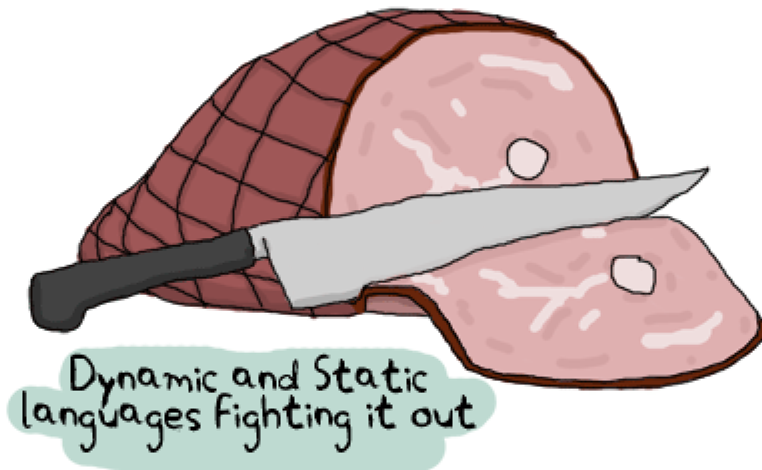
DYNAMITE-STRONG TYPING

As you might have noticed when typing in examples from Starting Out (for real), and then modules and functions from Modules and Syntax in Functions, we never needed to write the type of a variable or the type of a function. When pattern matching, the code we had written didn't have to know what it would be matched against. The tuple `{X,Y}` could be matched with `{atom, 123}` as well as `{"A string", <<"binary stuff!">>}`, `{2.0, ["strings","and",atoms]}` or really anything at all.

When it didn't work, an error was thrown in your face, but only once you ran the code. This is because Erlang is *dynamically typed*: every error is caught at runtime and the compiler won't always yell at you when compiling modules where things may result in failure, like in Starting Out (for real)'s `"llama + 5"` example.



One classic friction point between proponents of static and dynamic typing has to do with the safety of the software being written. A frequently suggested idea is that good static type systems with compilers enforcing them with fervor will catch most errors waiting to happen before you can even execute the code. As such, statically typed languages are to be seen as safer than their dynamic counterparts. While this might be true when comparing with many dynamic languages, Erlang begs to differ and certainly has a track record to prove it. The best example is the often reported *nine nines* (99.9999999%) of availability offered on the Ericsson AXD 301 ATM switches, consisting of over 1 million lines of Erlang code. Please note that this is not an indication that none of the components in an Erlang-based system failed, but that a general switch system was available 99.9999999% of the time,

planned outages included. This is partially because Erlang is built on the notion that a failure in one of the components should not affect the whole system. Errors coming from the programmer, hardware failures or [some] network failures are accounted for: the language includes features which will allow you to distribute a program over to different nodes, handle unexpected errors, and *never* stop running.

To make it short, while most languages and type systems aim to make a program error-free, Erlang uses a strategy where it is assumed that errors will happen anyway and makes sure to cover these cases: Erlang's dynamic type system is not a barrier to reliability and safety of programs. This sounds like a lot of prophetic talking, but you'll see how it's done in the later chapters.

Note: Dynamic typing was historically chosen for simple reasons; those who implemented Erlang at first mostly came from dynamically typed languages, and as such, having Erlang dynamic was the most natural option to them.

Erlang is also strongly typed. A weakly typed language would do implicit type conversions between terms. If Erlang were to be weakly typed we could possibly do the operation `6 = 5 + "1".` while in practice, an exception for bad arguments will be thrown:

```
1> 6 + "1".
** exception error: bad argument in an arithmetic
expression
in operator  +/2
called as 6 + "1"
```

Of course, there are times when you could want to convert one kind of data to another one: changing regular strings into bit strings to store them or an integer to a floating point number. The Erlang standard library provides a number of functions to do it.

Source : http://learnyousomeerlang.com/types-or-lack-thereof