```
remove :: (Eq a) => a -> [a] -> [a]
remove x = filter (\v -> v/=x)


freevars :: Lexp -> [String]
freevars (Atom s)            = [s]
freevars (Lambda v e)        = remove v (freevars e)
freevars (Apply e1 e2)       = (freevars e1)++(freevars e2)


alpha :: Lexp -> [String] -> Lexp
alpha exp@(Atom s) freeVars
    | elem s freeVars = Atom(s ++ "0")
    | otherwise = Atom s
alpha exp@(Lambda s e) freeVars
    | elem s freeVars = Lambda newS (alpha e freeVars)
    | otherwise = Lambda s (alpha e freeVars)
    where newS = s ++ "0"
alpha exp@(Apply e1 e2) freeVars = Apply (alpha e1 freeVars) (alpha e2 freeVars)


beta :: String -> Lexp -> Lexp -> Lexp -> Lexp
beta x e@(Atom tmp) m lexp
    | x == tmp = m
    | otherwise = e
beta x e@(Lambda tmp1 tmp2) m lexp
    | elem tmp1 (freevars lexp) = Lambda newtmp1 (beta x newtmp2 m lexp)
    | otherwise = Lambda tmp1 (beta x tmp2 m lexp)
    where newtmp1 = tmp1 ++ "0"
          newtmp2 = alpha tmp2 (freevars lexp)
beta x e@(Apply tmp1 tmp2) m lexp = Apply (beta x tmp1 m lexp) (beta x tmp2 m lexp)


eta :: String -> Lexp -> Lexp -> Lexp -> Lexp
eta x e m@(Atom tmp) lexp
    | x == tmp && notElem tmp (freevars e) = e
    | otherwise = lexp
eta x e _ lexp = lexp


reducer :: Lexp -> Lexp
```

```
reducer (Atom s) = Atom s
reducer exp@(Apply (Lambda s e) v)
    | exp == reduced = exp
    | otherwise = reducer reduced
    where reduced = beta s e v exp

reducer exp@(Apply e1 e2)
    | exp == reduced = exp
    | otherwise = reducer reduced
    where reduced = Apply (reducer e1) (reducer e2)

reducer exp@(Lambda s exp2@(Apply (Lambda s1 e) v))
    | exp == reduced = exp
    | otherwise = reducer reduced
    where reduced = Lambda s (beta s1 e v exp2)

reducer exp@(Lambda s (Apply e1 e2))
    | exp == reduced = exp
    | otherwise = reducer reduced
    where reduced = eta s e1 e2 exp
reducer exp@(Lambda s e)
    | exp == reduced = exp
    | otherwise = reducer reduced
    where reduced = Lambda s (reducer e)
```