```
id x = x
```

This simply returns its argument, which can be of *any type*. The type of `id` is given by:

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

Consider these other examples of functions:

```
cond (x, y, z) = if x then y else z
apply (f, x)   = f x
```

Their types can be derived as follows:

```
cond   :: (Bool, a, a) -> a
apply  :: (a -> b, a)  -> b
```

2.6.4 List polymorphism

Consider the *list* data type. If this was not already built into Haskell, it could be defined as a data type by specifying the properties of a list. For example, the following (recursive) data type represents a list of integers:

```
data IntList = Nil
             | Cons Int IntList
    deriving Show
```

We could symbolically represent what we understand as the list `[14, 9, 12]` by the expression:

```
Cons 14 (Cons 9 (Cons 12 Nil))
```

We could similarly define a function to find the length of such a list:

```
lengthIntList  Nil = 0
lengthIntList (Cons _ xs) = 1 + lengthIntList xs
```

However, the list we have defined is restricted to integers. The property of 'listness' has nothing to do with the type of elements in the list (although they *are* constrained to be all of the same type), yet we have had to specify such an unnecessary property when defining the list.

Using *polymorphic data definitions* we can represent such data structures in the way we would like, without unnecessary constraints. Consider the following definition of a list data structure:

```
data List a = Cons' a (List a)
            | Nil'
    deriving Show
```

List is a type constructor, Cons' a value constructor
is a function returning a value of type List a

We have parameterized the type name `List` with a type variable `a`. At least one of the arguments of a constructor for that type must be of the type given by this type variable. In this case we have one type variable, and we have specified that the first argument to the `Cons'` constructor (that is, the list element) will be of this type. The second argument to the `Cons'` constructor is of type `List a`, which means that it is a list which must have been constructed with the same type variable, thus ensuring that all list elements are of the same type.

Using this data type, we can write down lists of any element type, for example:

```
Cons' 5 (Cons' 14 (Cons' (-3) Nil'))                  = [5, 14, -3]
Cons' False Nil'                                      = [False]
Cons' Nil'(Cons'(Cons' 4.7(Cons' 2.6 Nil'))Nil') = [[],[4.7,2.6]]
```

The last example is a value of type `List (List Float)`.

The predefined list type has the same property and so most general purpose functions which operate on lists are polymorphic. For example:

```
length []     = 0
length (x:xs) = 1 + length xs
```

We know that the argument accepted by `length` must be a list, but the function is independent of the type of the list elements. Therefore, the type of `length` is given by:

```
length :: [a] -> Int
```

We are now in a position to determine the type of the empty list (that is, `[]`): it is denoted by `[a]`. Not all list functions are polymorphic. For example, the formal argument patterns may dictate that the list elements are of a certain type, as in the following case:

```
anyTrue []       = False
anyTrue (True:_) = True
anyTrue (_:xs)   = anyTrue xs
```

Because of the `True` constant appearing in the argument pattern, the type of `anyTrue` is given by:

```
anyTrue :: [Bool] -> Bool
```

2.6.5 Trees

The list data structure is the main data structure considered so far. It is called a linear structure because its items appear in a predetermined sequence, that is each item has at most one immediate successor. A tree is a *non-linear* data structure because an item can have one or more successors. Trees have many applications: for example, to represent expressions containing nested function calls.

:t Cons'
=> Cons' :: a -> List a -> List a