Lesson 17

design by composition:
semigroups and monoids
Introduction to composability – combining functions

It is possible in Haskell to compose two functions in the following way:

```haskell
myLast :: a -> a
myLast = head . reverse

myMin :: Ord a -> [a] -> a
myMin = head . sort

myAll :: (a -> Bool) -> [a] -> Bool
myAll testfunc = (foldr (&&) True) . (map testFunc)
```

Note that using the `sort` function requires to import the `Data.List` module.
Introduction to composability – combining functions

It is possible in Haskell to compose two functions in the following way:

myLast :: a -> a
myLast = head . reverse

myMin :: Ord a -> [a] -> a
myMin = head . sort

myAll :: (a -> Bool) -> [a] -> Bool
myAll testfunc = (foldr (&&) True) . (map testFunc)

Here, the myAll function tests whether a property is true of all items in a list.
The **Semigroup** type class

The **Semigroup** type class has a unique method: the $<>$ operator of type

$$ (<>) :: \text{Semigroup } a \Rightarrow a \rightarrow a \rightarrow a $$

The $<>$ operator can be defined as the binary function + in the case of integers:

```
instance Semigroup Integer where
  (<>) x y = x + y
```

The `instance` keyword is used to make `Integer` an instance of the `Semigroup` type class.

The operator $<>$ is defined as addition between integers.
Defining the **Color** type

In this lesson, we are interested in mixing colors together!

To that purpose, we define the type **Color** whose data constructors are colors:

```haskell
data Color = Red | Yellow | Blue | Green | Purple | Orange | Brown deriving (Show,Eq)
```
Turning the **Color** type into an instance of **Semigroup**

We then implement the type class **Semigroup** for the **Color** type just defined. The idea is to define $<>$ as the operation of mixing two colors:

```haskell
instance Semigroup Color
        (<> ) Red Blue = Purple
        (<> ) Blue Red = Purple
        (<> ) Yellow Blue = Green
        (<> ) Yellow Red = Orange
        (<> ) Red Yellow = Orange
        (<> ) a b = if a == b
            then a
            else Brown
```

Playing with the **Color** type

We can then play with colors in the following way:

```
ghci> Red <> Yellow
Orange
ghci> Red <> Blue
Purple
ghci> Green <> Purple
Brown
```
Challenge: the operator $\langle\rangle$ is not associative!

Associativity of addition means that the order

left-to-right: $(x + y) + z$  
right-to-left: $x + (y + z)$

in which one applies the binary operation $+$ on three elements does not matter:

$$x + (y + z) = (x + y) + z$$

Note that the binary operation $\langle\rangle$ is not associative since:

```
ghci> (Green <$> Blue) <$> Yellow
  Brown
ghci> Green <$> (Blue <$> Yellow)
  Green
```

The associativity property is (morally) required of the type class `Semigroup` although it cannot be checked by the Haskell compiler (one needs to shift to a proof assistant based on dependent types such as Agda, Coq or Lean for that).
Guards work much like pattern matching but allow you to do some computation on the arguments you are going to compare.

The argument is checked first, and then the function is defined much as in pattern matching.

```haskell
howMuch :: Int -> String
howMuch n | n > 10 = "a whole bunch"
           | n > 0  = "not much"
           | otherwise = "we're in debt!"
```

Guards separate conditions, and otherwise is the default case.
Making **Color** associative using guards

We obtain the following implementation of **Semigroup** for the **Color** type:

```
instance Semigroup Color
  (<>) Red Blue = Purple
  (<>) Blue Red = Purple
  (<>) Yellow Blue = Green
  (<>) Yellow Red = Orange
  (<>) Red Yellow = Orange
  (<>) a b | a == b = a
    | all ('elem' [Red,Blue,Purple]) [a,b] = Purple
    | all ('elem' [Blue,Yellow,Green]) [a,b] = Green
    | all ('elem' [Red,Yellow,Orange]) [a,b] = Orange
    | otherwise Brown
```
Making **Color** associative using guards

Once defined in this way, the operator `<>` becomes associative.

In particular, the previous case of non-associativity has been repaired:

```
ghci> (Green <> Blue) <> Yellow
Green
ghci> Green <> (Blue <> Yellow)
Green
```
The **Monoid** type class

The **Monoid** type class is very similar to **Semigroup**. The only major difference is that **Monoid** requires a neutral element for the type. Here is the definition of the type class:

```haskell
class Monoid a where
  mempty :: a
  mappend :: a -> a -> a
  mconcat :: [a] -> a
```

The binary operator **mappend** has the same type signature and plays the same role for the type class **Monoid** as the binary operator `<>` for the type class **Semigroup**.

