Haskell: Higher Order Functions

Higher order functions can take other functions as parameters or return another function. There are three important higher order functions built into functional programming languages. **They all perform actions on lists**

- **Map**
  This applies a function to every item on the list and returns a list. It takes as its arguments (parameters) the function to be used and the list it is to be used on…

  ```
  Map (function) [list]
  ```

  ```
  >Map double [1,2,3]
  [2,4,6]
  ```

- **Filter**
  This applies a condition to every item on the list and returns a list of items that meet the condition

  ```
  Filter (function) [list]
  ```

  ```
  >Filter (>3) [1..5]
  [4,5]
  ```

- **Fold**
  This applies a 'combining function' to a list continually until the list is reduced to a single value.

  Folds are among the most useful and common functions in Haskell. They are an often-superior replacement for what in other language would be loops, but can do much more.

  There are two types of fold

  Foldr : Starts from the last item and works backwards
  Foldl: Starts from the first item and works forwards

TASK 2 – Functions and Higher Order Functions

Try this first.

Open the file **HaskellFunctions.hs**

Load this into the Haskell interpreter. Enter at prompt

```
Main> square 2

Main> sumUpList [1,2,3,4]
```

Open **HaskellFunctions.hs** in notepad and …

a) Write a function called **DoubleMe** which receives an integer and outputs an integer twice the value of the argument provided.

Answer

b) If you can, write a factorial function (remember a factorial will take in an **Int** and output an **Int**). It will use recursion and so needs a base case! Look at the other functions such as **sumuplist** to help.

Test your function works!

Answer

c) The **min** function takes two arguments and gives the lower. e.g.

```
>min 5 6
5
```

In one line of code write a command to find the minimum of the numbers 4 6 and 8. *Think of it as the minimum of a number with the minimum of two numbers.*

Answer

d) Write a **map** function and the **square** function to square every number in the list [1..10]

`So… [1,2,4,9,16,25,36,49,64,81,100]`

Answer

e) Use the **filter** function and the **isPrime** Function to find every prime number between 1 and 1000.

Answer

f) Try this, what do you get?

```
Main> foldr (\acc x -> acc + x) 0 [5,7,8,4]
```

Answer

g) Adjust this function to find the product of this list (i.e. multiply them all together rather than add them).

Answer

h) Adjust this fold function to count how many odd numbers there are in the list – use the **isOdd** function in the program which returns **1** – if odd, **0** - if even. Be careful you may need to consider the type of fold you need!

Answer

i) Write a fold function, using the **min** function we encountered before to find the smallest number in a list

Answer

j) Look at the 'interesting' function in the program. It seems to take a list and output a list.

a. What does it do?

Answer

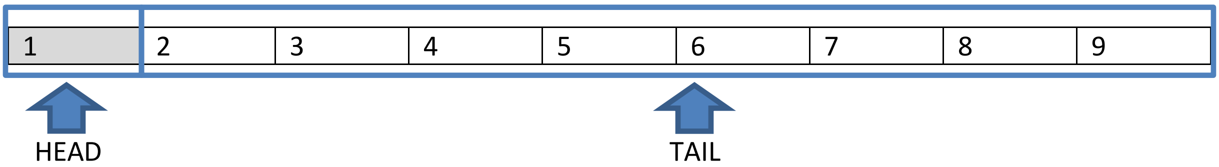
b. What specific algorithm is it performing? (One for the A level mathematicians!)

Answer

Lists in Functional Programming

Lists in functional languages are considered as a combination of **head** and **tail**

List



The **head** is an **element**

The **tail** is itself another list *(which in turn has its own head and tail)*

In Haskell lists are represented as : `[1,2,3,4,5,6,7,8,9]`

An empty list would look like this : `[]`

Typing in **`head [1,2,3,4,5,6,7,8,9]`** gives **1**

Typing in **`tail [1,2,3,4,5,6,7,8,9]`** gives **`[2,3,4,5,6,7,8,9]`**

When describing lists in Haskell we use the form **`X:XS`** e.g. `1: [2,3,4,5,6,7,8,9]`

Where **X** is the head

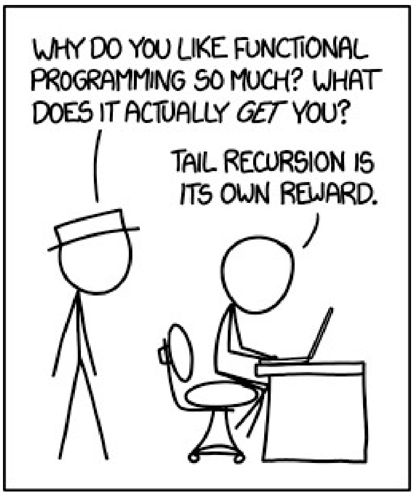
Where **XS** is the tail

We often process lists using **recursion**. We process the head then recursively process the tail.

```
sumUpList :: (Num a) => [a] -> a

sumUpList [] = 0     ← if it's an empty list return 0 (base case)

sumUpList (x:xs) = x + sumUpList xs  ← Add the head value to the sum of the tail.
```



© xkcd

HASKELL - WORKSHOP 3 Lists and processing lists

Haskell: Using Lists (some more functions!)

