Standard Haskell Types

In this section we learn about the standard values and types that Haskell offers. These form the bread and butter of working with Haskell.

Haskell Types

Haskell is what is known as a **statically typed** language: Every value and expression in the language has a *type*. A **type** is in effect *a collection of related values*. A key premise is that whenever an operation can be performed for a value of a certain type, it can also be performed for all values of that same type.

Types effectively allow us to organize our program. Knowing that a function returns a value of a certain type allows us to safely feed that value as input into another function that accepts that type of value.

Types in Haskell always start with a capital letter.

Type ascription

We can specify the type of an expression by appending a double colon :: followed by the type. For example we can write:

```
True :: Bool          -- The value "True" is of type Bool
not :: Bool -> Bool   -- The value "not" has the function type Bool->Bool
not True :: Bool      -- The expression "not True" is of type Bool
```

You can ask the Haskell interpreter for the type of an expression by prepending it with :type:

```
:type not True    -- will print "not True :: Bool"
```

An important point to make is that the above does not actually attempt to evaluate the expression `not True`, it just determines its type.

Type inference

Haskell has a very powerful *type inference* process. **Type inference** is the process of determining the type of an expression without any required input from the programmer.

This means that most of the time we do not need to specify the types of expressions, we can let Haskell figure it all out. It is however customary to specify the types of functions in scripts, as we will see later on.

The type inference model is based on a simple idea:

> In the expression `f x` where `f` is a function of type `A -> B`, then `x` must have type `A` and the result `f x` has type `B`.

This simple principle introduces type constraints, and the Haskell typechecker solves these constraints to determine the type of more complex expressions.

Basic Types

Here is a listing of the basic types in Haskell

`Bool` for logical values. The only values are `True` and `False`.

`Char` for single Unicode characters, surrounded in quotes like so: `'a'`.

`String` for strings of characters. Surrounded in double-quotes like so: `"hello there!"`. Note that technically strings are the same thing as lists of chars. So `String` is actually a *type alias* for what we will call `[Char]` in a bit.

`Int` for fixed-precision integers, extending up to $\pm 2^{63}$.

`Integer` for arbitrary precision integers. These can be as large as desired, and will cause no overflow errors. But operations on them are a lot slower.

`Float` for single-precision floating point numbers. In general these are to be avoided in favor of `Double`. These typically contain no more than a total of 8 digits.

`Double` for double-precision floating point numbers. These typically contain 16 digits.

Practice

- Start your session with `ghci Chapter3` then add your code to the `Chapter3.hs` file.
- The book provides a definition for an "exclusive or" function on page 42. Implement it in a file and load it, then determine the full truth table for it:

```
exOr True True
exOr True False
exOr False True
exOr False False
```

- Exercises 3.1, 3.3 (name your functions exOr1 and exOr2)
- To test that the implementations of `exOr` and `exOr1` are in fact behaving the same way, we can write a "quickCheck" property, and add it to the same file:

```
prop_exOr1_equals_exOr :: Bool -> Bool -> Bool
prop_exOr1_equals_exOr x y =
    exOr1 x y == exOr x y
-- Do the same for exOr2
```

  To execute these properties, run:

```
quickCheck prop_exOr1_equals_exOr
quickCheck prop_exOr2_equals_exOr
```

- Add a property that should fail: `prop_exOr_equals_or x y = exOr x y == (x || y)` and run a quickCheck on it, and it should give you an example of values that make it fail.
- Exercises 3.5-3.7