Lesson 12: Creating your own types
**Type synonyms**

Imagine that you want to implement a `patientInfo` function which takes

the first name the last name the age the height

of a patient, and turns them into a string:

```
patientInfo :: String -> String -> Int -> Int -> String
patientInfo fname lname age height = name ++ " " ++ ageHeight
  where name = lname ++ ", " ++ fname
  ageHeight = "(" ++ show age ++ "yrs. " ++ show height ++ "in.)"
```

The `patientInfo` function behaves in the following way:

```
ghci> patientInfo "John" "Doe" 43 74
"Doe, John (43yrs. 74in.)"
ghci> patientInfo "Jane" "Smith" 25 62
"Smith, Jane (25yrs. 62in.)"
```
Type synonyms

We have seen in Lesson 11 that the two types

\[
\text{String} \quad \quad \quad \quad \quad [\text{Char}]
\]

are in fact two different ways to describe \textbf{the very same type} in Haskell.

More generally, one can create new type synonyms using the \texttt{type} keyword:

\[
\begin{align*}
\text{type FirstName} &= \text{String} \\
\text{type LastName} &= \text{String} \\
\text{type Age} &= \text{Int} \\
\text{type Height} &= \text{Int}
\end{align*}
\]

In this way, one can rewrite the original type signature as:

\[
\text{patientInfo :: FirstName} \to \text{LastName} \to \text{Age} \to \text{Height} \to \text{String}
\]

Does not change much for the compiler, and very convenient for the programmer!
Type synonyms

It makes sense to store the patient names as a tuple of strings.

It may be thus convenient to create the type synonym

\[
\text{type PatientName} = (\text{String},\text{String})
\]

as well as a few helper functions such as

\[
\begin{align*}
\text{firstName} & : \text{PatientName} \to \text{String} \\
\text{firstName} \text{ patient} & = \text{fst \ patient} \\
\text{lastName} & : \text{PatientName} \to \text{String} \\
\text{lastName} \text{ patient} & = \text{snd \ patient}
\end{align*}
\]

The resulting code can be tested in ghci, in the following way:

\[
\begin{align*}
\text{ghci}\text{> testPatient} & = ("John","Doe") \\
\text{ghci}\text{> firstName testPatient} & \\
"John" & \\\n\text{ghci}\text{> lastName testPatient} & \\
"Doe"
\end{align*}
\]
Creating a new type \textit{Gender}

Now, let us add the gender of the patient, which can be male or female.

Creating a new type can be done using the \textit{data} keyword, as follows:

\begin{verbatim}
data Gender = Male | Female
\end{verbatim}

The symbol | indicates that a value of type \textit{Gender} is either \textit{Male} or \textit{Female}. 

\textit{Gender} is the \textbf{type constructor}, \textit{Male} and \textit{Female} are the \textbf{data constructors}. 

The data constructors can be used just like values (for example True and False).
Creating a new type Gender

Interestingly, the type `Bool` of booleans is defined in just the same way:

```
data Bool = True | False
```

Here, `Bool` is the **type constructor**, `True` and `False` are the **data constructors**. The symbol `|` indicates that a value of type `Bool` is either `True` or `False`. 
Creating a new type \texttt{Gender}

Once the type \texttt{Gender} created using

\begin{verbatim}
data Gender = Male | Female
\end{verbatim}

we can use this new type to define functions and their type signature.

For instance, the \texttt{genderInitial} function can be defined in this way:

\begin{verbatim}
genderInitial :: Gender \rightarrow \text{Char} 
genderInitial Male = 'M' 
genderInitial Female = 'F'
\end{verbatim}

Although the type \texttt{Bool} could be equivalently used to describe the gender of patients, it makes more sense for readability and clarity to use the type \texttt{Gender} just created.
Creating a new type **BloodType**

Now, imagine that we want to describe the **blood type** of our patients.

The **ABO blood type** may have four values: A, B, AB and O.

The **Rhesus** can be positive or negative.

