Lambda Calculus with Extensions

Note: This is a Literate Haskell file. Embedded code blocks (lines starting with >) contain code that will be executed when compiled with GHC, or loaded with GHCi.

```
type Variable = String
```

Our original Lambda expressions: variables, lambda abstractions, and applications.

```
data Lambda = Var Variable
            | Lam Variable Lambda
            | App Lambda Lambda
```

Let us add some extensions from our previous example languages.

A conditional:

```
            | If Lambda Lambda Lambda
```

Some operations on numbers:

```
            | Add Lambda Lambda
            | Sub Lambda Lambda
```

Some operaions on booleans:

```
            | And Lambda Lambda
            | Not Lambda
            | Leq Lambda Lambda
```

Values are either booleans or integers.

```
            | Val Value
            deriving (Show)

data Value = Integer Integer
           | Boolean Bool
           deriving (Show)
```

We keep our notion of beta reduction. And for the core lambda-calculus the notion or Call-by-name reduction is the same.

```
-- |β-reduce a lambda expression
beta :: Lambda -> Maybe Lambda
beta (App (Lam x e1) e2) = Just $ subst x e2 e1
beta _ = Nothing
```

```
stepCBN :: Lambda -> Maybe Lambda
stepCBN (Var _) = Nothing
stepCBN e@(App (Lam _ _) _) = beta e
stepCBN (App e1 e2) =
    case stepCBN e1 of
         Just e1' -> Just (App e1' e2)
         Nothing ->
             case stepCBN e2 of
                  Just e2' -> Just (App e1 e2')
                  Nothing -> Nothing
stepCBN (Lam x e) = Nothing
```

Delta reduction

Let's now implement the extensions. Just like with `App`, we can separate expressions with the additional constructors into redexes and non-redexes, except instead of β-redexes, we customarily call them δ-redexes. For a conditional, a redex is when we have a Boolean value in the first position. Just like with `beta`, we can delegate δ-reduction to a separate helper function.

```
stepCBN e@(If (Val (Boolean _)) _ _) = delta e
```

If we don't have an "If redex", we can try reducing the first argument.

```
stepCBN e@(If e1 e2 e3) = do
    e1' <- stepCBN e1
    return $ If e1' e2 e3
```

Arithmetic operations need both of their arguments to be integer values, only then do we have a redex.

```
stepCBN e@(Add (Val (Integer _)) (Val (Integer _))) = delta e
stepCBN (Add e1 e2) =
    case stepCBN e1 of
         Just e1' -> return (Add e1' e2)
         Nothing -> case stepCBN e2 of
                         Just e2' -> return (Add e1 e2')
                         Nothing -> Nothing
stepCBN e@(Sub (Val (Integer _)) (Val (Integer _))) = delta e
stepCBN (Sub e1 e2) =
    case stepCBN e1 of
         Just e1' -> return (Sub e1' e2)
         Nothing -> case stepCBN e2 of
                         Just e2' -> Just (Sub e1 e2')
                         Nothing -> Nothing
```

Short-circuiting And is a redex when the first operand is a boolean value.

```
stepCBN e@(And (Val (Boolean _)) _) = delta e
stepCBN (And e1 e2) = do
    e1' <- stepCBN e1
    return $ Add e1' e2
```

The rest of the extra operations is straightforward:

```haskell
stepCBN e@(Not (Val (Boolean _))) = delta e
stepCBN (Not e) = do
    e' <- stepCBN e
    return $ Not e'
stepCBN e@(Leq (Val (Integer _)) (Val (Integer _))) = delta e
stepCBN (Leq e1 e2) =
  case stepCBN e1 of
       Just e1' -> return (Leq e1' e2)
       Nothing -> case stepCBN e2 of
                       Just e2' -> return (Leq e1 e2')
                       Nothing -> Nothing
```

Finally, values cannot be reduced.

```haskell
stepCBN (Val v) = Nothing
```

Now, we need to complete the delta reduction rules.

```haskell
delta :: Lambda -> Maybe Lambda
delta (If (Val (Boolean b)) e2 e3) | b = Just e2
                                   | not b = Just e3
delta (Add (Val (Integer i1)) (Val (Integer i2))) = Just $ Val $ Integer $ i1 + i2
delta (Sub (Val (Integer i1)) (Val (Integer i2))) = Just $ Val $ Integer $ i1 - i2
delta (And (Val (Boolean b)) e2) | b = Just e2
                                 | not b = Just $ Val $ Boolean False
delta (Not (Val (Boolean b))) | b = Just $ Val $ Boolean False
                              | not b = Just $ Val $ Boolean True
delta (Leq (Val (Integer i1)) (Val (Integer i2))) = Just $ Val $ Boolean $ i1 <= i2
delta _ = Nothing
```

