Defining your own “show” function

FLOLAC 2008

```
data Tree a = Node a (Tree a) (Tree a)
            | Leaf a
```

```
Main> Leaf 3
ERROR: Cannot find "show" function for:
*** Expression : Leaf 3
*** Of type : Tree Integer
```

```
instance (Show a) => Show (Tree a) where
   show = showTree

showTree = …
```


Defining your “show” function

FLOLAC 2008

```
data Tree a = Node (Tree a) (Tree a)
            | Leaf a
```

```
instance (Show a) => Show (Tree a) where
   show = showTree
```

```
showTree :: (Show a) => Tree a -> String
showTree (Leaf x)   = show x
showTree (Node l r) =
 "<" ++ showTree l ++ "|" ++ showTree r ++ ">"
```

```
> Node (Leave 1) (Node (Leave 2) (Leave 3))
 "<1|<2|3>>"
```

Derived Instances

FLOLAC 2008

```
data Tree a = Node (Tree a) (Tree a)
            | Leaf a
          deriving (Eq, Show)
```

Constructs a "default instance" of class Show. Works for standard classes.

```
Main> Node (Leaf 1) (Node (Leaf 2) (Leaf 3))
 Node (Leaf 1) (Node (Leaf 2) (Leaf 3))
Main> (Leaf 3) == (Leaf 4)
 False
```


A Data type for λ-terms

FLOLAC 2008

```
data Term = V VarName
          | L VarName Term
          | A Term Term
          | I Int
          | Term :+ Term          -- addition
          | IFZ Term Term Term    -- if zero
                deriving (Show, Eq)
```

E ::= Id
| \x.E
| E_1 E_2
| i
| E_1 '+' E_2
| ifzero E_1 then E_2 else E_3

Environment-Based Evaluator

FLOLAC 2008

- Previously, we see substitution-based evaluation for lambda calculus with constants
- Now we show an environment-based evaluator.

```
eval env expr =  ...   --compute a value
```

where an environment is a finite function from Identifiers to values.

```
Pseudo code:
   eval [x=5, y=10] “x+y” = 15
```


LC Evaluator, 1

FLOLAC 2008

```
data Term = V VarName
          | L VarName Term          --lambda
          | A Term Term             --application
          | I Int
          | Term :+ Term             -- addition
          | IFZ Term Term Term    deriving (Show, Eq)
```

```
data Value = VI Int | VC (Value -> Value)
```

```
type VarName = String
-- Environment: associating values with `free' variables
type Env = [(VarName, Value)]
lookup :: Eq a => a -> [(a, b)] -> Maybe b
maybe              :: b -> (a -> b) -> Maybe a -> b
maybe n f Nothing  =  n
maybe n f (Just x) =  f x
lkup :: Env -> VarName -> Value
lkup env x = maybe err id $ lookup x env
   where err = error $ "Unbound variable " ++ x
```

LC Evaluator, 2


```
data Term = V VarName
          | L VarName Term
          | A Term Term
          | I Int
            | Term :+ Term           -- addition
          | IFZ Term Term Term     deriving (Show, Eq)
```

```
data Value = VI Int | VC (Value -> Value)
```

```
type VarName = String
-- Environment: associating values with `free' variables
type Env = [(VarName, Value)]
lkup :: Env -> VarName -> Value
lkup env x = maybe err id $ lookup x env
   where err = error $ "Unbound variable " ++ x
-- Denotational semantics. Why? How to make it operational?
eval :: Env -> Term -> Value
eval env (V x)   = lkup env x
  …
```


LC Evaluator, 3


```
data Term = … | I Int
          | Term :+ Term             -- addition
          | IFZ Term Term Term       deriving (Show, Eq)
```

```
eval env (I n) = VI n     -- already a value
eval env (e1 :+ e2) =
   let v1 = eval env e1
       v2 = eval env e2
    in case (v1,v2) of
        (VI n1, VI n2) -> VI (n1+n2)
        vs -> error $ "Trying to add non-integers:" ++ show vs
eval env (IFZ e1 e2 e3) =
    let v1 = eval env e1
     in case v1 of
          VI 0 -> eval env e2
          VI _ -> eval env e3
          v    -> error $ "Trying to compare a non-integer
                    to 0: " ++ show v
```

```
--data Value = VI Int | VC (Value -> Value)
```

LC Evaluator, 4


```
data Term = … | L VarName Term    --lambda
              | A Term Term       --app
          … deriving (Show, Eq)
```

```
data Value = VI Int | VC (Value -> Value)
```

```
eval env (L x e) = VC (\v -> eval (ext env (x,v)) e)

ext :: Env -> (VarName,Value) -> Env
ext env xt = xt : env

eval env (A e1 e2) =
   let v1 = eval env e1
       v2 = eval env e2
    in case v1 of
        VC f  -> f v2
        v     -> error $ "Trying to apply a non-function: “
                         ++ show v
```