Hence, every patient has a **blood type** of the form:

\[ A^+, A^-, B^+, B^-, AB^+, AB^-, O^+ \text{ and } O^- \]

The blood type could be represented by a Haskell type **BloodType** naively defined by a long range of data constructors:

```haskell
data BloodType = APos | ANeg | BPos | BNeg | ABPos | ABNeg | OPos | ONeg
```

However, a more elegant and modular definition of **BloodType** is also possible.
Creating a new type **BloodType**

One starts by creating the type **RhType** in the same way as the type **Gender**:

```haskell
data RhType = Pos | Neg
```

Then one creates the type **ABOType** using four data constructors:

```haskell
data ABOType = A | B | AB | O
```

Finally, one defines the type **BloodType** as follows:

```haskell
data BloodType = BloodType ABOType RhType
```

We choose to use the same name **BloodType** for the type and data constructor.
Creating a new type **BloodType**

We are now ready to produce **BloodType** data:

```haskell
patient1BT :: BloodType
patient1BT = BloodType A Pos

patient2BT :: BloodType
patient2BT = BloodType 0 Neg

patient3BT :: BloodType
patient3BT = BloodType AB Pos
```

We explain now how to

1. print these **BloodType** data
2. use them to determine when a patient can donate blood to another patient.
Printing a **BloodType** data (in two steps)

First, the two functions `showRh` and `showABO` are defined by pattern matching:

```haskell
showRh :: RhType -> String
showRh Pos = "+
showRh Neg = "-

showABO :: ABOType -> String
showABO A = "A"
showABO B = "B"
showABO AB = "AB"
showABO O = "O"
```

The `showBloodType` function is then defined by combining the two functions:

```haskell
showBloodType :: BloodType -> String
showBloodType (BloodType abo rh) = showABO abo ++ showRh rh
```
Displaying a **BloodType** data (in two steps)

First, the two functions `showRh` and `showABO` are defined by **pattern matching**:

```haskell
showRh :: Rhtype -> String
showRh Pos = "+
showRh Neg = "-

showABO :: ABOType -> String
showABO A = "A"
showABO B = "B"
showABO AB = "AB"
showABO O = "O"
```

The `showBloodType` function is then defined by combining the two functions:

```haskell
showBloodType :: BloodType -> String
showBloodType (BloodType abo rh) = showABO abo ++ showRh rh
```

Notice in this last step how pattern matching is used to easily extract the **ABOType** and **RhType** components `abo` and `rh` of the data constructor `BloodType`. 
Testing *BloodType* for donor compatibility

An important question is when a patient can donate blood to another patient.

The rules for blood donation are the following ones:

\begin{center}
\begin{tabular}{ll}
(1) & A can donate to A and AB, \\
(2) & B can donate to B and AB, \\
(3) & AB can donate only to AB, \\
(4) & O can donate to anyone. \\
\end{tabular}
\end{center}

Note that O can donate to everyone and that AB can receive from everyone.

*NB: we do not worry about the Rhesus compatibility here.*
Testing **BloodType** for donor compatibility

The `canDonateTo` function is defined by **pattern matching**:

```haskell
    canDonateTo :: BloodType -> BloodType -> Bool
    canDonateTo (BloodType 0 _) _ = True
    canDonateTo _ (BloodType AB _) = True
    canDonateTo (BloodType A _) (BloodType A _) = True
    canDonateTo (BloodType B _) (BloodType B _) = True
    canDonateTo _ _ = False
```

Remember that pattern matching is performed sequentially.

The first case of pattern matching

```haskell
    canDonateTo (BloodType 0 _) _ = True
```

indicates that 0 is a universal donor.
Testing **BloodType** for donor compatibility

The `canDonateTo` function is defined by pattern matching:

```haskell
    canDonateTo :: BloodType -> BloodType -> Bool
    canDonateTo (BloodType 0 _) _ = True
    canDonateTo _ (BloodType AB _) = True
    canDonateTo (BloodType A _) (BloodType A _) = True
    canDonateTo (BloodType B _) (BloodType B _) = True
    canDonateTo _ _ = False
```

Remember that pattern matching is performed sequentially.

The second case of pattern matching

```haskell
    canDonateTo _ (BloodType AB _) = True
```

indicates that AB is a universal receiver.
Testing **BloodType** for donor compatibility

The `canDonateTo` function is defined by **pattern matching**:

```haskell
    canDonateTo :: BloodType -> BloodType -> Bool
    canDonateTo (BloodType 0 _) _ = True
    canDonateTo _ (BloodType AB _) = True
    canDonateTo (BloodType A _) (BloodType A _) = True
    canDonateTo (BloodType B _) (BloodType B _) = True
    canDonateTo _ _ = False
```

Remember that pattern matching is performed sequentially.

The third case of pattern matching

```
    canDonateTo (BloodType A _) (BloodType A _) = True
```

indicates that A can donate to A.
Testing **BloodType** for donor compatibility

The `canDonateTo` function is defined by **pattern matching**:

```haskell
  canDonateTo :: BloodType -> BloodType -> Bool
  canDonateTo (BloodType 0 _) _ = True
  canDonateTo _ (BloodType AB _) = True
  canDonateTo (BloodType A _) (BloodType A _) = True
  canDonateTo (BloodType B _) (BloodType B _) = True
  canDonateTo _ _ = False
```

Remember that pattern matching is performed sequentially.

