Functional Programming
Homework no 1
Paul Mellies

Deadline for submission: Monday 9 March at 11pm (China time)

Exercise 1 : destructing & constructing pairs in Haskell

§1.a Describe the functions constructed using the \texttt{fst} and \texttt{snd} functions, which enables one to extract the three components

"hello" 8 "world"

of the expression built with pairs

\(("hello", 8), "world"\)

§1.b Define a \texttt{rebracket} function with the following behavior:

\begin{verbatim}
ghci> rebracket (("hello", 8), "world")
(("hello", (8, "world")))
ghci> rebracket (("blue", "green"), "red")
(\"blue\", (\"green\", \"red\"))
\end{verbatim}

Exercise 2 : Church numerals in Haskell

Alonzo Church introduced in the 1930s a very clever encoding of natural numbers in the \(\lambda\)-calculus, based on the idea that an integer should be understood as the number of times a given function \(s\) is applied to an argument \(z\). Typically, the natural numbers 0, 1, 2 and 3 are interpreted by Church as the following \(\lambda\)-terms:

\begin{align*}
\Gamma 0 &= \lambda s. \lambda z. z \\
\Gamma 1 &= \lambda s. \lambda z. \text{App}(s, z) \\
\Gamma 2 &= \lambda s. \lambda z. \text{App}(s, \text{App}(s, z)) \\
\Gamma 3 &= \lambda s. \lambda z. \text{App}(s, \text{App}(s, \text{App}(s, z)))
\end{align*}

written in the notation with an explicit application node (on the left), in the traditional notation (on the right), as well as in the 2-dimensional binary tree notation below:
The λ-terms 0, 1, 2 and 3 can be easily represented as Haskell expressions. Typically,
\[
\text{zero} = \lambda s \ z \rightarrow z \quad \text{one} = \lambda s \ z \rightarrow s \ z
\]

§2.a Write down the Haskell expressions for the Church numerals two and three.

§2.b Implement a function churchDecode which translates a Church numeral into the underlying natural number, in the following way:

```haskell
ghci> churchDecode zero
0
ghci> churchDecode one
1
ghci> churchDecode two
2
ghci> churchDecode three
3
```

§2.c Implement a function churchPrint which takes a Church numeral \( m \) as argument and translates it into a string, in the following way:

```haskell
ghci> churchPrint zero
"zero"
ghci> churchPrint one
"succ zero"
ghci> churchPrint two
"succ succ zero"
ghci> churchPrint three
"succ succ succ zero"
```

§2.d Implement a function churchOddEven which takes a Church numeral \( m \) and returns the integer 0 when the Church numeral is even, and the integer 1 when the Church numeral is odd.

```haskell
ghci> churchOddEven zero
0
ghci> churchOddEven one
1
ghci> churchOddEven two
0
ghci> churchOddEven three
1
```

§2.e Implement a unary function churchInc which takes a Church numeral as arguments and returns it as a Church numeral, incremented by one. The function churchInc is thus expected to behave in the following way:

```haskell
ghci> churchPrint (churchInc zero)
"succ zero"
ghci> churchPrint (churchInc one)
"succ succ zero"
ghci> churchPrint (churchInc two)
"succ succ succ zero"
```
§2.f Implement a binary function `churchAdd` which takes two Church numerals as arguments and returns their sum as a Church numeral. The function `churchAdd` is thus expected to behave in the following way:

```
ghci> churchPrint (churchAdd zero one)
"succ zero"
ghci> churchPrint (churchAdd one two)
"succ succ succ zero"
ghci> churchPrint (churchAdd two three)
"succ succ succ succ succ zero"
```

§2.g More difficult: implement a binary function `churchMult` which takes two Church numerals as arguments and returns their product as a Church numeral. The function `churchMult` is thus expected to behave in the following way:

```
ghci> churchPrint (churchMult zero one)
"zero"
ghci> churchPrint (churchMult one two)
"succ succ zero"
ghci> churchPrint (churchMult two three)
"succ succ succ succ succ succ zero"
```

§2.h Even more difficult: implement a binary function `churchPower` which takes two Church numerals `m` and `n` as arguments and returns their exponential `m^n` as a Church numeral. The function `churchPower` is thus expected to behave in the following way:

```
ghci> churchDecode (churchPower two three)
8
ghci> churchDecode (churchPower three two)
9
```

§2.i Define a `churchCompare` function which takes two Church numerals `m` and `n` as arguments and returns the following values:

- GT when `m` is strictly larger than `n`
- LT when `m` is strictly smaller than `n`
- EQ when `m` and `n` are equal.

§2.j Use the `churchCompare` function in order to define a `churchSort` function which sorts every list of Church numerals. The `churchSort` function is thus expected to have the following behavior:

```
ghci> churchSort [two,one,zero,three]
[zero,one,three,two]
```