**HW 3** **Prof. Caldwell**
**Due:** 11 September 2011 **COSC 3015**

# 1

In class we discussed the following code.

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

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

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

We discussed how Haskell supports a notation for describing a function without forcing you to choose a name for it.

The general form is

$$\backslash x \;\; \rightarrow \;\; e$$

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

Note that in Haskell "$\rightarrow$" is used to denote the *type constructor* for functions (*e.g.* if $\gamma$ and $\delta$ are types, then $\gamma \rightarrow \delta$ is the type of functions from $\gamma$ to $\delta$. Also, "$\rightarrow$" is used in the expression language to describe an actual function, $(\backslash x \rightarrow e)$ 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 $\gamma$ and $\delta$ are types then $(\gamma, \delta)$ is the type whose elements are the pairs where the first element comes from $\gamma$ and the second element comes from $\delta$. 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> :type plusc
plusc :: Integer -> Integer -> Integer
Main> :type plusc 7
plusc 7 :: Integer -> Integer
```

Evidently `plusc 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* ($\lambda$-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.

$f\,x = e$ is the same as $f = \backslash x \rightarrow e$
$g\,x\,y = e$ is the same as $g\,x = \backslash y \rightarrow e$ is the same as $g = \backslash x\,y \rightarrow e$

We say that `plusc` is in *Curry 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. It takes its arguments one at a time – as opposed to packaged up in a pair.

Consider the following function (as defined in class):

```
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 p =  f (fst p) (snd p)
```

Or equivalently,

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

Here is the type of `uncurry`.

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

Now, these functions work on functions of 2 arguments. What about functions of three arguments?

**Problem 1.1.** Write the function `curry3` having the following type:

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

**Problem 1.2.** Write the function `uncurry3` having the following type:

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

HINT: You might want to use the second definition of uncurry as a model (where the pair is destructured in the argument). As an experiment you might try to evaluate the term `snd (1,''xxx'',[])`.