Magic with Algebraic Data Types

Functional Programming 2017
Christophe Scholliers

Examples from Don Sannella University of Edinburgh
Informatics 1 Functional Programming Lecture 9

Everything is an algebraic type

```
data Bool       = False   | True
data List a     = Nil     | Cons a (List a)
data Nat        = Zero    | Succ Nat
data Maybe  a   = Nothing | Just a
data Either a b = Left a  | Right b
data Pair   a b = Pair a b

data Exp    = Lit Int | Add Exp Exp | Mul Exp Exp
data Tree a = Empty   | Leaf a | Branch (Tree a) (Tree a)
data Season = Winter  | Spring | Summer | Fall
data Shape  = Circle Float | Rectangle Float Float
```

Booleans

Boolean

```
data Bool = False | True

not :: Bool -> Bool
not False  =  True
not True   =  False

(&&) :: Bool -> Bool -> Bool
False && q = False
True &&  q = q

(||) :: Bool -> Bool -> Bool
False|| q = q
True || q = True
```

Showing Booleans

```
eqBool :: Bool -> Bool -> Bool
eqBool False False  =  True
eqBool False True   =  False
eqBool True  False  =  False
eqBool True  True   =  True

showBool :: Bool -> String
showBool False  =  "False"
showBool True   =  "True"
```

Seasons

Seasons

```haskell
data Season = Winter | Spring | Summer | Fall

next :: Season -> Season
next Winter  =  Spring
next Spring  =  Summer
next Summer  =  Fall
next Fall    =  Winter
```

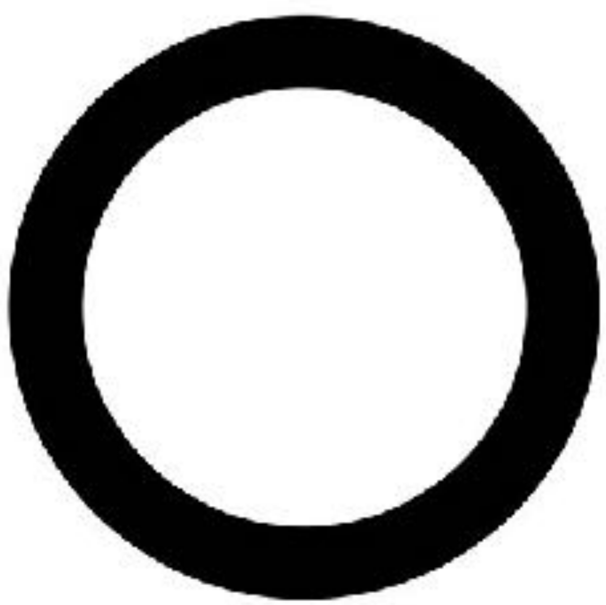
Seasons

```
data Season = Winter | Spring | Summer | Fall

eqSeason :: Season -> Season -> Bool
eqSeason Winter Winter  =  True
eqSeason Spring Spring  =  True
eqSeason Summer Summer  =  True
eqSeason Fall   Fall    =  True
eqSeason x      y       =  False

showSeason :: Season -> String
showSeason Winter  =  "Winter"
showSeason Spring  =  "Spring"
showSeason Summer  =  "Summer"
showSeason Fall    =  "Fall"
```

Seasons

```haskell
data Season = Winter | Spring | Summer | Fall

toInt :: Season -> Int
toInt Winter = 0
toInt Spring = 1
toInt Summer = 2
toInt Fall   = 3

fromInt :: Int -> Season
fromInt 0  =  Winter
fromInt 1  =  Spring
fromInt 2  =  Summer
fromInt 3  =  Fall

next' :: Season -> Season
next' x  =  fromInt ((toInt x + 1) `mod` 4)

eqSeason' :: Season -> Season -> Bool
eqSeason' x y  =  (toInt x == toInt y)
```

Shape



Radius

Height

Width

Shape

```haskell
type Radius = Float
type Width  = Float
type Height = Float

data Shape  = Circle Radius | Rect Width Height

area :: Shape -> Float
area (Circle r) = pi * r^2
area (Rect w h) = w*h
```

Shape eq and show

```
eqShape :: Shape -> Shape -> Bool
eqShape (Circle r) (Circle r’) = (r == r’)
eqShape (Rect w h) (Rect w’ h’) = (w==w’)&&(h==h’)
eqShape x          y           = False

showShape :: Shape -> String
showShape (Circle r) = "Circle " ++ showF r
showShape (Rect w h) = "Rect "++ showF w++" "++showF h

showF :: Float -> String
showFx | x >= 0      = show x
       | otherwise   = "(" ++ show x ++ ")"
```

