Recursive Evaluators for Lambda Calculus

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.

Substitution-based evaluators

Step-wise reduction is not the only way to evaluate lambda terms. We can write a recursive evaluator which fully evaluates a lambda term to a *value*. This means that we need to define a datatype for values for representing results of `eval`. This type will contain integers and booleans as before.

```
data Value = Integer Integer
           | Boolean Bool
```

Since λ-abstractions are values themselves, we need a representation for them.

```
           | LamV Variable (Lambda Value)
           deriving (Show)

type Variable = String
```

The rest of our abstract syntax remains the same, but we leave the value type as a parameter of `Lambda`, so we can use different representations.

```
data Lambda a = Val a
              | Var Variable
              | Lam Variable (Lambda a)
              | App (Lambda a) (Lambda a)
              | Add (Lambda a) (Lambda a)
              | Sub (Lambda a) (Lambda a)
              | Leq (Lambda a) (Lambda a)
              | If (Lambda a) (Lambda a) (Lambda a)
              deriving (Show)
```

First, let's implement a call-by-name evaluator using substitution.

```
evalCBN :: Lambda Value -> Maybe Value
```

Values evaluate to themselves.

```
evalCBN (Val v) = Just v
```

Like before, encountering a variable means an error: all variables should be substituted in a well-defined program.

```
evalCBN (Var x) = Nothing
```

λ-abstractions evaluate to the corresponding value.

```
evalCBN (Lam x e) = Just (LamV x e)
```

Evaluating an application means first evaluating the left-hand side fully, expecting a λ-abstraction value.

```
evalCBN (App e1 e2) = do
    LamV x e <- evalCBN e1
```

Call-by-name semantics means we substitute the unevaluated argument expression into the body of the abstraction.

```
    evalCBN $ subst x e2 e
```

That's it for the core λ-calculus fragment. The remainder of the evaluator deals with extensions, which are evaluated as previously.

```
evalCBN (Add e1 e2) = do
    Integer v1 <- evalCBN e1
    Integer v2 <- evalCBN e2
    return $ Integer (v1 + v2)
evalCBN (Sub e1 e2) = do
    Integer v1 <- evalCBN e1
    Integer v2 <- evalCBN e2
    return $ Integer (v1 - v2)
evalCBN (Leq e1 e2) = do
    Integer v1 <- evalCBN e1
    Integer v2 <- evalCBN e2
    return $ Boolean (v1 <= v2)
evalCBN (If e1 e2 e3) = do
    Boolean b <- evalCBN e1
    if b
        then evalCBN e2
        else evalCBN e3
```

Call-by-value evaluation only differs in the `App` case.

```
evalCBV :: Lambda Value -> Maybe Value
evalCBV (Val v) = Just v
evalCBV (Var x) = Nothing
evalCBV (Lam x e) = Just (LamV x e)
```

For application, call-by-value semantics dictates that we first evaluate the argument to a value, before substituting it into the body of the λ-abstraction.

```
evalCBV (App e1 e2) = do
    LamV x e <- evalCBV e1
    v2 <- evalCBV e2
    evalCBV $ subst x (Val v2) e
```

As before, evaluating the remaining constructs is the same.

```
evalCBV (Add e1 e2) = do
    Integer v1 <- evalCBV e1
    Integer v2 <- evalCBV e2
```

```haskell
    return $ Integer (v1 + v2)
evalCBV (Sub e1 e2) = do
    Integer v1 <- evalCBV e1
    Integer v2 <- evalCBV e2
    return $ Integer (v1 - v2)
evalCBV (Leq e1 e2) = do
    Integer v1 <- evalCBV e1
    Integer v2 <- evalCBV e2
    return $ Boolean (v1 <= v2)
evalCBV (If e1 e2 e3) = do
    Boolean b <- evalCBV e1
    if b
        then evalCBV e2
        else evalCBV e3
```

