HW 8 **Prof. Caldwell**
Due: 26 September 2006 **COSC 3015**

1 Lambda Abstractions

We have talked about Haskell expressions of the form

$$(\backslash x\text{->}b)$$

which denote (nameless) functions of one argument. Expressions of this form are called λ-*abstractions* (read: lambda abstractions) or simply *abstractions.* Abstractions of this kind are syntactically well-formed if x is a variable, and b is an expression (which we call the *body*). The variable x is said to be *bound* in the body b of the abstraction. Mathematically, this term is typically written

$$\lambda x.e$$

The "\" in the Haskell notation is meant to suggest λ and the use of the symbols "–>" instead of "." for separating the name of the bound variable from the body of the definition is meant to suggest that expressions of this kind are functions. Recall that function types are constructed by taking two types (say α and β) and putting them together as $\alpha\text{->}\beta$.

⋆ ⋆ ⋆

Now, it turns out that, and we mentioned this in class more than once and in hw4, that definitions in the left and right columns mean exactly the same thing.

i.)	$f\ x\ =\ e$	$f\ =\ (\backslash x \text{ -> } e)$
ii.)	$f\ x\ y\ =\ e$	$f\ x =\ (\backslash y \text{ -> } e)$
iii.)	$f\ x\ y\ =\ e$	$f\ =\ (\backslash x \text{ ->}(\backslash y \text{ -> } e))$
iv.)	$(\backslash x \text{ ->}(\backslash y \text{ -> } e))$	$(\backslash x\ y \text{ -> } e)$

i.) The left side of (i.) is the definition of a function named f which takes one argument x and whose definition is given by the expression e. The right side says, f is defined to be the expression $(\backslash x \text{ -> } e)$, which is function of one argument named x and whose definition is given by the expression e.

ii.) The left side of (ii.) defines a function of two arguments x and y named f whose definition is given by the expression e. The right side defines a function named f of one argument x whose body is the (λ-abstraction) $(\backslash y \text{ -> } e)$ which is a function of one argument (named y here.)

iii.) The left side of (iii.) is as for (ii) – but the right side defines f to be an expression of the form $(\backslash x \text{ ->}e_1)$ where e_1 itself is an abstraction of the form $(\backslash y\text{->}e)$.

iv.) The left side is an expression containing nested abstractions and the right side shows that Haskell supports a nicer way of writing this down, you can just write down the variable names next to one another – but both expressions mean the same thing. Thus, the right side of (iii.) could have been more simply writen as $f = (\backslash x\ y \text{ -> } e)$.

Example 1.1. We have previously defined the compose function as follows:

```
compose f g x = f (g x)
```

From the discussion above, we can conclude that the following definition is entirely equivalent.

```
compose = (\f g x -> f (g x))
```

So, evidently, we don't even need to give it a name to use it in the interpreter. Instead of entering the following into the interpreter(after loading a Haskell script file containing the module called `Compose`):

```
Compose> compose (2+) (4*)
```

we could directly enter the following expression.

```
Hugs> (\f g x -> f (g x)) (2+) (4*)
```

Note: If you do not understand that the expression `(2+)` is a well-formed Haskell expression or if you don't know what function it is, you need to reread Bird pg. 14.

⋆ ⋆ ⋆

In Haskell, as in many other languages, to apply a function to an argument we just write them next to one another. So, the expression

$$f\ e$$

means, apply the function f to the argument given by the expression e. So evidently,

$$(\backslash x\text{->}b)\ e$$

means apply the function denoted by the expression $(\backslash x\text{->}e)$ to the argument e.

To do this, *i.e.* to actually perform the application of the function $(\backslash x\text{->}b)$ to the expression e – we replace all the x's in the expression b by e. We write $b[x := e]$ (read: substitute e for x in b) to mean[1], replace all the x's in b by the expression e.

We will write $e_1 \leadsto e_2$ if expression e_1 evaluates to e_2. Using this notation,

$$(\backslash x\text{->}b)\ e \ \leadsto\ b[x := e]$$

Example 1.2. Here are some examples:

`(\x -> x + 1) 5` $\leadsto$ `(x + 1)[x:=5]` which is equal to `(5 + 1)`

The following example shows that the name of the variable does not matter.

`(\y -> y + 1) 5` $\leadsto$ `(y + 1)[y:=5]` which is equal to `(5 + 1)`

Note that `(\x y -> x - y)` is the same as `(\x -> \y -> x - y)`.

`(\x -> \y -> x - y) 5` $\leadsto$ `(\y -> x - y)[x:=5]` which is equal to `(\y -> 5 - y)`

[1] There are some complications in the definition of substitution if there are abstractions in the expression b and there are name clashes of variables used to define abstractions and variable names in the expression e, but we will not worry about them here.

The following example shows that the order of the arguments might matter.

`(\y -> \x -> x - y) 5` $\rightsquigarrow$ `(\x -> x - y)[y:=5]` which is equal to `(\x -> x - 5)`

Here's what happens when more than one argument is applied.

`(\y -> \x -> x - y) 5 7` $\rightsquigarrow$ `((\x -> x - y)[y:=5]) 7` which is equal to `(\x -> x - 5) 7`
`(\x -> x - 5) 7` $\rightsquigarrow$ `(x - 5)[x:=7]` which is equal to `7 - 5`

Exercise 1.1. Rewrite the following function definitions so they are in the purely functional form (*i.e.* where the definition does not have anything left of the "=" except the function name.

a.) `constf x y = x`
b.) `substf g h x = g x (h x)`
c.) `applyf g x = g x`
d.) `flipf g x y  = g y x`

Enter you definitions into a Haskell script, and then evaluate each of them on at least three inputs.

Exercise 1.2. Rerun the examples you made up for Exercise 1.2 but without using any named functions. *e.g.* instead of:

```
Hw8> constf 5 7
5
Hw8>
```

Your interaction with Hugs should look something like:

```
Hugs> (\x y -> x) 5 7
5
Hugs>
```

(Depending on your setup – the prompt might read `Prelude>` instead of `Hugs>`.