Variable Scope: Imperative vs Functional Programming

# Note

When I am referring to objects in Haskell as immutable, I am referring to the general use case of the language as the scope of mutable objects in Haskell is narrow. Mutable objects do exist in Haskell but are atypical, as the paradigm asks for immutable objects.

# Introduction

I’ll start with a brief example of variable re-assignment in the C language

int a = 5;

a = 3;

For those familiar with C (which should be all of us), this code is simple. We declare a variable **a** and initialize the value to **5**. On the next line, the variable a is then reassigned to the value **3**. We have changed the state of the variable **a**.

# Behind the scenes

So, what exactly is going on when we assign a variable in C?

* We declare our variable, **a**
* A memory location is chosen for **a** and the 32 (or 64)-bit binary equivalent of the value **5** is stored there.
* Now we can talk about the variable **a**
* We reassign the value of **a**
* a keeps the same memory location, but the value stored at the memory location is now the binary-equivalent of the value of **3**

So as you can probably tell just by the fact that this blog post exits, Haskell does not handle variables like this. So...

This is imperative programming. A more formal description for imperative programming is as follows:

*“A programming paradigm that uses statements that change a program's state”*

Contra a formal description of functional programming:

*“A programming paradigm that treats computation as the evaluation of mathematical functions and avoids changing-state and mutable data”*

## What are these saying?

The major difference between imperative programming and functional programming is a matter of dealing with state. In imperative programming, objects are mutable. This means that you can set the value of some object, in this case **a**, to some value and later change the value of that variable. In functional programming in Haskell objects are immutable.

### Example

Constants in C are immutable (defined with #define) as they are evaluated at compile time. They cannot change.

# How do Haskell variables work?

To better understand Haskell, we must recall how variables are assigned in recursion

Consider the following C pow function:

[programming paradigm](https://en.wikipedia.org/wiki/Programming_paradigm) that uses [statements](https://en.wikipedia.org/wiki/Statement_(computer_science)) that change a program's [state](https://en.wikipedia.org/wiki/State_(computer_science))

pow :: Integer -> Integer -> Integer

// pretend I handle negative powers here

pow \_ 0 = 1

pow x y = x \* (pow x (y - 1))

Tracing through the code, we can see that for input **x** and **y**, we recursively call power and decrement **y** until it reaches **0**, then our recursion works its way back up the stack. This should be a pretty natural expectation at this point. What is different about how we assign x in this case is that because we are using recursion, we make a new variable reference for x every time a new function is called, and that reference is associated with the enclosing scope.

While this may not seem immediately interesting, in the grand scheme of things, this is what allows us to and as an added bonus, write our code so concisely.

## References

<https://en.wikibooks.org/wiki/Haskell/Variables_and_functions#Variables_in_imperative_languages>