Shape test and selectors

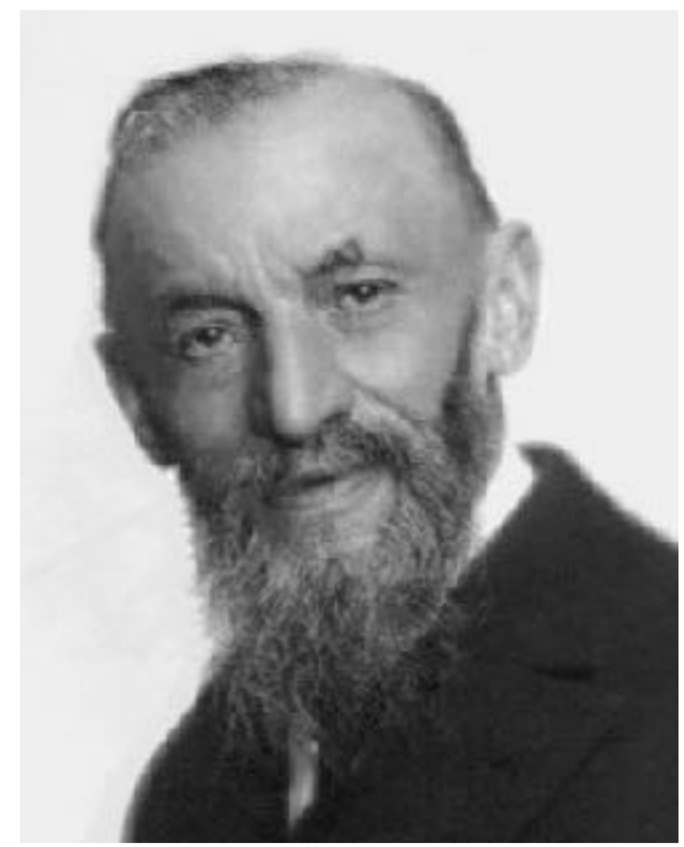
```
isCircle :: Shape -> Bool
isCircle (Circle r)  =  True
isCircle (Rect w h)  =  False

isRect :: Shape -> Bool
isRect (Circle r)  =  False
isRect (Rect w h)  =  True

radius :: Shape -> Float
radius (Circle r)  =  r

width :: Shape -> Float
width (Rect w h)   = w

height :: Shape -> Float
height (Rect w h)  =  h
```

Pattern Matching

```
area :: Shape -> Float
area (Circle r) = pi * r^2
area (Rect w h) = w*h

area :: Shape -> Float
area s =
  if isCircle s then
    let
      r = radius s
    in
        pi * r^2
  else if isRect s then
      let
        w = width s
        h = height s
      in
        w*h
  else error "impossible"
```

Natural Numbers

```
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15
,16,17,18,19,20,21,22,23,24,25,26,27
,28,29,30,31,32,33,34,35,36,37,38,39
,40,41,42,43,44,45,46,47,48,49,50,51
,52,53,54,55,56,57,58,59,60,61,62,63
,64,65,66,67,68,69,70,71,72,73,74,75
,76,77,78,79,80,81,82,83,84,85,86,87
,88,89,90,91,92,93,94,95,96,97,98,99
,100,101,102,103,104,105,106,107,108
,109,110,111,112,113,114,115,116,117
,118,119,120,121,122,123,124,125,126
,127,128,129,130,131,132,133,134,135
,136,137,138,139,140,141,142,143,144
,145,146,147,148,149,150,151,152,153
,154,155,156,157,158,159,160,161,162
,163,164,165,166,167,168,169,170,171
,172,173,174,175,176,177,178,179,180
,181,182,183,184,185,186,187,188,189
,190,191,192,193,194,195,196,197,198
,199,200,201,202,203,204,205,206,207
,208,209,210,211,212,213,214,215,216
,217,218,219,220,221,222,223,224,225
,226,227,228,229,230,231,232,233,234
,235,236,237,238,239,240,241,242,243
,244,245,246,247,248,249,250,251,252
,253,254,255,256,257,258,259,260,261
```



Natural Numbers

```
data Nat = Zero
         | Succ Nat

power :: Float -> Nat -> Float
power x Zero       =  1.0
power x (Succ n)   =  x * power x n
```

Natural Numbers

```haskell
data Nat = Zero
         | Succ Nat

add :: Nat -> Nat -> Nat
add m Zero     = m
add m (Succ n) = Succ (add m n)

mul :: Nat -> Nat -> Nat
mul m  Zero    = Zero
mul m (Succ n) = add (mul m n) m
```

Lists

A list of what ?