Environment-based evaluators

```haskell
type Env a = Map Variable a
```

We can also write recursive evaluators using environments. We will run into some complications. These are related to the handling of λ-abstractions. If we keep the same approach we used with the substitution-based evaluator, we run into a subtle problem:

```haskell
evalCBV' :: Env Value -> Lambda Value -> Maybe Value
evalCBV' _   (Val v) = Just v
evalCBV' env (Var x) = get env x
evalCBV' env (Lam x e) = Just $ LamV x e
evalCBV' env (App e1 e2) = do
    LamV x e <- evalCBV' env e1
    v2 <- evalCBV' env e2
    evalCBV' (set env x v2) e
```

Let's use this evaluator to evaluate ((λx. λy. x) 1 2), which we should expect to return 1:

```
evalCBV' empty (App (App (Lam "x" (Lam "y" (Var "x"))) (Val $ Integer 1)) (Val $ Integer 2))
=> Nothing
```

Why do we not get a result? Let's see what happens with the inner application, which gets reduced first:

```
evalCBV' empty (App (Lam "x" (Lam "y" (Var "x"))) (Val $ Integer 1))
=> Just (LamV "y" (Var "x"))
```

Where did the `Integer 1` disappear? Using our previous substitution-based evaluator, we can see that, when the outermost lambda is applied, 1 should be substituted inside of the partially applied lambda:

```
evalCBV (App (Lam "x" (Lam "y" (Var "x"))) (Val $ Integer 1))
=> Just (LamV "y" (Val (Integer 1)))
```

But in our environment-based version this information is absent! The 1 is just gone. That's because the call sequence of eval in this case is as follows:

```
evalCBV' empty (App (Lam "x" (Lam "y" (Var "x"))) (Val $ Integer 1))
  | evalCBV' empty (Lam "x" (Lam "y" (Var "x")))
  | => Just $ LamV "x" (Lam "y" (Var "x"))
```

```
  | evalCBV' empty (Val $ Integer 1)
  | => Just $ Integer 1
  | evalCBV' (set empty "x" $ Integer 1) (Lam "y" (Var "x"))
  | => Just $ LamV "y" (Var "x")
=> Just $ LamV "y" (Var "x")
```

As we can see, while we evaluate the inner lambda (λy. x) with an environment containing a binding for `"x"`, it doesn't really go anywhere: the evaluation of `Lam "y" (Var "x")` simply ignores its environment and returns `LamV "y" (Var "x")`. Then when we are evaluating evaluating the outer application, the environment where "x" should be found is actually the top-level, empty environment. Here's the full call tree.

```
evalCBV' empty (App (App (Lam "x" (Lam "y" (Var "x"))) (Val $ Integer 1)) (Val $ Integer 2))
  | evalCBV' empty (App (Lam "x" (Lam "y" (Var "x"))) (Val $ Integer 1))
      | evalCBV' empty (Lam "x" (Lam "y" (Var "x")))
      | => Just $ LamV "x" (Lam "y" (Var "x"))
      | evalCBV' empty (Val $ Integer 1)
      | => Just $ Integer 1
      | evalCBV' (set empty "x" $ Integer 1) (Lam "y" (Var "x"))
      | => Just $ LamV "y" (Var "x")
  | => Just $ LamV "y" (Var "x")
  | evalCBV' empty (Val $ Integer 2)
  | => Just $ Integer 2
  | evalCBV' (set empty "y" $ Integer 2) (Var "x")
  | => Nothing
=> Nothing
```

We need a way to "save" the environment that was used to evaluate the lambda, so it can be used when evaluating the body of the lambda in the application. This is what *closures* are for. They package a lambda with the environment that was in active when the lambda was being evaluated. Let us replace `LamV` with a closure. We'll name this datatype `ValueV`, where V represents call-by-Value.

```
data ValueV = IntegerV Integer
            | BooleanV Bool
            | ClosureV Variable (Lambda ValueV) (Env ValueV)
            deriving (Show)
```

Now let's try again.

```
evalCBV' :: Env ValueV -> Lambda ValueV -> Maybe ValueV
evalCBV' _   (Val v) = Just v
evalCBV' env (Var x) = get env x
```

In the lambda case, we use `ClosureV` to capture the current environment and package it up with the lambda argument and body.

```
evalCBV' env (Lam x e) = Just $ ClosureV x e env
```

