List a -> List a
append Nil ys = ys
append (Cons x xs) ys = Cons x (append xs ys)
```

Because this is a very general definition for a (i.e a can of be any data type you wish) the function here is possible. Note that because both are of type `a` both lists must be of the same type, you could not add a list of `Bools` to a list of `Ints` here.

13 More About Lists

Note that the `ListInt` and `List` datatypes that have been defined earlier on in this set of pages can be used, although it is far easier and shorter to write out lists using the shorthand in square brackets. The first line of the old style first defines a list type using the algebraic data type, the second line is the list itself. The two other examples are both examples of the built in polymorphic list type; The first one is a list of `Strings` and the second a list of `Ints`.

— *Old Style*

```
data ListInt = Nil | Cons Int ListInt
Cons 1 Nil
```

— *Different Style*

```
[ "This is a" , "List" , "Of Strings" ]
```

— *Another Example*

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

When it comes to pattern matching on lists like this, it is fairly simple:

```
addListOfIntegers :: [Int] -> Int
addListOfIntegers [] = 0
addListOfIntegers [x:xs] = 1 + addListOfIntegers xs
```

Note that the Haskell prelude already has a function a little bit like this for finding the number of elements in a list simply called `length`:

```
Hugs.Base> :t length
length :: [a] -> Int
Hugs.Base> length ["hello", "world"]
2
```

14 Map and Filter

These are two higher order functions that operate on lists and are very useful indeed! They perform quite different tasks.

Map performs an action on a list. If we had a list of integers, we can add three to the whole list in one very simple step:

```
mapDemo :: [Int] -> [Int]
mapDemo [] = []
mapDemo x = map (+3) x
```

```
Main> mapDemo [1,2,4,6]
[4,5,7,9]
```

If you feed this function a list of Integers, it will spit out a list of integers with all the elements three higher than previously.

Filter is different in that it will return a list that has had a boolean operation performed on each element in the list and any False results would be removed from the list. This is useful if you wish to filter out invalid values from a list, play with sets of numbers and remove some that do not match a certain condition and so on. The example below shows a use of filter.

```
filterDemo :: [Int] -> [Int]
filterDemo [] = []
filterDemo x = filter (>4) x
```

```
Main> filterDemo [1,3,5,6,7]
[5,6,7]
```

15 Thanks To...

I would like to thank Olaf Chitil and Florence Benoy for allowing me to use some of their code from the lecture slides in this document and Thomas Davie for encouraging remarks and continued support regarding this document. Thanks also to Florence Benoy for proof-reading the pages.