The constant **mempty** is the name used for the neutral element.
The **Monoid** type class

The choice of names for the functions comes from the fact that for every type \( a \), the associated type \([a]\) of lists is an instance of the **Monoid** type class, with the common list functions:

\[
\begin{align*}
\text{empty} &: [a] \\
\text{append} &: [a] \to [a] \to [a] \\
\text{concat} &: [[a]] \to [a]
\end{align*}
\]

The strange names `mempty`, `mappend` and `mconcat` of the **Monoid** type class are just obtained by extending each name with the letter "m" for "monoid".
Combining multiple elements at once: the `mconcat` function

We illustrate how to use the `mconcat` function

\[
mconcat :: \text{Monoid } a \Rightarrow [a] \rightarrow a
\]

with the basic example of a list of strings to concatenate:

\[
ghci> \ mconcat ["does," this"," make"," sense?"]
"does this make sense?"
\]

Remember that `String` is a synonym for `[Char]` and thus an instance of `Monoid`. 
The type **PTable** of event-probability tables

The two types **Events** and **Probs** are defined as the list types:

```haskell
type Events = [String]
type Probs = [Double]
```

The type **PTable** is then defined as follows:

```haskell
data PTable = PTable Events Probs
```

A value of type **PTable** is thus a term of the form

```
PTable events probs
```

consisting of a data constructor **PTable**, of a list **events** of strings (the events) and of a list **probs** of doubles (the probabilities).
The `createPTable` function

The `createPTable` function takes two arguments

1. a value `events` of type `Events`
2. a value `probs` of type `Probs`

and creates a value of type `PTable` in this way:

```haskell
createPTable :: Events -> Probs -> PTable
createPTable events probs = PTable events normalizedProbs
  where totalProbs = sum probs
        normalizedProbs = map (\x -> x/totalProbs) probs
```

The probabilities are normalized (that is, their sum is 1) thanks to the function:

```haskell
map (\x -> x/totalProbs) probs
```
Defining **dice** and **spinner** as probability tables

Using `createPTable`, we can define **dice** and **spinner** as values of **PTable** type:

```haskell
coin :: PTable
coin = createPTable ["heads","tails"] [1,1]

spinner :: PTable
spinner = createPTable ["red","blue","green"] [1,2,7]
```

Here, **coin** represents a fair coin while **spinner** represents a color spinner with different probabilities for each color red, blue and green.
Making **PTable** an instance of the **Show** type class

We would like to print the values of type **PTable**.
To that purpose, we start by implementing the following function:

```haskell
showPair :: String -> Double -> String
showPair event prob = mconcat [event,"|",show prob,\"\n"]
```

The **showPair** function may be illustrated as follows:

```haskell
ghci> showPair "heads" 0.5
heads|0.5
ghci> showPair "tails" 0.5
tails|0.5
```
Making **PTable** an instance of the **Show** type class

This enables us to implement the type class **Show** for the **PTable** type, in the following way:

```haskell
instance Show PTable where
  show (PTable events probs) = mconcat pairs
  where pairs = zipWith showPair events probs
```

Here, the `zipWith` function behaves as follows:

```haskell
ghci> zipWith (+) [1,2,3] [4,5,6] [5,7,9]
```
Next step: making PTable an instance of Monoid

One obtains the following show function for the PTable type:

```
ghci> createPTable ["heads","tails"] [0.5,0.5]
heads|0.5
tails|0.5
```

We would like to turn PTable into an instance of Monoid so that we can combine two PTables (or more). For instance, if we have two coins, we can combine them in order to obtain an outcome like this:

```
  heads-heads|0.25
  heads-tails|0.25
  tails-heads|0.25
  tails-tails|0.25
```

This requires generating a combination of all events and all probabilities of two PTables using what is called cartesian product.
The \textsf{cartCombine} function for the cartesian product of lists

The \textsf{cartCombine} function takes three arguments:

1. a function \texttt{func:a->b->c} for combining the two lists
2-3. two lists of respective types \texttt{[a]} and \texttt{[b]}.

