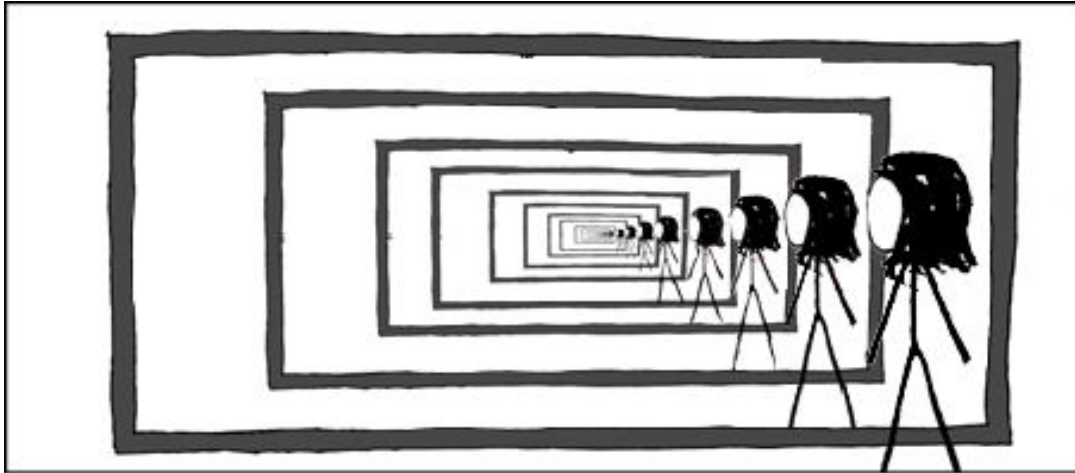
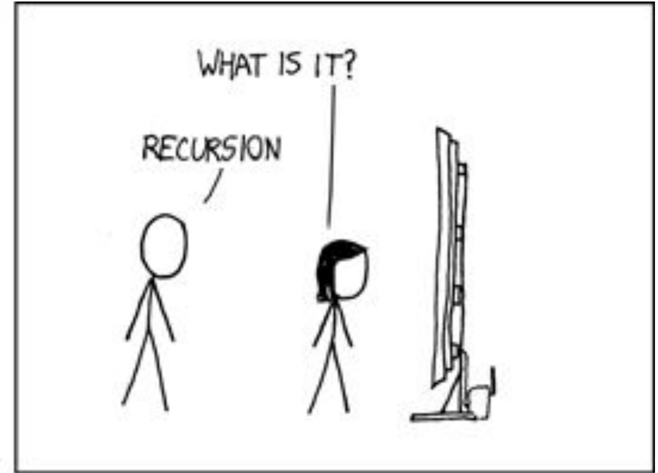
3: Pattern matching, Recursion, and the ALU





Recap and Agenda

Last week we looked at

- Software:
  - Haskell functions, types, names and expression
- Hardware:
  - How memory is created using integrated circuits

Today, we'll:

- Software:
  - Look more at Haskell functions, pattern match on them and look at recursion
- Hardware:
  - Create an arithmetic unit using integrated circuit (our last lesson using breadboards!)

To start let’s code together

Go to repl.it and search for Haskell

We’re going to code today’s examples together

Starting with Haskell

Remember, *everything in haskell has a type*, which describes the data.

Type signatures tell you what type something is. They look like this:

`age :: Int`. This reads: age has type `int`.

You can use `:t` in GHCi (the black screen) to get the type signature of *anything*

Functions have type signatures too. These signatures have `->` to represent input and output.

Let's look at the type signature of take using `:t take`

In these type signatures, the **last type is the output**, and everything before it is **input.**

`:t` all the things!

Let's look at the type of `head`, a function that gives back the first element of a list.

```
head :: [a] -> a
```

We notice something strange about its signature, namely what is `[a]`??

That is a **polymorphic type** meaning `a` can polymorphise into any type it so pleases. `a` can be of type `Char`, `Int`, another list, and so on.

More on type signatures and functions

It’s great to know the input(s) to a function, but how does the computer *know* what to do with these inputs to make an output?

Pattern matching!

According to http://learnyouahaskell.com: “Pattern matching consists of **specifying patterns** to which some data **should conform** (and then checking to see if it does) and **deconstructing** the data **according to those patterns**.”

In other words, pattern matching is *looking for certain inputs*, and *applying* certain changes to those inputs to make them outputs (if it’s possible to do so).

Let’s recreate `head`

To understand pattern matching, we’ll write the pattern matching for head.

head takes the first index of a list and returns it to you. To do this, we’ll have to pull the first index out of the list and return it.

On lists: we know lists need one thing, the `[]`. Data is added to a `[]` using `:`, the **cons** operator. As such these two lists represent the same:

```
[1, 2, 3] = 1:2:3:[]
```

If we wanted to pull out the first index from this list of characters we’d have to *assign a variable to the first index* (and a variable for everything that comes after) and *return the first index*. Let’s do it!

