CS 320—Homework One

Write your answers to the problems in the space indicated. Scan your solution and submit to Gradescope as a PDF file. You may do this from your phone using free scanning apps, or with a desktop scanner. Do NOT edit this file and move things around, as the format was designed for easier grading in Gradescope.

NOTE: You may be able to test some of your solutions to these using Haskell; but you will learn more if you try these first by hand, and remember that you won’t have your computer with you on the first midterm!

**Problem One:** Consider the following data declarations:

```plaintext
data A = B | C | D A

data E = E A | G E A | H
```

(A) Draw tree diagrams of the following four data expressions. The first one is done for you above and to the right.

(i)  (G H C)  
(ii)  (E (D C))  
(iii)  (G (G H B) (D B))  
(iv)  (G (G (E B) (D (D C))) C )

(B) For the following, cross out with a big X any expression which would not be legal (i.e., would be flagged in Haskell as having a type error if you typed it into the Haskell interpreter). Haskell pays attention to parentheses and you should too. You don’t have to explain any errors, just cross them out. The first one is done for you.

```
(A)  (G (E (D C))) (D A)  
(G (E (D C))) (D A)  
(G (E B) C)  
(G (E B) C)  
```

```
(B)  X  
X  
(E (D C))  
(D A)  
(G (E B) (D (D C)))  
(G (D H) (G H C))  
```
Problem Two: Consider the following data and function declarations (assume this is only a part of the program, and that it works correctly).

(A) Underline all occurrences of constructors in this program fragment.
(B) Circle all occurrences of variables (including names of functions, wildcards, and type variables)
(C) Put an asterisk (*) on each occurrence of a type name.

```haskell
data Either* a b = Left a | Right b

eq :: Either* Nat* Bool* -> Either* Nat* Bool* -> Bool*
eq (Left x) (Left y) = x == y
eq (Right x) (Right y) = x == y
eq _ _ = False

eqList :: List* (Either* Nat* Bool*) -> List* (Either* Nat* Bool*) -> Bool*

eqList Nil Nil = True
eqList (Cons x xs) (Cons y ys) = and (eq x y) (eqList xs ys)

Notice that type expressions consist only of type names, parentheses, and the arrow type constructor ->. Type names can, but in this example do not, occur in the function definitions themselves. Constructors can ONLY occur in the function definitions or in data expressions.
**Problem Three:** Consider the following declarations:

```
data Nat = Zero | Succ Nat
data List a = Nil | Cons a (List a)
data Expr = Val Nat | Plus Expr Expr | Times Expr Expr
```

For each of the following terms and patterns, give the bindings for the variables in the pattern which would match it to the term, AND give the type of each variable binding, OR state that such a matching substitution is not possible and describe where the conflict is that prevents matching.

(A) `term1 = (Cons Zero (Cons (Succ Zero) (Cons Zero Nil)))`

```
-- pattern1 = (Cons x (Cons (Succ Zero) y))
```

Solution: 
```
x = Zero :: Nat
y = Cons Zero Nil :: List Nat
```

(B) `term2 = Plus (Times (Val Zero) (Val (Succ Zero))) (Val Zero)`

```
-- pattern2 = (Plus (Times x (Val (Succ y))) (Val Zero))
```

Solution: 
```
x = Val Zero :: Expr
y = Zero :: Nat
```

(C) `term3 = (Plus (Val Zero) (Plus (Val Zero) (Val (Succ Zero)))))`

```
-- pattern3 = Plus (Val y) (Plus (Val (Succ Zero)) (Val x))
```

-- no match, underlined terms are different!

(D) `term4 = Cons (Cons (Cons Zero Nil) Nil) (Cons (Cons (Succ Zero) Nil) Nil)`

```
-- pattern4 = (Cons (Cons x y) (Cons z Nil))
```

Solution: 
```
x = Zero :: Nat
y = Nil :: List Nat
z = (Cons (Succ Zero) Nil) :: List Nat
```
Problem Four: Consider the following declarations:

```haskell
data Bool = F | T -- Same as hw1 but abbreviated
data Nat = Z | S Nat -- to make solutions easier to write.

not :: Bool -> Bool
not T = F
not F = T

if' :: Bool -> Nat -> Nat -> Nat -- same as cond in lecture
if' T x _ = x
if' F _ y = y

red :: Nat -> Nat
red Z = Z
red (S x) = red x

inf :: Nat -> Nat
inf x = inf x
```

Recall that evaluation of expressions proceeds by searching for a redex (a subexpression where some rule defining a function will match) and rewriting the subexpression (using the bindings generated by the match). Consider the following algorithm, which thinks of the expression as a tree (as in Problem One):

```
while the expression contains some function symbol:
  for each subexpression in a preorder search of the expression:
    for each rule in order (top to bottom) in the program:
      if the rule matches:
        then do one rewrite and start over at the top of the while loop
```

For each of the following, show the expressions at each step, ending in the final value; if the sequence would not terminate, show a few steps and indicate that the rewrite sequence is infinite.

(A) if (not T) (red (S (S Z))) (if T Z (red Z))

```
=> if F (red (S (S Z))) (if F Z (red Z))
=> (if F Z (red Z))
=> (red Z)
=> Z
```

(B) if (not F) (red (S Z)) (red (inf Z))

```
=> if T (red (S Z)) (red (inf Z))
=> (red (S Z))
=> (red Z)
=> Z
```
NOW, for the next two problems, I want you to do the exact same expressions, but in the rewriting algorithm, search for the redex in a postorder (bottom-up, left to right) traversal of the expression (considered as a tree).

(C) \[
\begin{align*}
\text{if} & \ (\not T) \ (\text{red} \ (S \ (S \ Z))) \ (\text{if} \ T \ Z \ (\text{red} \ Z)) \\
\Rightarrow & \ \text{if} \ F \ (\text{red} \ (S \ (S \ Z))) \ (\text{if} \ F \ Z \ (\text{red} \ Z)) \\
\Rightarrow & \ \text{if} \ F \ (\text{red} \ (S \ Z)) \ (\text{if} \ F \ Z \ (\text{red} \ Z)) \\
\Rightarrow & \ \text{if} \ F \ (\text{red} \ Z) \ (\text{if} \ F \ Z \ (\text{red} \ Z)) \\
\Rightarrow & \ \text{if} \ F \ Z \ (\text{if} \ F \ Z \ (\text{red} \ Z)) \\
\Rightarrow & \ \text{if} \ F \ Z \ (\text{if} \ F \ Z \ Z) \\
\Rightarrow & \ \text{if} \ F \ Z \ Z \\
\Rightarrow & \ Z
\end{align*}
\]

(D) \[
\begin{align*}
\text{if} & \ (\not F) \ (\text{red} \ (S \ Z)) \ (\text{red} \ (\text{inf} \ Z)) \\
\Rightarrow & \ \text{if} \ T \ (\text{red} \ (S \ Z)) \ (\text{red} \ (\text{inf} \ Z)) \\
\Rightarrow & \ \text{if} \ T \ (\text{red} \ Z) \ (\text{red} \ (\text{inf} \ Z)) \\
\Rightarrow & \ \text{if} \ T \ Z \ (\text{red} \ (\text{inf} \ Z)) \\
\Rightarrow & \ \text{if} \ T \ Z \ (\text{red} \ (\text{inf} \ Z)) \\
\Rightarrow & \ \text{if} \ T \ Z \ (\text{red} \ (\text{inf} \ Z)) \quad \text{INFINITE REWRITE SEQUENCE!}
\end{align*}
\]

(E) **Short answer:** What was the difference between preorder (top down) and postorder (bottom up) in these examples? Be sure to comment on what happened in A compared with C, and B compared with D. Which seems to be a better strategy overall?

**Solution:** The top-down algorithm is clearly more efficient in that it uses fewer steps in comparing A and C; it also produces an answer in B, whereas it just runs forever in D. Seems like preorder is better in both cases!