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

Reading

- Sections 3.1-3.6
- Practice exercises (3.11): 1, 2, 3 (might not be able to do some parts until next class)

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

> for logical values. The only values are `True` and `False`.
>
> for single Unicode characters, surrounded in quotes like so: `'a'`.
>
> 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.
>
> for fixed-precision integers, extending up to $\pm 2^{63}$.
>
> 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.
>
> 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.
>
> for double-precision floating point numbers. These typically contain 16 digits.

Compound Types

There are a number of ways of producing more complex types out of simpler types. These are some times called *compound types*.

List Types The first example of that is the list type. As elements of the list all must have the same type, we can specify the type of a list with two pieces of information:

- The fact that it is a list. This is denoted by using a single pair of square brackets: `[...]`
- The fact that the entries have a certain type. That type goes between the brackets.

```
[False, True, True, True] :: [Bool]
['a', 'b', 'c'] :: [Char]              -- Can also be called String
"abc" :: [Char]                        -- Same as above
["abc", "def"] :: [[Char]]             -- Or also [String]
```

Practice: Write a value of type `[[Int]]`.

Tuple Types A **tuple** is a collection of values separated by commas and surrounded by parentheses. Unlike lists:

- A tuple has a fixed number of elements (fixed *arity*), either zero or at least two.
- The elements can have different types, from each other.
- The types of each of the elements collectively form the type of the tuple.

Examples:

```
(False, 3) :: (Bool, Int)
(3, False) :: (Int, Bool)     -- This is different from the one above
(True, True, "dat")  :: (Bool, Bool, [Char])
() :: ()                      -- The empty tuple, with the empty tuple type
```

We can also mix list types and tuple types. For instance:

```
[(1, 2), (0, 2), (3, 4)] :: [(Int, Int)]       -- A list of pairs of integers
 -- A list of pairs of strings and booleans
[("Peter", True), ("Jane", False)] :: [([Char], Bool)]
```

Activity: Think of uses for tuple types. What information might we choose to represent with tuples?

Function types A function type is written as `A -> B` where `A` is the type of the input and `B` is the type of the output. For example:

```
add3 x = x + 3            :: Int -> Int
add (x, y) = x + y        :: (Int, Int) -> Int
oneUpTo n = [1..n]        :: Int -> [Int]
range (a, b) = [a..b]     :: (Int, Int) -> [Int]
```

When writing functions, we tend to declare their type right before their definition, like so:

```
range :: (Int, Int) -> [Int]
range (a, b) = [a..b]
```

You may be tempted to think of this function as a function of two variables. It technically is not, and we will discuss this topic on the next section.

Type Practice

Work out the types of the following expressions:

1. (5 > 3, 3 + head [1, 2, 3])
2. [length "abc"]
3. The function f defined by f lst = length lst + head lst
4. The function g defined by g lst = if head lst then 5 else 3

Curried functions

Looking at the example of the range function above:

```
range :: (Int, Int) -> [Int]
range (a, b) = [a..b]
```

You may be tempted to think of this function as having as input *two parameters*, the a and the b. In reality it has only *one parameter*, namely the *tuple* (a, b). This is why the type for the function has one thing on the left side of the arrow, namely the compound type (Int, Int).

This is an important step: Compound types allow us the illusion of multiple parameters when in reality there is only one parameter.

There is however one other way of allowing multiple parameters, which is called *currying* in honor of Haskell Brooks Curry once again. The main idea is that functions can be specified to take *multiple parameters one at a time*. An example is in order, using the function take we saw earlier. A typical call to take would look like this:

```
take 3 [1..10]
```

So we are calling take, providing it with two parameters, and get back the result list.

However, the "curried" nature of the function lies in the fact that we could provide only the first argument, and thus create a new function that simply expects a list as input:

```
prefix = take 3          -- prefix is now a function
prefix [1..10]           -- This is the same as `take 3 [1..10]`
```

Providing only partial arguments to a curried function, and thus effectively creating a new function, is an extremely common practice, and the system is built so that this process is very efficient.

Let us look at another example:

```
f x y = x + y            -- function of two variables
add3 = f 3               -- new function
add3 10                  -- same as f 3 10
```

Types for carried functions A curried function is basically *a function whose return value is again a function*. When we write `f x y = x + y` what Haskell reads is:

> `f` is a function of one argument `x`, whose result is a new function of one argument `y`, whose result is adding the `x` to the `y`.

This helps us understand the type of such a function:

```
f :: Int -> (Int -> Int)
```

Since these functions are so common, it is customary to omit the parentheses: *Arrow types are right-associative*.

Practice. Determine the types for the following functions. Do not worry about implementing the functions, you just need to determine their type.

1. take from the standard functions. Assume the elements in the list are integers.
2. take from the standard functions. Assume the elements in the list are integers.
3. hasEnough from last time. Assume the elements in the list are integers.
4. isSubstring: Given a string and another string, it returns whether the first string is contained somewhere within the second string.
5. max3: Given three numbers, returns the maximum of the tree.
6. evaluate: This function is called with two (curried arguments). The first argument is a function `f` that takes as input an integer, and returns as output an integer. The second argument is an integer. The result is what happens when we apply `f` to that second argument.