and returns a list of type \texttt{[c]} defined as follows:

\begin{verbatim}
cartCombine :: (a -> b -> c) -> [a] -> [b] -> [c]
cartCombine func l1 l2 = zipWith func newL1 cycledL2
  where nToAdd = length l2
    repeatedL1 = map (take nToAdd . repeat) l1
    newL1 = mconcat repeatedL1
    cycleL2 = cycle l2
\end{verbatim}
The `cartCombine` function for the cartesian product of lists

Suppose that `l1` and `l2` are two lists of strings equal to

\[ l1 = ["heads","tails"] \quad l2 = ["blue","red","green"] \]

The `repeat` function takes the list `l1` as argument and returns the infinite list

\[ ["heads","tails"],["heads","tails"],["heads","tails"],... \]

The `take nToAdd` function defined by partial application applied to `repeat l1` returns a finite list with the same length `nToAdd` as the length of `l2`:

\[ ["heads","tails"],["heads","tails"],["heads","tails"] \]

The `mconcat` then "flattens" the list of lists into the list `newL1`:

\[ ["heads","tails","heads","tails","heads","tails"] \]

Finally, one applies the `zipWith` function to the `func` function, to the list `newL1` and to the infinite cyclic list `cycledL2` to get the cartesian product of `l1` and `l2`. 

```python
l1 = ["heads","tails"]
l2 = ["blue","red","green"]
```
The *combineEvents* and *combineProbs* functions

We then use the *cartCombine* function just defined

\[
\text{cartCombine} :: (a \rightarrow b \rightarrow c) \rightarrow [a] \rightarrow [b] \rightarrow [c]
\]

to implement the two *combineEvents* and *combineProbs* functions below:

- \[
\text{combineEvents} :: \text{Events} \rightarrow \text{Events} \rightarrow \text{Events}
\]
  \[
\text{combineEvents} \ e1 \ e2 = \text{cartCombine} \ \text{combiner} \ e1 \ e2
\]
  \[
\text{where combiner} = (\ x \ y \rightarrow \text{mconcat} \ [x,"-",y])
\]

- \[
\text{combineProbs} :: \text{Probs} \rightarrow \text{Probs} \rightarrow \text{Probs}
\]
  \[
\text{combineProbs} \ p1 \ p2 = \text{cartCombine} \ (*) \ p1 \ p2
\]

When combining events, one hyphenates the event names.
To combine probabilities, one multiplies them.
The `combineEvents` and `combineProbs` functions

By way of illustration,

```
ghci> combineEvents ["heads","tails"] ["red","blue","green"]
["heads-red","heads-blue","heads-green","tails-red",
"tails-blue","tails-green"]

ghci> combineProbs [0.5,0.5] [0.1,0.2,0.7]
[5.0e-2,0.1,0.35,5.0e-2,0.1,0.35]
```
Making **PTable** type as an instance of **Semigroup**

Finally, we use the `combineEvents` and `combineProbs` functions to implement the type class **Semigroup** for the **PTable** type:

```haskell
instance Semigroups PTable where
    (<>) ptable1 (PTable [] []) = ptable1
    (<>) (PTable [] []) ptable2 = ptable2
    (<>) (PTable e1 p1) (PTable e2 p2) = createPTable newEvents newProbs
        where newEvents = combineEvents e1 e2
              newProbs = combineProbs p1 p2
```

Handles the special case of having an empty **PTable**
PTable as an instance of Semigroup

We may then combine two probability tables (or more) using the $<$> operator of the Semigroup type class.

By way of illustration, it is possible to combine a coin and a spinner in this way:

```ghci
ghci> coin <> spinner
heads-red|5.0e-2
heads-blue|0.1
heads-green|0.35
tails-red|5.0e-2
tails-blue|0.1
tails-green|0.35
```
Making PTable an instance of Monoid

It is not difficult to make the PTable type an instance of Monoid using what we have already done for the Semigroup type class.

Indeed, we may simply write for the instance of Monoid for PTable:

```haskell
instance Monoid PTable where
  mempty = PTable [] []
  mappend = (<>)
```

Note in particular that it is not necessary to define mconcat.

We thus gain the power of mconcat for free!
PTable an instance of Monoid

This is illustrated by the following example:

```
ghci> mconcat [coin,coin,coin]
heads-heads-heads|0.125
heads-heads-tails|0.125
heads-tails-heads|0.125
heads-tails-tails|0.125
tails-heads-heads|0.125
tails-heads-tails|0.125
tails-tails-heads|0.125
tails-tails-tails|0.125
```

Note that, in this case, each outcome has the same probability: $1/8 = 12.5\%$
Thank you for your attention!