Type systems
Expressing constraints
type: a set of values, and (implicitly) a set of behaviours on those values
**type system**: the rules governing the use of types in a program, and how types affect the program semantics
dynamic typing: types are checked during the execution of a program
static typing: types are checked before the execution of a program (during “compilation”)


From fighting the \texttt{@(*#&$ compiler...}
...to having a conversation with it.
A brief introduction to Haskell’s type system
Inspecting types: :type

Built-in types: Int Bool [Char] ...
Note about function types and currying
Declaring types

data `<type-name> = <type-expr>`
Struct-like types (product types)

data Point = P Int Int

Point type name
P data constructor
Enum-like types (sum types)

```haskell
data Day = Mon | Tue | Wed
         | Thu | Fri | Sat
         | Sun
```
**algebraic data type**: a type formed by any combination of sum and product types
algebraic data types vs. inheritance
- no inheritance of methods
- no inheritance of attributes
- closed
data IntList = Empty
              | Cons Int IntList

data StringList = Empty
                 | Cons String String String List

data BoolList = Empty
                | Cons Bool BoolList
Polymorphism

Greek: "poly" (many) and "morphe" (form)
generic (or parametric) polymorphism

the ability for an entity to behave identically regardless of “input” or “contained” type
Haskell lists are generically polymorphic

data List a = Empty |
              Cons a (List a)
Haskell lists are generically polymorphic (built-in syntax)

```haskell
data [] a = [] |
            (:) a ([][] a)
```
data [] a = []
  | (:) a ([] a)

a is a type parameter
[] is a type constructor
("function" from types to types)
length :: [a] -> Int
length [] = 0
length (x:xs) = 1 + length xs
Deriving constraints from function genericity

\[ f :: a \to a \]
\[ f \; x = \ldots \]
f :: a -> [a]

f x = ...
\( f :: a \rightarrow b \)

\( f \ x = \ldots \)