Types

We have not written down any type information for these functions. This is permitted in Haskell because the compiler can generally work out the types without help. It is good practice, however, to document the types we expect the functions to have. In Haskell the syntax for this is to place a type declaration (anywhere) before the equation that defines the function. Here are some possible types for functions which we wrote earlier today:

\[
\text{takesome} :: \text{Int} \rightarrow \text{[Char]} \rightarrow \text{String} \\
\text{qsort} :: \text{[Int]} \rightarrow \text{[Int]} \\
\text{transpose} :: \text{[[Int]]} \rightarrow \text{[[Int]]}
\]

Type inference

When a type is provided the compiler checks to see that it is consistent with the equations for the function. The following function is valid, but the compiler will reject it because the type the programmer has specified does not match what the compiler deduces:

\[
f :: \text{[Char]} \rightarrow \text{Int} \\
f \ x = (\text{length } x) > 0
\]

More about types

What is the type of this function?

\[
\text{lengthof} \ [ ] = 0 \\
\text{lengthof} \ (x:xs) = 1 + \text{lengthof} \ xs
\]

Could it be:

\[
\text{lengthof} :: \text{[Integer]} \rightarrow \text{Integer}
\]

What about

\[
> \text{lengthof} \ "abcde" \ 5
\]

Which would imply a type:

\[
\text{lengthof} :: \text{[Char]} \rightarrow \text{Integer}
\]

We could make an arbitrary decision...
### Parametric polymorphism

In Haskell we are allowed to give that function a general type:

```haskell
lengthOf :: [a] -> Integer
```

This type states that the function `lengthOf` takes a list of values and returns an integer. There is no constraint on the kind of values that must be contained in the list.

What about this:

```haskell
head (x:xs) = x
```

This takes a list of values, and returns one of them. There is no constraint on the types of things that can be in the list, but the kind of thing that is returned must be the same type:

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

We will explore this parametric polymorphism in more detail later.

### Defining new types

Haskell makes it easy to define new data types. This helps to model data within the program.

```haskell
data Day = Monday | Tuesday | Wednesday | Thursday |
          | Friday  | Saturday | Sunday
```

We can define operations on values of this type by the pattern matching:

```haskell
weekend :: Day -> Bool
weekend Saturday = True
weekend Sunday = True
weekend _ = False
```

The “tags” on these types, and the types themselves, must begin with uppercase letters (functions and parameters in Haskell begin with lowercase letters).

### Recursive data types

Haskell also allows data types to be defined recursively.

We are familiar by now with lists in Haskell; writing the list `[1,2,3]` is just a shorthand for writing `1:2:3:[]`.

If lists were not built in we could define them

```haskell
data ListOfNumbers = EmptyList |
                    ConsNode Int ListOfNumbers
```

Using this definition the list would be written:

```haskell
ConsNode 1 (ConsNode 2 (ConsNode 3 EmptyList))
```

Recursive types usually mean recursive functions:

```haskell
lengthOfList :: ListOfNumbers -> Integer
lengthOfList EmptyList = 0
lengthOfList (ConsNode _ rest) = 1+(lengthOfList rest)
```

### Parameterised data types

Of course, those lists are not as flexible as the built-in lists, because they are not polymorphic. We can fix that:

```haskell
data List a = EmptyList |
            ConsNode a (List a)
```

No change to the length function, but the type becomes:

```haskell
lengthOfList :: (ListOfAnything a) -> Integer
```
Multiply-parameterised data types

Here is a useful data type:

```haskell
data Pair a b = Pair a b
```

```haskell
divmod :: Integer -> Integer -> (Pair Integer Integer)
divmod x y = Pair (x / y) (x `mod' y)
```

Actually, like lists, “tuples” (of various sizes) are built in to Haskell and have a convenient syntax:

```haskell
divmod :: Integer -> Integer -> (Integer,Integer)
divmod x y = (x / y, x `mod' y)
```

As you would expect, we can pattern match on them:

```haskell
f (x,y,z) = x + y + z
```

Another example: failure

A type that is often used in Haskell is one to model failure. While we can write functions such as `head` so that they fail outright:

```haskell
head (x:xs) = x
```

It is sometimes useful to model failure in a more manageable way:

```haskell
data Maybe a = Just a
             | Nothing
```

Every `Maybe` value represents either a success or failure:

```haskell
mhead :: [a] -> Maybe a
mhead [] = Nothing
mhead (x:xs) = Just x
```

This technique is so common that `Maybe` and some useful functions are included in the standard library.

Constraints

Take this function:

```haskell
addpair (x,y) = x*y
```

Which of the following would be the correct type:

- `addpair :: (Integer,Integer) -> Integer`
- `addpair :: (Float,Float) -> Float`
- `addpair :: ([Bool],[Bool]) -> [Bool]`

Is there a type that covers the first two but not the third? This is too generous:

```haskell
addpair :: (a,a) -> a
```

In order to narrow the acceptable values to things that can be added we can write this:

```haskell
addpair :: Num a => (a,a) -> a
```

The declaration `Num a => (a,a) -> a` is what is known as a class constraint. A number of predefined classes may be mentioned: `Num` (numeric), `Ord` (ordered values), `Eq` (things that have equality tests, but maybe not ordering), `Show` (things that can be converted to strings), and more (you can define new classes easily). The mention of the class name is a promise that some set of functions will work on the values of that class.
Being classy

```haskell
data Day = Monday | Tuesday | ... Sunday

> Monday
ERROR - Cannot find "show" function for:
*** Expression : Monday
*** Of type : Day
```

New types do not automatically belong to any classes. We can get a default class instance for certain built-in classes by expanding the type definition:

```haskell
data Day = Monday | Tuesday | ... Sunday
deriving (Show, Eq)
```

Custom class instances

We can provide a custom class instance by writing an instance declaration:

```haskell
data Day = Monday | Tuesday | ... Sunday

instance Show Day where
  show Monday = "Maandag"
  show Tuesday = "Dinsdag"
  ...
  show Sunday = "Zondag"
```

Show is easy because there is only one function. We will return to other type classes later.

An example

Take a program that has to read arithmetic expressions and calculate their value. Rather than representing 1+5 as a string (which might require us to consider what "1+5"++"worrisome" means) we can define a new type:

```haskell
data Expression = Val Float
  | Add Expression Expression
  | Multiply Expression Expression
  | Subtract Expression Expression
  | Divide Expression Expression

deriving Show
```

An evaluator

We can write a function to calculate the result of these expressions:

```haskell
eval :: Expression -> Float
eval (Val x) = x
eval (Add x y) = eval x + eval y
eval (Multiply x y) = eval x * eval y
eval (Subtract x y) = eval x - eval y
eval (Divide x y) = eval x / eval y
```

```haskell
> eval (Add (Val 10) (Multiply (Val 5) (Val 90)))
460.0
```