Now you try to create `tail’`

Which is a function that takes in a list and returns everything **but** the head!

Typeclasses

Let’s continue our exploration by examining the type of `elem` which returns the data at a certain index in a list.

```
:t elem
```

What’s new here?

Everything before `=>` is called a class constraint. When we write `Eq a => …`, we can use any type, *provided it belongs to the `Eq` typeclass.* This typeclass let’s us *check whether two values are equal* using `==` and `/=`

This is how elem works, it goes through each index of a list, checks if each value is the same as the value you want to see is inside the list, and tells you with a Bool (true or false) if its there.

Recursion

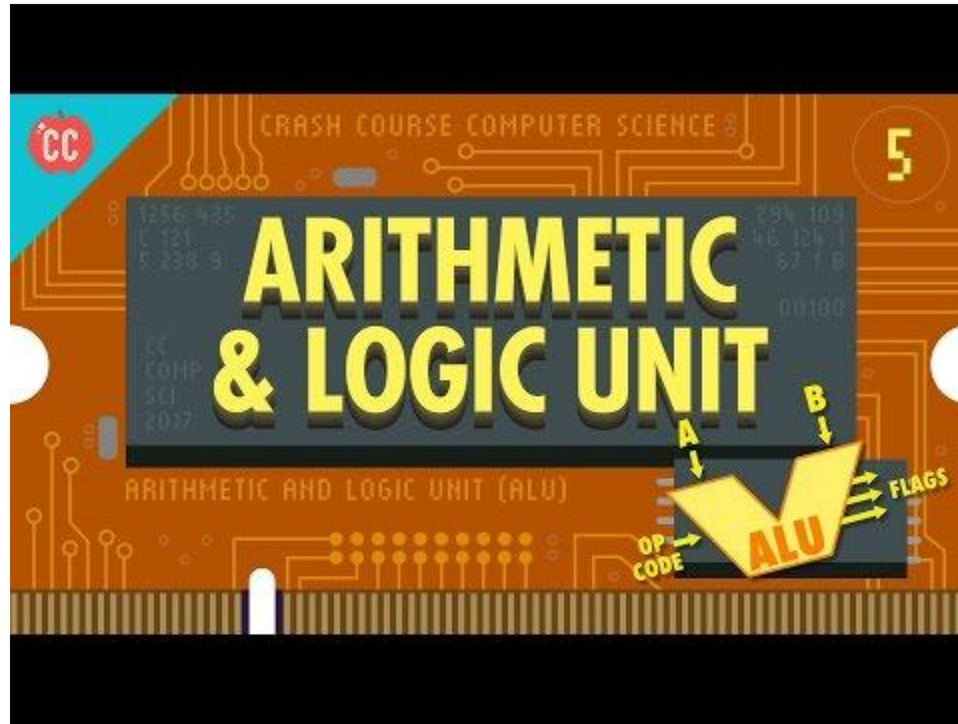
Challenger approaches!

Try implementing 3 of the list processing functions from this list:

building21.ca/list-examples

If you feel comfortable with your solution, come present it to the class!

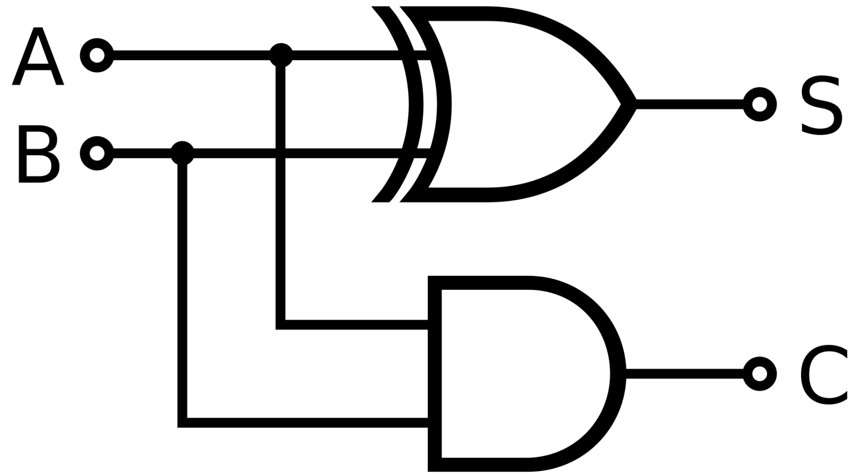
Hardware: the ALU



Constructing our own half adder

Our last activity with the breadboard will be creating our own half adder!

Remember: the smiling spaceship is the XOR gate!



Recap and agenda

This week, we talked about:

- Pattern matching
- Recursion
- The ALU.

Next week we're going to talk about:

- What happens when you turn on a computer?
- Operating systems
- Your first introduction to Linux, woohoo!!