CIS 670, Homework 5
Due 20 Nov 2002

1. Using type classes in Haskell, implement a version of J or type-safe cast. In other words, define the following type class:

```
class Cast a where
   cast :: Cast b => a -> b
```

   and instances for integers, products and function types.

```
instance Cast Int where
   cast = ...
instance (Cast a, Cast b) => Cast (a,b) where
   cast = ...
instance (Cast a, Cast b) => Cast (a->b) where
   cast = ...
```

   In order to write this function, you'll have to add some additional operations to the `Cast` type class. If the types `a` and `b` don't match, feel free to have cast go into an infinite loop (instead of returning zero). If that bothers you, you can rewrite cast to be of type `Cast b => Maybe(a -> b)`.

   The answer to this exercise can be found in the paper "Type-Safe Cast" from my homepage.

2. A second way to implement type-safe cast in Haskell is to encode λ_R. Fill in the blanks for the following implementation of *typecase*. (It's also possible to encode *typerec*, but as Haskell has built-in recursion *typecase* is a little easier to work with.)

```
newtype R a = ...
rint :: R Int
rint = ...
rarrow :: R a -> R b -> R (a -> b)
rarrow = ...
rprod :: R a -> R b -> R (a,b)
rprod = ...
data Branches c =
  B { int   :: c Int,
      arrow :: forall a b. R a -> R b -> c (a -> b),
      prod  :: forall a b. R a -> R b -> c (a,b)
    }
typecase :: R a -> Branches c -> c a
typecase = ...
```

   Because of the type constructor in the return type of `typecase`, getting it to typecheck in Haskell is a little tricky. You'll almost always need to use a newtype for this type constructor. For example:

```
newtype S a = S (a -> String)
tostring :: R a -> S a
tostring ra = typecase ra
    B { int   = S show,
        arrow = \r1 -> \r2 -> S (\x -> "function"),
        prod  = \r1 -> \r2 -> S (\(x1, x2) ->
                   "(" ++ tostring r1 x1 ++ "," ++ tostring r2 x2 ++ ")"
      }
```

Now implement `cast` using `typecase`.

3. With multi-parameter typeclasses and functional dependencies we can also implement a form of type-level type analysis in Haskell. The core idea is that we write the operation on types in "prolog" style, the second parameter of the typeclass is the result of a "Typerec" on the first argument.

   Using this idea, write the instances of swap for integers, booleans, products and function types. The functional dependency `a -> b` means that the parameter `b` is determined by the parameter `a`.

   ```
   class Swap a b | a -> b where
     swap   :: a -> b
     unswap :: b -> a
   ```

   such that `swap ((4::Int, 5::Int),True) = (True, (5,4))`

4. Write a few paragraphs comparing intensional type analysis and Haskell type classes, as if you were writing a related work section of a paper. Think about their differences in terms of (for example) expressiveness, ease of use, ease of implementation, and complexity of language design.