Functional Programming

Paul Mellies

Lesson 4 : First-Class Functions
Starting point: three basic functions

Consider the three very similar Haskell functions:

```haskell
ifEvenInc n = if even n
            then n + 1
            else n
```

```haskell
ifEvenDouble n = if even n
               then n * 2
               else n
```

```haskell
ifEvenSquare n = if even n
                then n^2
                else n
```

Can we encode the three functions in a more uniform way?
Higher-order functions

The idea is to implement a higher-order function

```haskell
ifEven myFunction x = if even x
    then myFunction x
    else x
```

which will take the first-order functions below as arguments:

- `inc n = n + 1`
- `double n = n*2`
- `square n = n^2`

and produce in this way each of the three functions:

- `ifEvenInc n = ifEven inc n`
- `ifEvenDouble n = ifEven double n`
- `ifEvenSquare n = ifEven square n`
Lambda functions as arguments

Note that naming a function such as

\[
\text{double } n = n \times 2
\]

is a good idea, but you can also use the lambda function (without a name)

\[
\lambda x \rightarrow x \times 2
\]

in order to define your function of interest, in the expected way:

\[
\text{ghci}\text{> ifEven } (\lambda x \rightarrow x \times 2) 6 \\
12
\]
Illustration: custom sorting

Suppose that you have a list of first names and last names of authors.

The full name of each author is represented as a tuple of two items:

\[
\text{author } = ("Alonzo","Church")
\]

The functions \texttt{fst} and \texttt{snd} give us access to the first and second element of the tuple (also called a pair)

\begin{verbatim}
ghci> fst author
"Alonzo"
ghci> snd author
"Church"
\end{verbatim}
Illustration: custom sorting

Once the **Data.List** module imported using the instruction

```
import Data.List
```

the built-in function **sort** can be applied on the list of names

```
names = [("Ian","Curtis"),
         ("Bernard","Sumner"),
         ("Peter","Hook"),
         ("Stephen","Morris")]
```

in order to obtain the list sorted by first names:

```
ghci> sort names
[("Bernard","Sumner"),("Ian","Curtis"),
 ("Peter","Hook"),("Stephen","Morris")]
```
Illustration: custom sorting

Now, consider the function

\[
\text{compareLastNames } \text{name1 } \text{name2 } = \begin{cases} 
\text{GT} & \text{if } \text{lastName1} > \text{lastName2} \\
\text{LT} & \text{if } \text{lastName1} < \text{lastName2} \\
\text{EQ} & \text{else}
\end{cases}
\]

where \( \text{lastName1} = \text{snd name1} \)

\( \text{lastName2} = \text{snd name2} \)

which takes two names \( \text{name1} \) and \( \text{name2} \) and returns the special value

- \( \text{GT} \) when \( \text{name1} \) is strictly greater than \( \text{name2} \) on the last name
- \( \text{LT} \) when \( \text{name1} \) is strictly less than \( \text{name2} \) on the last name
- \( \text{EQ} \) when \( \text{name1} \) and \( \text{name2} \) are equal on the last name
Illustration: custom sorting

The built-in function `sortBy` is then used to sort the list:

```
ghci> sortBy compareLastNames names
[("Ian","Curtis"),("Peter","Hook"),
 ("Stephen","Morris"),("Bernard","Sumner")]
```

where the names are now sorted with respect to the comparison function

```
compareLastNames
```

which compares the last names of two names `name1` and `name2`. 
Refining the original comparison function

The previous comparison function

```plaintext
compareLastNames
```

does not compare the first names when the two last names are equal.

This undesired behavior is corrected in the code below:

```plaintext
compareLastNames name1 name2 = if lastName1 > lastName2
  then GT
  else if lastName1 < lastName2
    then LT
    else if firstName1 > firstName2
      then GT
      else if firstName1 < firstName2
        then LT
        else EQ

where lastName1 = snd name1
  lastName2 = snd name2
  firstName1 = fst name1
  firstName2 = fst name2
```
The benefits of functions returning functions

Imagine that you wish to send a newsletter to the members of your Secret Society dispatched at three regional post office boxes, in San Francisco, New York and Reno:

- PO Box 1234, San Francisco, CA, 94111
- PO Box 789, New York, NY, 10013
- PO Box 456, Reno, NV, 89523

To that purpose, you will find useful to write the Haskell function

```haskell
addressLetter name location = nameText ++ " - " ++ location
where nameText = (fst name) ++ " " ++ (snd name)
```

which transforms `name` and `location` into the postal address of the letter:

```ghci
ghci> addressLetter ("Bob","Smith") "PO Box 1234 - San Francisco, CA, 94111"
"Bob Smith - PO Box 1234 - San Francisco, CA, 94111"
```
The benefits of functions returning functions

Now, imagine (1) that San Francisco added a new address

\[
\text{PO Box 1010, San Francisco, CA, 94109}
\]

for members whose last names start with the letter \( L \) of later in the alphabet,
(2) that New York wants the name followed by a colon rather than an hyphen,
(3) that Reno wants only the last names to be used for greater secrecy.

Clearly, one needs to define a new function for each office. Typically:

```plaintext
sfOffice name = if lastName < "L"
    then nameText
        ++ " - PO Box 1234 - San Francisco, CA, 94111"
    else nameText
        ++ " - PO Box 1010 - San Francisco, CA, 94109"

where lastName = snd name
nameText = (fst name) ++ " " ++ (snd name)
```
The three basic functions

The three functions for the San Francisco, New York and Reno offices:

```plaintext
sfOffice name = if lastName < "L"
   then nameText
       ++ " - PO Box 1234 - San Francisco, CA, 94111"
   else nameText
       ++ " - PO Box 1010 - San Francisco, CA, 94109"
where lastName = snd name
nameText = (fst name) ++ " " ++ (snd name)
```

```plaintext
nyOffice name = nameText ++ " : PO Box 789 - New York, NY, 10013"
where nameText = (fst name) ++ " " ++ (snd name)
```

```plaintext
renoOffice name = nameText ++ " - PO Box 456 - Reno, NV, 89523"
where nameText = snd name
```
The function `getLocationFunction` returns a function!

Once implemented each of the three basic functions

```
    nyOffice       sfOffice       renoOffice
```

which transforms `name` into the postal address, one defines the function

```
getLocationFunction location = case location of
  "ny" -> nyOffice
  "sf" -> sfOffice
  "reno" -> renoOffice
  _  -> (\name -> (fst name) ++ " " ++ (snd name))
```

which takes as argument the location represented as a string

```
    ny       sf       reno
```

and turns it into the corresponding function, or to a generic function otherwise.
Putting back all together and testing

One obtains a refined implementation of the function `addressLetter` as

```
addressLetter name location = locationFunction name
   where locationFunction = getLocationFunction location
```

The function `addressLetter` can be then tested on these basic examples:

```
ghci> addressLetter ("Bob","Smith") "ny"
"Bob Smith: PO Box 789 - New York, NY, 10013"

ghci> addressLetter ("Bob","Jones") "sf"
"Bob Jones - PO Box 1234 - San Francisco, CA, 94111"

ghci> addressLetter ("Samantha","Smith") "sf"
"Samantha Smith - PO Box 1010 - San Francisco, CA, 94109"

ghci> addressLetter ("Bob","Smith") "reno"
"Smith - PO Box 456 - Reno, NV, 89523"

ghci> addressLetter ("Bob","Smith") "la"
"Bob Smith"
```