HW 9 **Prof. Caldwell**
Due: 28 September 2006 **COSC 3015**

1 Disjoint Unions

Exercise 1.1. Reread section 2.5 of Bird.

The Hugs implementation of the the type `Either` differs from the older version of Haskell described in the book in that the following code is in the prelude.

```
data  Either a b  =  Left a | Right b   deriving (Eq, Ord, Read, Show)

either               :: (a -> c) -> (b -> c) -> Either a b -> c
either f g (Left x)  =  f x
either f g (Right y) =  g y
```

So, in Hugs, we use the destructor `either` rather than `case` as described in Bird. In class, we said:

```
case                :: ((a -> c), (b -> c)) -> (Either a b) -> c
case (f,g) (Left x)  =  f x
case (f,g) (Right y) =  g y
```

So, `either` is just like `case`, except it is fully curried.

2 The Unit Type

There is a type of tuples of length zero written `()`. There are two inhabitants of theis type, $\perp$ (which inhabits every type) and the element `()`. This can be a bit confusing since, for the unit type, the element of the unit type has the same syntax as the type.

Now, we can make a constant into a function by making it a function of the unit type.

```
pifun :: () -> Float
pifun () = 3.14159
```

In the signature for `pifun`, "`()`" denotes the unit type. In the definition of `pifun`, "`()`" denotes the single element of the unit type.

3 Another encoding of Booleans using Disjoint Unions

Now, conside the following module.

```
module Boolean where

  type Boolean = Either () ()

  true, false :: Boolean
  true = Left ()
  false = Right ()

  ifthenelse :: Boolean -> a -> a  -> a
  ifthenelse b e1 e2 = either (\() -> e1) (\() -> e2) b
```

Here, the type `Boolean` is defined as a disjoint union of the single element unit type. So, there are four elements,

`Boolean` = {`Left()`, `Right()`, $\perp$, `Left` $\perp$, `Right` $\perp$}

The proper values of the type are `Left ()` and `Right ()`.

Now, we define constants `true` and `false` to be `Left ()` and `Right ()` respectively.

The definition of `ifthenelse` is somewhat interesting. The function `either` is expecting functions as its first and seconf arguments. We want `ifthenelse` to just evaluate to the expression `e1` if the condition `b == true` (recall `true` is defined to be `Left()`) and to evaluate to `e2` if the conditon `b==false` (where `false` has been defined as `Right()`). We wrap `e1` and `e2` up as lambda expressions whose argument is the constant `()`]emi.e. the single element of unit. This may seem odd – but the interpreter is using pattern matching – and since the only that that can be unwrapped by *either* if the argument `b` is `true` or `false` is `()`, these functions `(\() -> e1)` and `(\() -> e2)` will only ever be applied to the constant `()`.[1]

Exercise 3.1. Using the definition of `ifthenelse, true` and `false` given, prove that

```
ifthenelse true e1 e2 = e1
ifthenelse false e1 e2 = e2
```

Exercise 3.2. Using definitions of `ifthenelse, true` and `false` given above, extend the code to include definitions of `and` (`/\`), or (`\/`), and implies (`.=>`).

```
(/\),(\/),(.=>) :: Boolean -> Boolean -> Boolean
```

You may use the destructor `either` directly if you like, or define the functions in terms of `ifthenelse`. There is a link on the hw page to `hw9_expected.txt` which contains an example test run of your code showing what the results should look like.

[1] As an experiment, you might try typing in the following expressions to the interpreter: `(\1 -> 2)1` and `(\3 -> 2)1` to see what Haskell does.