□ Datatypes With Field Labels

```
data T = C1 {f1,f2 :: Int}
       | C2 {f1 :: Int,  f3,f4 :: Char}

C1 1 1 ≡ C1{f1 = 1, f2 = 1}
C1{f2 = 2, f1 = 3} ≡ (C1 1 2){f1 = 3} ≡ C1 3 2


f1 :: T -> Int
f2 :: T -> Int
f3 :: T -> Char
f4 :: T -> Char


f1 C1{f1 = 1, f2 = 2} ↪ 1

f3 C1{f1 = 1, f2 = 2} :: Char
f3 C1{f1 = 1, f2 = 2} ↪ match error
```

□ Higher-order Abstract Syntax

```
data Exp = Lam (Exp -> Exp) | App Exp Exp | OpenVar Int

normalize e = case e of
    Lam e -> Lam $ \x -> normalize $ e x
    App e1 e2 -> case normalize e1 of
        Lam e -> normalize $ e e2
        f -> App f $ normalize e2
    OpenVar i -> Openvar i

instance Show Exp where
    show e = fst $ go 0 e where
        go n (OpenVar i) =
            if i < 0 then ("p"++(show $ -i), n) else ("x"++(show i), n)
        go n (App e1 e2) =
            let (s1, n1) = go n e1
                (s2, n2) = go n1 e2
            in ("("++s1++" "++s2++")", n2)
        go n (Lam e) =
            let (s, n1) = go (n+1) (e $ OpenVar n)
            in ("\\x"++(show n)++" -> "++s++")", n1)
```

Note: `show` assumes open vars have negative index.

□ Purity

- Haskell is a pure language.
- Referential transparency
  - Value of expression unchanged when any subexpression is replaced by it's value.
  - Order of evaluation doesn't matter.
  - Equational Reasoning.
- Lazy Evaluation
- Naive programs are harder to modify than impure languages (e.g. ML).
- Careful structuring allows modular design and clean incorporation of impure features.

□ Simple Language

```
data Term = Var String
          | Con Int
          | Add Term Term
          | Lam String Term
          | App Term Term
  deriving Show

data Value = Wrong
           | Num Int
           | Fun (Value -> Value)

instance Show Value where
    show Wrong = "<wrong>"
    show (Num  n) = show n
    show (Fun _) = "<function>"
```

□ Interpreter

```
type Env = [(String, Value)]

interp :: Term -> Env -> Value
interp (Var x) env = getVar env x
interp (Con n) _ = Num n
interp (Add e1 e2) env = add (interp e1 env) (interp e2 env)
interp (Lam x e) env = Fun $ \v -> interp e ((x,v):env)
interp (App e1 e2) env = apply (interp e1 env) (interp e2 env)

getVar :: Env -> String -> Value
getVar ((x,v):xs) y = if x == y then v else getVar xs y
getVar [] _ = Wrong

add :: Value -> Value -> Value
add (Num m) (Num n) = Num $ m + n
add _ _ = Wrong

apply :: Value -> Value -> Value
apply (Fun v1) v2 = v1 v2
apply _ _ = Wrong
```

□ Error Messages

```
data Err a = Err String | Succ a

data Value = Wrong
           | Num Int
           | Fun (Value -> Err Value)

instance Show a => Show (Err a) where
    show (Err s) = "Error: "++s
    show (Succ v) = "Success: "++(show v)

getVar :: Env -> String -> Err Value
getVar ((x,v):xs) y = if x == y then Succ v else getVar xs y
getVar [] y = Err $ "unbound variable "++y

add :: Value -> Value -> Err Value
add (Num m) (Num n) = Succ $ Num $ m + n
add m n = Err $ "expected numbers, found "++(show m)++" and "++(show n)

apply :: Value -> Value -> Err Value
apply (Fun v1) v2 = v1 v2
apply f _ = Err $ "expected function, found "++(show f)
```

□ Error Reporting Interpreter

```
interp :: Term -> Env -> Err Value

interp (Var x) env = getVar env x

interp (Con n) _ = Succ $ Num n

interp (Add e1 e2) env = case interp e1 env of
    Err s -> Err s
    Succ v1 -> case interp e2 env of
        Err s -> Err s
        Succ v2 -> add v1 v2

interp (Lam x e) env = Succ $ Fun $ \v -> interp e ((x,v):env)

interp (App e1 e2) env = case interp e1 env of
    Err s -> Err s
    Succ v1 -> case interp e2 env of
        Err s -> Err s
        Succ v2 -> apply v1 v2
```

□ Abstract Common Functionality

```
unit :: Value -> Err Value
unit v = Succ v

bad :: String -> Err Value
bad s = Err s

bind :: Err Value -> (Value -> Err Value) -> Err Value
bind x f = case x of
    Succ v -> f v
    Err s -> bad s
```

□ Error Reporting Interpreter, Again

```
getVar :: Env -> String -> Err Value
getVar ((x,v):xs) y = if x == y then unit v else getVar xs y
getVar [] y = bad $ "unbound variable "++y


add :: Value -> Value -> Err Value
add (Num m) (Num n) = unit $ Num $ m + n
add m n = bad $ "expected numbers, found "++(show m)++" and "++(show n)


apply :: Value -> Value -> Err Value
apply (Fun v1) v2 = v1 v2
apply f _ = bad $ "expected function, found "++(show f)


interp :: Term -> Env -> Err Value
interp (Var x) env = getVar env x
interp (Con n) _ = unit $ Num n
interp (Add e1 e2) env =
    bind (interp e1 env) (\v1 ->
    bind (interp e2 env) (\v2 ->
    add v1 v2))
interp (Lam x e) env = unit $ Fun $ \v -> interp e ((x,v):env)
interp (App e1 e2) env =
    bind (interp e1 env) (\v1 ->
    bind (interp e2 env) (\v2 ->
    apply v1 v2))
```

□ Different Common Functionality

```
unit :: Value -> Value
unit v = v

bad :: String -> Value
bad s = Wrong

bind :: Value -> (Value -> Value) -> Value
bind x f = f x


data Value = Wrong
           | Num Int
           | Fun (Value -> Value)
```

□ Original Interpreter, Again

```
getVar :: Env -> String -> Value
getVar ((x,v):xs) y = if x == y then unit v else getVar xs y
getVar [] y = bad $ "unbound variable "++y


add :: Value -> Value -> Value
add (Num m) (Num n) = unit $ Num $ m + n
add m n = bad $ "expected numbers, found "++(show m)++" and "++(show n)


apply :: Value -> Value -> Value
apply (Fun v1) v2 = v1 v2
apply f _ = bad $ "expected function, found "++(show f)


interp :: Term -> Env -> Value
interp (Var x) env = getVar env x
interp (Con n) _ = unit $ Num n
interp (Add e1 e2) env =
    bind (interp e1 env) (\v1 ->
    bind (interp e2 env) (\v2 ->
    add v1 v2))
interp (Lam x e) env = unit $ Fun $ \v -> interp e ((x,v):env)
interp (App e1 e2) env =
    bind (interp e1 env) (\v1 ->
    bind (interp e2 env) (\v2 ->
    apply v1 v2))
```