The Call-by-value version of our stepper only differs in the App case. If we have a redex, we attempt to reduce the argument before we apply β-reduction.

```haskell
stepCBV :: Lambda -> Maybe Lambda
stepCBV (Var _) = Nothing
stepCBV (App (Lam x e1) e2) =
    case stepCBV e2 of
         Just e2' -> Just (App (Lam x e1) e2')
         Nothing -> beta (App (Lam x e1) e2)
stepCBV (App e1 e2) =
    case stepCBV e1 of
         Just e1' -> Just (App e1' e2)
         Nothing ->
             case stepCBV e2 of
                  Just e2' -> Just (App e1 e2')
                  Nothing -> Nothing
stepCBV (Lam _ _) = Nothing
```

The new constructs are handled in the same way as with call-by-name.

```haskell
stepCBV e@(Add (Val (Integer _)) (Val (Integer _))) = delta e
stepCBV (Add e1 e2) =
    case stepCBV e1 of
         Just e1' -> return (Add e1' e2)
```

```haskell
         Nothing -> case stepCBV e2 of
                         Just e2' -> return (Add e1 e2')
                         Nothing -> Nothing
stepCBV e@(Sub (Val (Integer _)) (Val (Integer _))) = delta e
stepCBV (Sub e1 e2) =
    case stepCBV e1 of
         Just e1' -> return (Sub e1' e2)
         Nothing -> case stepCBV e2 of
                         Just e2' -> Just (Sub e1 e2')
                         Nothing -> Nothing
stepCBV e@(And (Val (Boolean _)) _) = delta e
stepCBV (And e1 e2) = do
    e1' <- stepCBV e1
    return $ Add e1' e2
stepCBV e@(Not (Val (Boolean _))) = delta e
stepCBV (Not e) = do
    e' <- stepCBV e
    return $ Not e'
stepCBV e@(Leq (Val (Integer _)) (Val (Integer _))) = delta e
stepCBV (Leq e1 e2) =
  case stepCBV e1 of
       Just e1' -> return (Leq e1' e2)
       Nothing -> case stepCBV e2 of
                       Just e2' -> return (Leq e1 e2')
                       Nothing -> Nothing
stepCBV (Val v) = Nothing
```

As we add extensions, we need to update helper functions that recur on the structure of thelambda expression.

```haskell
-- |Substitution with variable renaming
subst :: Variable -> Lambda -> Lambda -> Lambda
subst x s t@(Var y) | x == y = s
                    | otherwise = t
subst x s t@(Lam y t') | x == y = t
                       | otherwise = Lam y' (subst x s (subst y (Var y') t'))
  where
    y' = makeFresh y [Var x, s, t']
subst x s (App e1 e2) = App (subst x s e1) (subst x s e2)
subst x s (If e1 e2 e3) = If (subst x s e1) (subst x s e2) (subst x s e3)
subst x s (Add e1 e2) = Add (subst x s e1) (subst x s e2)
subst x s (Sub e1 e2) = Sub (subst x s e1) (subst x s e2)
subst x s (And e1 e2) = And (subst x s e1) (subst x s e2)
subst x s (Not e) = Not (subst x s e)
subst x s (Leq e1 e2) = Leq (subst x s e1) (subst x s e2)
subst x s e@(Val _) = e
```

```haskell
-- |Collect the free variables of a lambda expression
freeVars :: Lambda -> [Variable]
freeVars (Var x) = [x]
freeVars (Lam x e) = delete x $ freeVars e
freeVars (App e1 e2) = freeVars e1 `union` freeVars e2
freeVars (If e1 e2 e3) = freeVars e1 `union` freeVars e2 `union` freeVars e3
```

```haskell
freeVars (Add e1 e2) = freeVars e1 `union` freeVars e2
freeVars (Sub e1 e2) = freeVars e1 `union` freeVars e2
freeVars (And e1 e2) = freeVars e1 `union` freeVars e2
freeVars (Not e) = freeVars e
freeVars (Leq e1 e2) = freeVars e1 `union` freeVars e2
freeVars (Val _) = []
```

```haskell
-- |Generate a new variable name that's fresh for the given set of expressions
makeFresh :: Variable -> [Lambda] -> Variable
makeFresh x es | x `notElem` fv = x
               | otherwise = findFresh 0
  where
    findFresh n =
        let x'' = x' ++ show n
        in if x'' `elem` fv
              then findFresh (n + 1)
              else x''
    fv = foldr (\e xs -> freeVars e `union` xs) [] es
    x' = stripNumericSuffix x

-- |Strip a numeric suffix of a variable name
stripNumericSuffix :: Variable -> Variable
stripNumericSuffix = reverse . dropWhile isDigit . reverse
  where
      isDigit x = x `elem` "0123456789"
```

```haskell
{-

 e1 --> e1'    ::    step?? e1 = Just e1'
 e1 -->* e1'   ::    either e1 --> e1'' and e1'' -->* e1'
                     otherwise e1 = e1' (transitive reflexive closure of -->)
-}
```