Problem Set #7

due November 5

Problem 1. Answer the following questions. You can answer just YES or NO and get full credit (if correct) though, of course, it's always good to explain your answer.

(1) Is this code type-correct in Hindley Milner?

```
h f =
  if (f True) then (f 1) else (f 0)
```

(2) Is this code type-correct in Hindley Milner?

```
(\x -> x x (x True)) (\y -> y)
```

(3) Is this term type correct in Hindley Milner?

```
h x =
  let
      m y = if (y==0) then True else (n y)
      n y = if (y==0) then False else (m y)
  in
      (n x, m x)
```

(4) How about in Simple types, is the term from (3) type-correct in Simple types?

Problem 2. Next, implement Simple types for the pure Lambda calculus with integer and boolean constants using on-the-fly typing (Strategy Two).

The main datatypes are Expressions representing Lambda calculus expressions:

```
data Prim =
    PNum Int
  | PBool Bool

data Exp =
    EVar Ident
  | ELambda Ident Exp
  | EApp Exp Exp
  | EPrim Prim
```

and Types representing Simple types:

```
data BaseType = BTInt | BTBool

data Type =
    TBase BaseType
  | TVar TVar
  | TArrow Type Type
```

`type TEnv = [(String, Type)]` is the type environment, a mapping from identifiers to types (Γ in lecture), and `type Subst = [(TVar, Type)]` is the substitution environment, a mapping from type variables to types.

(1) Write the function that applies a substitution to a type:

```
substType :: Type -> Subst -> Type
substType = throw ToImplement

> substType (TVar "__t0") [("__t0",TArrow (TBase BTInt) (TBase BTInt))]
TArrow (TBase BTInt) (TBase BTInt)
```

You will also code a function (last) that prints the type in a friendlier form:

```
"BTInt -> BTInt"
```

(2) Next, write a function that substitutes in the type environment:

```
substEnv :: TEnv -> Subst -> TEnv
substEnv = throw ToImplement

> substEnv [("x",TVar "__t0")] [("__t0",TArrow (TBase BTInt) (TBase BTInt))]
[("x",TArrow (TBase BTInt) (TBase BTInt))]
```

(3) Implement Robinson's unification algorithm:

```
unify :: Type -> Type -> Subt
unify = throw ToImplement

> unify (TArrow (TVar "__t0") (TVar "__t1")) (TArrow (TBase BTInt) (TBase BTInt))
[("__t0",TBase BTInt),("__t1",TBase BTInt)]
```

(4) Now, the main function, `inferTypes`:

```
inferTypes :: TEnv -> Integer -> Exp -> (Subst, Type, Integer)
inferTypes = throw ToImplement

> inferTypes [] 0 (ELambda "x" (EVar "x"))
([],TArrow (TVar "__t0") (TVar "__t0"),1)
```

The integer input is the next available fresh variable at entry of `inferTypes` and the integer output is the next available fresh variable at exit. Recall that as we infer types, we need to assign fresh type variables to identifiers and subexpressions.

(5) The final function is `canonicalize`, which renames type variables and pretty prints the type of the expression (mainly for ease of testing on Submitty):

```
canonicalize :: Exp -> String
canonicalize = throw ToImplement
```

```
> canonicalize sComb -- sComb is the S-combinator term from Quiz 3
"(t1 -> t2 -> t3) -> (t1 -> t2) -> t1 -> t3"
```

It works as follows: given a type, fold the type tree into a list and compute a "renaming" substitution in the order variables appear in the list. E.g., if the fold of the type tree is `["__t4","__t2","__t4"]` then the corresponding substitution is `[("__t4",TVar "t1"),("__t2",TVar "t2")]`.

Note: In this function, you will need to (1) call your `inferTypes` on the input expression to compute its type `t`, (2) fold `t` into a list, (3) compute the renaming substitution, (4) apply the substitution on `t`, and finally, (5) print the type nicely, keeping only necessary parentheses in function types.

Note: Download minimal starter code in files `Data.hs` and `Ps7.hs` and submit in Submitty.

Haskell Style Guide. Adapted from Stephanie Weirich (UPenn CIS 5520).

- **Write a type signature for every function.** We will be strict about this when grading. Hint: try writing the signature *before* writing the function. If you do write the function first, try deducing the signature and if this doesn't work, there is always the `:t` command.
- Make sure that your code produces no errors or warnings. Code with errors receives 0 on Submitty and we will mark down warnings during TA grading.

- Use consistent indentation.
- Do not use tab characters, use space for indentation. GHC should be flagging tabs, but nevertheless, be careful.
- No line should have more than **80 characters**.
- Use whitespace to make your code readable. Add whitespace on either side of binary operators, e.g., write `3 * n + 1` instead of `3*n+1`.

- Use descriptive names.
- Follow standard Haskell naming conversions: (1) use camelCase for compound names, and (2) use `x` and `xs` when you pattern-match lists.

- Use comments. Each function definition should be preceded by a comment.
- Comment should say what the function does, not how.
- Comments should be concise. Do not overcomment.
- Use full English sentences.

- Do not leave incomplete pattern matches. They will be marked down.
- Tuples, records and datatypes can be decomposed. You can also use the `@` operator if you need a reference to both the object and its components.
  For example, do not use this:

```
f arg1 arg2 = ...  where
   x = fst arg1
   y = snd arg1
   z = fst arg2
```

  Use this instead:

```
f (x,y) (z,_) = ...
```

- Combine nested case expressions.
  For example, do not use this:

```
case x of
      Red -> case y of
                Red  -> True
                Blue -> False
      Blue -> case y of
                Red  -> False
                Blue -> True
```

  Use this instead:

```
case (x,y) of
```

```
(Red,   Red) -> True
(Blue, Blue) -> True
(   _,    _) -> False
```

- Use library functions, unless the assignment explicitly forbids them. Use Haskell's search engine Hoogle to look up library functions.