FOPL - Homework 1

昂伟 PB11011058

Problem 4

(a) prob4.hs

```
data Tree a = Leaf a | Node (Tree a) (Tree a) deriving Show
maptree f (Leaf x) = Leaf (f x)
maptree f (Node x y) =  Node (maptree f x) (maptree f y)
```

(b) Using pattern match to define function. When *maptree* function's second argument matches "Leaf x", *Leaf (f x)* is evaluated; When function's second argument matches "Node x y", *Node (maptree f x) (maptree f y)* is evaluated, which calls *maptree* recursively.

(c) Function type: *maptree :: (t -> a) -> Tree t -> Tree a*. Because if *x* is of type *t*, there is no guarantee that *f(x)* is also of type *t*, so *f(x)*'s type is represented by *a*.

Problem 5

(a) prob5.hs

```
data Tree a = Leaf a | Node (Tree a) (Tree a)
reduce :: (a -> a -> a) -> Tree a -> a
reduce oper (Leaf x) = x
reduce oper (Node x y) = oper (reduce oper x) (reduce oper y)
```

(b) Using pattern match method to define function. When function's second arguement matches "Leaf x", result is *x*; Otherwise, *oper (reduce oper x) (reduce oper y)* is evaluated, which calls *reduce* recursively.

Problem 6

(a) prob6.hs

```
curry :: ((a, b) -> c) -> (a -> b -> c)
```

```
curry f a b = f (a, b)
uncurry :: (a -> (b -> c)) -> ((a, b) -> c)
uncurry f (a, b) = f a b
```

(b) let **f' = curry f**, so **f'** is of type *(a -> (b -> c))*. Then **f'** is taken as the first argument of function *uncurry*, so **uncurry f'** is of type *((a, b) -> c)*, which is the same type of **f**.
let **g' = uncurry g**, so **g'** is of type *((a, b) -> c)*. Then **g'** is taken as the first argument of function **curry**, so **curry g'** if of type of (a -> (b -> c)), which is the same type of **g**.

Problem 7

(a) prob7.hs

```
evens :: [Int]
evens = [x * 2| x <- [1..]]
odds :: [Int]
odds = [ x * 2 - 1 | x <- [1..]]
```

(b) prob7.hs

```
mergeAux [] [] l = l
mergeAux x [] l = l ++ x
mergeAux [] x l = l ++ x
mergeAux x1@(h1:t1) x2@(h2:t2) h3 =
    if h1 < h2
        then mergeAux t1 x2 (h3 ++ [h1])
    else mergeAux x1 t2 (h3 ++ [h2])

merge :: [Int] -> [Int] -> [Int]
merge l1 l2 = mergeAux l1 l2 []
```

(c) No, because *evens* and *odds* are both infinite lists. *merge* function will call itself infinitely.