Basics

- *Define a list xs*

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

- *Get the size of the list.*

```
length xs                    (4)
```

- *Turn a list backwards.*

```
reverse xs                   ([4,3,2,1])
```

Finding / searching

```
head xs                      (1)
```
(returns the *first* element of the list.)
```
last xs                      (4)
```
(returns the *last* element of the list.)
```
tail xs                      ([2,3,4])
```
(returns all but first element)
```
init xs                      ([1,2,3])
```
(returns all but last element)

Adding

- *Add an element to the start of a list.*

```
new_element : xs             (5:s → [5,1,2,3,4])
```

- *Add an element to the end of a list.*

```
xs ++ [new_element]          (s++[5] → [1,2,3,4,5])
```

Empty lists

- *Define an empty list z*

```
Z=[]
```

- *Check if a list is empty.*

```
null xs                      (null xs → false)
                             (null z → true)
```

TASK3 – Using Lists

a) Enter the following expressions at the **prelude>** prompt

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

Using Just using **ONE LINE OF CODE** Enter a command to display the first item in the list

Answer

b) Enter the following expressions at the **prelude>** prompt

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

Using Just using **ONE LINE OF CODE** Enter a command to add together (using +) the first and last item in the list

Answer

c) Enter the following expressions at the **prelude>** prompt

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

Using Just using **ONE LINE OF CODE** make a new list that removes the head of this list and adds it to the end

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

Answer

d) Enter the following expressions at the **prelude>** prompt

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

Using Just **ONE LINE OF CODE (functional composition!)** and using only HEAD and TAIL display the number 3 in the list.

Answer

e) Enter the following expressions at the **`prelude>`** prompt

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

Using Just **ONE LINE OF CODE** add the last item of the list to the front of the list

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

Answer

f) Enter the following expressions at the **`prelude>`** prompt

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

t=[]
```

Using Just **ONE LINE OF CODE** make a list of the length of list **t** and length of list **s**

So [0,4]

Answer

Partial Function Application

This is the principle that a function can be called with an incomplete number of arguments.

In mathematical terms

The function **`Add(x,y)`** may take two integers and return an integer. However we now that it is actually considered as a function of a sub function with single arguments i.e.

`Add(x,y) → Addx (y)`

Therefore its function application scheme is (note: **right** associative)

`Add: integer→(integer→integer)`

`Add` **`Addx(y)`**

We can drop the brackets and so it becomes

`Add: integer→integer→integer`

Whoopy do.

So when declaring a function signature in Haskell we can do this…

```
powerOf :: Int -> Int -> Int

powerOf x y  = x^y
```

HASKELL - WORKSHOP 4 Partial application of functions

Haskell : Partial Application of functions

Looking at your **HaskellFunc.hs** program you will see a function called **powerOf**

It looks like this…

```
powerOf :: Int -> Int -> Int

powerOf x y  = x^y
```

You will see from the function signature that it takes two ***integer*** arguments **x** and **y** and outputs an ***integer*** result x^y.

However, functional languages such as Haskell only actually accept a single argument to any function so what is actually happening when you type…

```
powerOf 2 3 ??
```

Haskell is translating this into a series of functions each with one parameter (called currying)….

```
powerOf 2 3
    ➜ (PowerOf2) 3
              ➜  8
```

So what? I hear you sigh. Well, by partially applying functions. i.e. not passing all the parameters what you return is not the result ***but another function*** e.g.

```
powerOf 2
    ➜ PowerOf2 ?
```

So you can create new functions without having to code them!!

For example, if I want a function to square a number (which is x^2)

I could do it all from scratch…

```
Square :: int -> int

Square x = x * x
```

Or.. I could create a function by ***partially applying*** the PowerOf function.

```
Square = PowerOf 2
```

Because a partially applied function returns another function! In this case to the PowerOf 2.

I can now enter:

```
>Square 4

16
```

TASK 4 – Partial application of functions

Open the file **HaskellFunctions.hs**

Load this into the Haskell interpreter

a) Write a function called 'cube' that produces the cube of any number provided as an argument. Use partial application of the **PowerOf** function to do this. ***Note*** *: you can do this on the command line and don't need to do it by changing the file in notepad*

Answer

b) Test your cube function with the number 3. (the answer should be 27!)

Answer

c) Write a command to map your new function to the list [1,2,3,4] to produce the list [1,8,27,64]

Answer

We can use partial application on infix functions too by putting them inside brackets

For example, let's create a function called 'double' :

```
>double=(*2)
```

You can see we are missing one of the arguments in our multiply calculation so it will return a function that effectively doubles the missing argument (by multiplying it by two)

d) The infix function **++** adds two string arguments together

E.g.

```
>"Long" ++"fellow"

"Longfellow"
```

Create a function **`question`** using partial application that will stick a question mark at the end of any string argument provided.

e.g.

```
>question "Why me"

"Why me?"
```

Answer

e) Write a command to map your new function to the list `["Who","Why","How","When"]`

Answer

Functional Programming – Past Paper Question

In a functional programming language, a recursively defined function named `map` and a function named `double` are defined as follows:

```
map  f  []        =  []
map  f  (x:xs)    =  f  x  :  map  f  xs

double  x         =  2  *  x
```

The function `x` has two parameters, a function `f`, and a list that is either empty (indicated as `[]`), or non-empty, in which case it is expressed as (`x:xs`) in which `x` is the head and `xs` is the tail, which is itself a list.

(a) In **Table 1**, write the value(s) that are the head and tail of the list `[ 1, 2, 3, 4 ]`.

Table 1

Head	
Tail	

(b) The result of making the function call `double 3` is `6`.

(1)

Calculate the result of making the function call listed in **Table 2**.

Table 2

Function Call	**Result**
`map double [ 1, 2, 3, 4 ]`	

(1)

(c) Explain how you arrived at your answer to part (**b**) and the recursive steps that you followed.

....

....

....

....

....

....

(3)

(Total 5 marks)