In the application case, we use the environment *from the closure* to evaluate the body.

```
evalCBV' env (App e1 e2) = do
    ClosureV x e env' <- evalCBV' env e1
    v2 <- evalCBV' env e2
    evalCBV' (set env' x v2) e
```

Now we can evaluate our example expression:

```
evalCBV' empty (App (App (Lam "x" (Lam "y" (Var "x"))) (Val $ IntegerV 1)) (Val
```

```
$ IntegerV 2))
=> Just (IntegerV 1)
```

Works! Let's observe what we get as the result of the partial application to `IntegerV 1`:

```
evalCBV' empty (App (Lam "x" (Lam "y" (Var "x"))) (Val $ IntegerV 1))
=> Just (ClosureV "y" (Var "x") [("x",IntegerV 1)])
```

As we can see, the closure "remembers" what x is bound to, so the variable can be looked up properly when needed. Here is the rest of the evaluator, which is pretty-much the same as our previous evaluators:

```
evalCBV' env (Add e1 e2) = do
    IntegerV v1 <- evalCBV' env e1
    IntegerV v2 <- evalCBV' env e2
    return $ IntegerV (v1 + v2)
evalCBV' env (Sub e1 e2) = do
    IntegerV v1 <- evalCBV' env e1
    IntegerV v2 <- evalCBV' env e2
    return $ IntegerV (v1 - v2)
evalCBV' env (Leq e1 e2) = do
    IntegerV v1 <- evalCBV' env e1
    IntegerV v2 <- evalCBV' env e2
    return $ BooleanV (v1 <= v2)
evalCBV' env (If e1 e2 e3) = do
    BooleanV b <- evalCBV' env e1
    if b
        then evalCBV' env e2
        else evalCBV' env e3
```

To implement call-by-name, we need to modify our environment to store unevaluated expressions. However, just like with lambda bodies, these expressions might refer to free variables. Thus we introduce a type of *thunks* which are like closures without arguments: they bundle an expression with an environment. Because we are changing the environment, we also need to update the value type.

```
data Thunk = Thunk (Lambda ValueN) (Env Thunk)
           deriving (Show)

data ValueN = IntegerN Integer
            | BooleanN Bool
            | ClosureN Variable (Lambda ValueN) (Env Thunk)
            deriving (Show)

evalCBN' :: Env Thunk -> Lambda ValueN -> Maybe ValueN
evalCBN' _ (Val v) = Just v
```

Variable lookup now needs to evaluate the expression stored in a thunk with its environment:

```
evalCBN' env (Var x) = do
    Thunk e env' <- get env x
    evalCBN' env' e
evalCBN' env (Lam x e) = Just $ ClosureN x e env
```

Application does not evaluate its left-hand side, instead, it packages it up with the current environment as a thunk, which is then bound to the given argument.

```haskell
evalCBN' env (App e1 e2) = do
    ClosureN x e env' <- evalCBN' env e1
    evalCBN' (set env' x $ Thunk e2 env) e
evalCBN' env (Add e1 e2) = do
    IntegerN v1 <- evalCBN' env e1
    IntegerN v2 <- evalCBN' env e2
    return $ IntegerN (v1 + v2)
evalCBN' env (Sub e1 e2) = do
    IntegerN v1 <- evalCBN' env e1
    IntegerN v2 <- evalCBN' env e2
    return $ IntegerN (v1 - v2)
evalCBN' env (Leq e1 e2) = do
    IntegerN v1 <- evalCBN' env e1
    IntegerN v2 <- evalCBN' env e2
    return $ BooleanN (v1 <= v2)
evalCBN' env (If e1 e2 e3) = do
    BooleanN b <- evalCBN' env e1
    if b
        then evalCBN' env e2
        else evalCBN' env e3
```

Remaining details: Substitution

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

-- |Substitution with variable renaming
subst :: Variable -> Lambda a -> Lambda a -> Lambda a
subst x s (Val v) = Val v
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 (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 (Leq e1 e2) = Leq (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)
```

```haskell
-- |Generate a new variable name that's fresh for the given set of expressions
makeFresh :: Variable -> [Lambda a] -> 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"
```