HW 2 **Prof. Caldwell**
Due: 30 August 2011 **COSC 3015**

1

Problem 1.1. Read chapter 1 of Bird – since the book is not in yet, I have linked to a pdf copy of that chapter on the course web-page.

2

Consider the following code.

```
plus :: (Integer, Integer) -> Integer
plus (x,y) = x + y

cplus :: Integer -> Integer -> Integer
cplus x y = x + y
```

The function `plus` takes its arguments all at once packaged in a pair while `cplus` takes its arguments one at a time.

In class, we discussed how Haskell supports a notation for describing a function without forcing you to choose a name for it.

The general form is

$$\backslash x \;\to\; e$$

where x is a variable and e is a Haskell expression.

Note that in Haskell "$\to$" is used to denote the *type constructor* for functions (*e.g.* if γ and δ are types, then $\gamma \to \delta$ is the type of functions from γ to δ. Also, "$\to$" is used in the expression language to describe an actual function, ($\backslash x \to e$) denotes a function whose single argument is referred to in the expression e by the variable x.

This overloading of syntax is similar to that for Cartesian products. If γ and δ are types then (γ, δ) is the type whose elements are the pairs where the first element comes from γ and the second element comes from δ. But also, if $a \in \gamma$ and $b \in \delta$, then the pair $(a, b) \in (\gamma, \delta)$. So the developers of Haskell have used the same notation for the type constructor and to construct the elements of the type in both cases.

Now, consider the following interaction with the Haskell interpreter.

```
Main> :t cplus
cplus :: Integer -> Integer -> Integer
Main> :t cplus 7
cplus 7 :: Integer -> Integer
```

Evidently `cplus 7` is a function of type `Integer -> Integer`. But what function is it? It is the function that is expecting an input `y` and will compute the expression `7 + y`. So, it is the function described by the following expression:

```
\y -> 7 + y
```

This form of Haskell expression is called a *lambda-term* (λ-term).

We can write ($\backslash x\ y \rightarrow e$) for ($\backslash x \rightarrow (\backslash y \rightarrow e)$).

Every function can be written in a form where no arguments are declared on the left side of the definition. If e is an arbitrary Haskell expression, then the following examples show how this works.

$$\begin{array}{llllll} f\,x = e & \text{is the same as} & f = \backslash x \rightarrow e & & \\ g\,x\,y = e & \text{is the same as} & g\,x = \backslash y \rightarrow e & \text{is the same as} & g = \backslash x\,y \rightarrow e \end{array}$$

We say that `cplus` is in *Curried Form.* In this form of function definition, where the function takes its arguments one at a time, is named after Haskell Curry (1900-1982), an American mathematician and logician. As you might guess, the Haskell programming language is named after him as well. A Curried function takes its arguments one at a time – as opposed to packaged up in a pair.

Consider the following function[1] called `curry`:

```
curry f   =  \x -> \y -> f(x,y)
```

We can also move the arguments over to the left side of the "=" and use the entirely equivalence definition

```
curry f x y  = f(x,y)
```

In Haskell we have the following.

```
Main> :t curry
curry :: ((a,b) -> c) -> a -> b -> c
Main>
```

So `curry` takes a function of type `((a,b) -> c)` and returns a function of type `(a -> b -> c)`.

Also, consider the following definition:

```
uncurry f = \(x,y) ->  f x y
```

An equivalent definition that does not contain any lambda terms is given as follows:

```
uncurry f (x,y) = f x y
```

Here is the type of `uncurry`.

```
Main> :t uncurry
uncurry :: (a -> b -> c) -> (a,b) -> c
Main>
```

[1]Note that curry and uncurry are defined in the standard prelude so if you want to try your own version give it a name like `curry1` or `curry'`.

The function `curry` works on functions expecting two arguments packaged up as a pair and returns a function that expects arguments one at a time. The `uncurry` functions works on functions expecting two arguments. What are the analogues of curry and uncurry for functions of three arguments?

Problem 2.1. Write the function `curry3` having the following type:

```
Main> :t curry3
:t curry3
curry3 :: ((a,b,c) -> d) -> a -> b -> c -> d
Main>
```

Problem 2.2. Write the function `uncurry3` having the following type:

```
Main> :t uncurry3
:t uncurry3
uncurry3 :: (a -> b -> c -> d) -> (a,b,c) -> d
Main>
```

Problem 2.3. Test you code to get the results in the hw2_expected.txt output (linked onthe webpage.)