The fourth case of pattern matching

```haskell
  canDonateTo (BloodType B _) (BloodType B _) = True
```

indicates that B can donate to B.
Testing **BloodType** for donor compatibility

The `canDonateTo` function is defined by **pattern matching**:

```haskell
    canDonateTo :: BloodType -> BloodType -> Bool
    canDonateTo (BloodType 0 _) _ = True
    canDonateTo _ (BloodType AB _) = True
    canDonateTo (BloodType A _) (BloodType A _) = True
    canDonateTo (BloodType B _) (BloodType B _) = True
    canDonateTo _ _ = False
```

Remember that pattern matching is performed sequentially.

The last case of pattern matching

```haskell
    canDonateTo _ _ = False
```

indicates that the donor and receiver are incompatible in all the remaining cases.
Testing **BloodType** for donor compatibility

One can then test the `canDonateTo` function in the following way:

```
ghci> canDonateTo patient1BT patient2BT
False
ghci> canDonateTo patient2BT patient1BT
True
ghci> canDonateTo patient2BT patient3BT
True
ghci> canDonateTo patient1BT patient3BT
True
ghci> canDonateTo patient3BT patient1BT
False
```

So nice and easy!
Supporting different names: *MiddleName* and *Name*

In order to illustrate the flexibility of Haskell type system, we create the type *Name*

which enables us to manipulate two different forms of names:

1. the names with a first name and a last name
2. the names with a first name, a middle name, and a last name

The type *Name* is created in the following way:

```haskell
  type MiddleName = String
  data Name = Name FirstName LastName
              | NameWithMiddle FirstName MiddleName LastName
```
Creating the **Patient** type

At the beginning of the lesson, we introduced a `patientInfo` function:

```haskell
patientInfo :: String -> String -> Int -> Int -> String
patientInfo fname lname age height = name ++ " " ++ ageHeight
  where name = lname ++ ", " ++ fname
    ageHeight = "(" ++ show age ++ "yrs. " ++ show height ++ "in."")"
```

with the idea of collecting all the useful information of a patient.

Fine!

However, now that we know how to create a Haskell type adapted to specific data... we want to create a **Patient** type which contains all the information and more!
Creating the **Patient** type

The first step is to enlist all the information we want to collect, as well as its type:

(1) Name: Name  
(2) Gender: Gender  
(3) Age (years): Int  
(4) Height (inches): Int  
(5) Weight (pounds): Int  
(6) Blood type: BloodType

One deduces from that list the appropriate **Patient** type to be created:

```haskell
data Patient = Patient Name Gender Int Int Int Int BloodType
```

Note that we use **Patient** as name for the type as well as for the data constructor.
Creating and using the Patient type

A patient will be then registered as a value of type Patient, in the following way:

```
johnDoe :: Patient
johnDoe = Patient (Name "John" "Doe") Male 30 74 200 (BloodType AB Pos)
```

The three helper functions are then defined by pattern matching:

```
getName :: Patient -> Name
getName (Patient name _ _ _ _) = name

getAge :: Patient -> Int
getAge (Patient _ _ age _ _) = age

getBloodType :: Patient -> BloodType
getAge (Patient _ _ _ _ _ bt) = bt
```

Nice... but could become a little bit annoying if there are too many fields!
Using the record syntax for the Patient type

Earlier in the lesson, we have defined the Patient type in the following way:

```haskell
data Patient = Patient Name Gender Int Int Int Int BloodType
```

Now, we define it using the record syntax:

```haskell
data Patient = Patient { name :: Name
                          , gender :: Gender
                          , age :: Int
                          , height :: Int
                          , weight :: Int
                          , bloodType :: BloodType }
```
Using the record syntax for the Patient type

The record syntax is extremely convenient.

For instance, one registers a new patient in that way:

```plaintext
jackieSmith :: Patient
jackieSmith = Patient { name = Name "Jackie" "Smith"
                         , age = 43
                         , gender = Female
                         , height = 62
                         , weight = 115
                         , bloodType = BloodType O Neg }
```

Note that the order of the fields does not matter here.
Using the record syntax for the **Patient** type

Moreover, each field in the record creates a function to access that value:

```
ghci> height JackieSmith
62
ghci> showBloodType (bloodtype jackieSmith)
"0-"
```

It is also possible to update values in the record syntax. Typically:

```
jackieSmithUpdated = jackieSmith { age = 44 }
```

Since we are still in the purely functional world, a new value of **Patient** type will be created, and must be assigned to a variable like `jackieSmithUpdated` to be useful.
Thank you for your attention!