Type Systems
A programming language is a form of communication (with the computer, among others).

The code we write expresses what/how we want the computer to compute.
Types are a way to express 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
(+ 1 "hi")
(or #t (+ 1 "hi"))
True || (1 + "hi")
dynamic typing

types are checked during the evaluation of a program
static typing

types are checked before the evaluation of a program (i.e., an operation on the abstract syntax tree)
From fighting the @(*#&$ compiler...
...to having a conversation with it.
A brief introduction to Haskell’s type system
Inspecting types: :type

Built-in types: Int  Bool  [Char]/String
Function types and automatic currying

\((\&\&) :: \text{Bool} \to (\text{Bool} \to \text{Bool})\)

\(\text{Bool} \to (\text{Int} + \to (\text{String} \to \text{Bool}))\)
Declaring types

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

data Point = P Int Int

\[\text{type} \quad \text{value} \quad \text{name} \quad \text{constructor}\]
Enum-like types (**sum types**)

data Day = Mo | Tu | We | Th | Fr | Sa | Su
Declaring types

data <type-name> = <type-expr>
**algebraic data type**

a type formed by any combination of sum and product types

data Shape = Circle Point Int
            | Rect Point Point
algebraic data types vs. inheritance

- no inheritance of methods
- no inheritance of attributes
- closed (can’t add new constructors)
data IntList = Empty
             | Cons Int IntList

data StringList = Empty
                 | Cons String StringList

data BoolList = Empty
               | Cons Bool BoolList
Polymorphism

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

the ability for an entity to behave in the same way regardless of “input” or “contained” type
Haskell lists are generically polymorphic (abstract version)

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

data [] a = []
   | (:) a ([] a)
Haskell lists are generically polymorphic (built-in version)

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

a is a type variable/parameter
[] is a type constructor ("function" from types to types)
\textbf{length} :: [a] \rightarrow \text{Int} \\
\text{length} \ [\ ] = 0 \\
\text{length} \ (x:xs) = 1 + \text{length} \ xs
The empty list `[]` is a generically polymorphic value.

During type-checking, the compiler determines how to instantiate the type variable so that the expression type-checks.
> :type []
[] :: [t]

> :type undefined
undefined :: a

> :type error
error :: [Char] -> a
Deriving constraints from generic polymorphism

If a function is generically polymorphic, there are many things it *can’t* do.
\( f :: a \to a \)

\( f \ x = \times \)
\[ f :: a \rightarrow [a] \]

\[ f \; \mathbf{x} = \begin{cases} \mathbf{x} \times \mathbf{x} \times \mathbf{x} \times \ldots \times \mathbf{x} & \text{if } \mathbf{x} \text{ is } \mathbf{a} \\ \mathbf{x} \times \mathbf{x} \times \mathbf{x} \times \ldots \times \mathbf{x} & \text{if } \mathbf{x} \text{ is } \mathbf{b} \\ \mathbf{x} \times \mathbf{x} \times \mathbf{x} \times \ldots \times \mathbf{x} & \text{if } \mathbf{x} \text{ is } \mathbf{c} \end{cases} \]
\[ f :: [a] \rightarrow [a] \]

\[ f \ x = \text{remove duplications} \bigg\} \text{index-based permutations} \]
\[ f :: (a \to b) \to a \to b \]

\[ f \, x \, y = x \cdot y \]
\( f :: a \to b \)

\( f \ x = \text{Undefined} \)
Deriving constraints from function genericity

\[
\langle T \rangle \ T \ f(T \ x) \ \{ \\
\quad \cdots \\
\}
\]
(not) Deriving constraints from function genericity

```cpp
<T> T f(T x) {
    return x;
}
```
Deriving constraints from function genericity

\[
\langle T \rangle \ T \ f(T \ x) \ \{ \\
\quad \text{blowUpUofT}(); \\
\quad \text{return} \ x; \\
\}
\]
One last example ("theorem for free")

Given...

\[
\text{map} :: (a \rightarrow b) \rightarrow [a] \rightarrow [b]
\]

\[
f :: a \rightarrow b \quad \text{(any function of this type)}
\]

\[
xs :: [a] \quad \text{(any list of this type)}
\]

\[
r :: [c] \rightarrow [c] \quad \text{(any function of this type)}
\]

\[
r \ (\text{map} \ f \ xs) == \text{map} \ f \ (r \ xs)
\]
But what about (+)?

(+) :: Int -> Int -> Int

(+):= a -> a -> a