June 2024

St Hilda's College
UNIVERSITY OF OXFORD

Computer Science

Congratulations on your offer to study Computer Science (and joint schools) at St Hilda's! I hope you are excited about coming to study with us in October.

I'm Matty Hoban, the tutor in computer science at St Hilda's. I am also an associate professor in the Department of Computer Science. You will see me in tutorials, but also possibly in lectures in the Department. I also will act as your personal tutor, which means we can discuss how things are going in a non-teaching context.

To help you prepare for your arrival, it is always good to have an idea of what to expect. I recommend you visit the following link to have a look at the structure of your degree:

https://www.cs.ox.ac.uk/ourstudents/degrees.html

There is also the following link, which has some useful background reading for your degree:

https://www.cs.ox.ac.uk/admissions/undergraduate/why_oxford/background_reading.html

In addition to the above, I have prepared an exercise sheet in functional programming. The degrees with a significant computer science component at Oxford are unusual in that the first programming language you learn is a purely functional programming language called Haskell. We do not assume that you have studied computer science before, nor have done any programming. Furthermore, if you have done some programming before, there is a good chance it was in a language that has an imperative style, where you can use loops for instance. Haskell does not have loops, and so regardless of your programming experience you might find learning it a challenge. The exercise "Getting Started with Functional Programming" should give you a taste of the kinds of problems you will encounter in your studies, and it should give you a few pointers to the important concepts in functional programming.

Finally, I want to wish you good luck with your results, and I look forward to meeting you in October.

Yours faithfully,

Dr Matty Hoban
Tutor in Computer Science

Getting Started with Functional Programming

St Hilda's College, University of Oxford

June 13, 2024

The first programming language you will encounter in your degree is Haskell. This is probably quite unusual compared with other computer science degrees. Haskell is a purely functional programming language, so all computations are modelled as functions, i.e. processes with only an input and an output. It's a bit like sitting an arithmetic test but you only produce the answer to basic sums, and you do not show your working. The following material should give you a taster of how to approach functional programming in Haskell.

Exercise 0

When you get to Oxford you will have access to the Bodleian library, and the relevant textbooks for the Functional Programming course. The two recommended textbooks are:

- Graham Hutton, *Programming in Haskell (2nd edition)*, Cambridge University Press, 2016.
- Richard Bird, *Introduction to Functional Programming using Haskell*, second edition, Prentice-Hall International, 1998.

For background reading, I recommend clicking on the following free online course:

- Miran Lipovača, *Learn You a Haskell for Great Good! A Beginner's Guide*, No Starch Press, 2011 – https://learnyouahaskell.com/chapters

It will first tell you how to install Haskell on your own machine. In particular, it will tell you how to install a Haskell interpreter called GCHI.

Another resource for downloading and learning the basics of Haskell is https://www.haskell.org/download/

Exercise 1

Functions are an essential part of functional programming (the clue is in the name). From the previous exercise, you should know the basics of writing functions in Haskell. Write functions that implement the following:

1. Given a number, compute the ceiling of half that number[1]
2. Given a number, compute the largest power of 2 that is smaller or equal to that number, e.g. the answer will be 128 for 150
3. Given numbers n and x, write a function that computes the largest power of x that is smaller or equal to n

In your functions above do not use additional libraries.

[1] The ceiling of a number x is the smallest integer larger than or equal to x.

Exercise 2

In Haskell, we have basic types such as integers and characters denoted as `Int` and `Char` respectively. Variables of a particular type can take values within these types.

In functional programming, functions are first class citizens. This is an elaborate way of saying that functions can be treated similarly to other basic types. To represent functions we use arrow notation `->` to denote input to output. For instance, a function called `even` takes an integer and outputs `True` if it is even, and `False` otherwise, where `True` and `False` are values of the Boolean type, `Bool`. The type of `even` is denoted as `even :: Int -> Bool`.

For all of the functions in Exercise 1, write out their types. Can you make other choices of types for these functions? If not, why not, or if so, what could they be?

Exercise 3

Recursion is another fundamental aspect of functional programming. Recursive functions are functions that call themselves. Consider the following recursive function, which is supposed to compute the factorial of an integer:

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

What is wrong with this function? Fix the function. Consider the following recursive function, which for a given non-negative integer n, is supposed to compute the sum of all integers from 0 to n (inclusive):

```
nats 0 = 0
nats n = if n > 0 then n + nat(n-1) else False
```

What is wrong with this function? Fix the function.

Prove (mathematically) that your corrected functions `fact` and `nats` are indeed correct.

For positive integers, the *harmonic numbers*, denoted H_n are defined as

$$H_n = \sum_{k=1}^{n} \frac{1}{k}.$$

Write a function that computes the nth harmonic number H_n, given positive integer n.

Exercise 4

For storing multiple values of a particular data type, we can use lists. You should have read that lists of a data type `a` will have the type `[a]`. Lists can be built up from single elements using the `:` operator, e.g. `(x:xs)` prepends the value `x` to the list `xs`. If you are unfamiliar with this or the concept of lists, you might want to do more background reading.

Imagine the elements of a list are indexed by integers from 0 to the length of the list minus 1: the first element in the list is indexed by 0, the second element is indexed by 1, and so on. The following function is supposed to take a list as input and produces a list of the even-indexed elements:

```
evens :: [Int] -> [Int]
evens x:y:xs = x : (evens xs)
```

What is wrong with this function? Fix the function so that it outputs the correct list for all possible lists.

Haskell uses lazy evaluation, which allows it to represent infinitely long lists. For instance, the list `[1..]` is the list of all integers from 1 onwards. We can write Haskell code to print `evens [1..]` to the screen, but since this is again an infinitely long list, it cannot be printed in finite time. But we

can apply functions like the following to infinite lists to produce a finite list:

```
elements ::  Int -> [Int] -> [Int]
elements 0 _ = []
elements n [] = []
elements n (x:xs) = x:  take (n-1) xs
```

This will return a list of the first n elements of a list when given n and the list, e.g. `elements 5 [1..]` will produce the list `[1,2,3,4,5]`.

1. Write a function called `sums` that takes a list of integers and adds up all the integers in the list
2. Write a function called `products` that takes a list of integers and multiplies all the integers in the list
3. Write new versions of `fact` and `nats` from Exercise 3 that use the functions `sums`, `products` and `elements`.

Author: Matty Hoban