Polymorphism

Identity Function

```
id :: a -> a
```

- `a` is a *type variable* – can be of any type
- This `id` function accepts a parameter of *any* type, and returns something of the *same type* (once a type is bound to the first (input) `a`, the second (output) `a` is bound to that same type)

Consider a call to this function:

```
x = id "Ih-Ah!"
```

In this specific call:

- The type variable `a` will be *bound* to the type `String`
- as a result, `x`'s type will also be `String`

Question

The `id` function stands for 'identity'. Can you guess what the body of the function is?

Hint:

1. Input is of *any type*
2. Output is of *same type* as input
3. `id` knows/assumes *nothing* of the input type
4. So what could `id` possibly return?

Answer:

```
id :: a -> a
id x = x
```

Tuples

We previously defined a `Point` type as:

```
data Point = Point Float Float
```

This is one way to do 'collection'. Another way is to use a tuple:

```
type Point = (Float, Float)
```

This might be weird to look at, but remember: the thing on the RHS, `(Float, Float)`{.hs}, is a *type*.

We could construct a value of this type as follows:

```
point :: Point -- same as `point :: (Float, Float)`
point = (1.2, 3.4)
```

Polymorphic Tuples

Consider the function:

```
fst :: (a, b) -> a
```

By looking at fst‘{.hs}’s type signature, we can see that:

- **Input:** A tuple containing two values of *any* type
- **Output:** The type of the first element of the input tuple

Question

What does the body of `fst` look like?

Answer:

```
fst :: (a, b) -> a
fst (x, _) = x
```

Question

Now consider a function `snd` that returns the second element of a tuple:

```
snd (_, y) = y
```

What should `snd`’s type signature look like?

Answer:

```
snd :: (a, b) -> b
```

Lists

Type type of an `ArrayList` of `Int`s in Java would look like:

```
ArrayList<Int>
```

The *type* of a list of `Int`s in Haskell looks like:

```
[Int] -- read as 'list of `Int`s'
```

An example of constructing such a list:

```haskell
l :: [Int]
l = [1, 2, 3]
```

We could replace the `Int` above with *any* type, because a list can hold anything. In Java, we could represent this with:

```
ArrayList<T>
```

where `T` is a type variable denoting the fact that `ArrayList` can hold any type.

The same idea in Haskell is expressed as:

```haskell
[a] -- read this as 'list containing elements of any type'
```

where `a` is the type variable (equivalent of `T` in the Java example).

Constructing Lists

As noted above, we can create a list like this:

```haskell
l :: [Int]
l = [1, 2, 3]
```

The `[1, 2, 3]` is just *syntactic sugar* for `1:2:3:[]`, so the following has the same effect:

```haskell
l :: [Int]
l = 1 : 2 : 3 : []
```

Think of it like this: The list type in Haskell has two constructors. The first is `[]`, which creates an empty list. This is akin to the `Zero` constructor that we used for the `Peano` data type. The type of `[]` is:

```haskell
[] :: [a]
```

We could create a value containing the empty list by:

```haskell
empty = []
```

The second constructor is `:`, and its type is:

```haskell
-- the parenthesis are needed since `:` is a symbol
(:) :: a -> [a] -> [a]
```

`:` is an *infix operator*, just like `+`, which means it's a function that appears *between* its arguments.

Let's go back to:

```haskell
l :: [Int]
l = 1 : 2 : 3 : []
```

The `:` operator is right-associative, which means the function applications take place like:

```haskell
l :: [Int]
l = 1 : (2 : (3 : []))
```

You can think of the `:` operator as doing something like this (note: the following is *not* valid haskell):

```
x0 : [x1, x2, ...] = [x0, x1, x2, ...]
```

So `:` simply prepends the LHS value to the RHS list.

Now we can follow the functions calls (again, not quite valid Haskell):

```
l = 1 : (2 : (3 : []))
  = 1 : (2 : [3])
  = 1 : [2, 3]
  = [1, 2, 3]
```