List

```
data  List a  =  Nil
              |  Cons a (List a)

append :: List a -> List a -> List a
append Nil ys             =  ys
append (Cons x xs) ys  =  Cons x (append xs ys)
```

List conversion

```
data  List a  =  Nil
              |  Cons a (List a)

toList :: [a] -> List a
toList []       =  Nil
toList (x:xs)   =  Cons x (toList xs)
```

Maybe

Maybe

```haskell
data Maybe a = Nothing | Just a

divide :: Int -> Int -> Maybe Int
divide n 0  =  Nothing
divide n m  =  Just (n `div` m)

power :: Maybe Int -> Int -> Int
power Nothing  n = 2 ^ n
power (Just m) n = m ^ n
```

Using a Maybe

```haskell
data Maybe a = Nothing | Just a

divide :: Int -> Int -> Maybe Int
divide n 0  =  Nothing
divide n m  =  Just (n `div` m)

divide_plus n m = case divide n m of
                      Nothing -> 3
                      Just r -> r + 3
```

Record Syntax

Record Syntax

```
data Person = Person String String Int Float String
String deriving (Show)
```

```
ghci> let guy = Person "Buddy" "Finklestein" 43 184.2
"526-2928" "Chocolate"
ghci> guy
Person "Buddy" "Finklestein" 43 184.2 "526-2928"
"Chocolate"
```

Record Syntax

```
firstName :: Person -> String
firstName (Person firstname _ _ _ _ _) = firstname

lastName :: Person -> String
lastName (Person _ lastname _ _ _ _) = lastname

age :: Person -> Int
age (Person _ _ age _ _ _) = age

height :: Person -> Float
height (Person _ _ _ height _ _) = height


...
```

Record Syntax

```
ghci> let guy = Person "Buddy" "Finklestein" 43 184.2
"526-2928" "Chocolate"
ghci> firstName guy
"Buddy"
ghci> height guy
184.2
ghci> flavor guy
"Chocolate"
```

Record Syntax

```haskell
data Person = Person { firstName   :: String
                     , lastName    :: String
                     , age         :: Int
                     , height      :: Float
                     , phoneNumber :: String
                     , flavor      :: String
                     } deriving (Show)
```

```
ghci> :t flavor
flavor :: Person -> String
ghci> :t firstName
firstName :: Person -> String
```

Typeclasses

```haskell
class Eq a where
    (==) :: a -> a -> Bool
    (/=) :: a -> a -> Bool
    x == y = not (x /= y)
    x /= y = not (x == y)
```

Typeclasses

```haskell
class Eq a where
    (==) :: a -> a -> Bool
    (/=) :: a -> a -> Bool
    x == y = not (x /= y)
    x /= y = not (x == y)
```

We only have to implement one !

```haskell
data TrafficLight = Red | Yellow | Green

instance Eq TrafficLight where
    Red    == Red    = True
    Green  == Green  = True
    Yellow == Yellow = True
         _ == _      = False
```

Typeclasses

```haskell
instance Show TrafficLight where
    show Red      = "Red light"
    show Yellow   = "Yellow light"
    show Green    = "Green light"
```

Example

```
ghci> Red == Red
True
ghci> Red == Yellow
False
ghci> Red `elem` [Red, Yellow, Green]
True
ghci> [Red, Yellow, Green]
[Red light,Yellow light,Green light]
```

Functor

```haskell
class Functor f where
    fmap :: (a -> b) -> f a -> f b
```

List Functor

```
instance Functor [] where
    fmap = map
```

Maybe Functor

```
instance Functor Maybe where
    fmap f (Just x) = Just (f x)
    fmap f Nothing = Nothing
```

```
ghci> fmap (*2) (Just 200)
Just 400
ghci> fmap (*2) Nothing
Nothing
```

Expression Trees

Expression Trees

```
data Exp  =  Lit Int
          |  Add Exp Exp
          |  Mul Exp Exp

evalExp :: Exp -> Int
evalExp (Lit n)    =  n
evalExp (Add e f)  =  evalExp e + evalExp f
evalExp (Mul e f)  =  evalExp e * evalExp f
```

Expression Trees

```haskell
data Exp  =  Lit Int
          |  Add Exp Exp
          |  Mul Exp Exp

showExp :: Exp -> String
showExp (Lit n)     =  show n
showExp (Add e f)   =  par (showExp e ++ "+" ++ showExp f)
showExp (Mul e f)   =  par (showExp e ++ "*" ++ showExp f)

par :: String -> String
par s  =  "(" ++ s ++ ")"
```

Expression Trees

```
e0 = Add (Lit 2) (Mul (Lit 3) (Lit 3))
```

```
         Add
       /     \
   Lit         Mul
    |         /   \
    2       Lit    Lit
             |      |
             3      3
```

Expression Trees Infix

```
data Exp = Lit Int
     | Exp `Add` Exp
     | Exp `Mul` Exp

evalExp :: Exp -> Int
evalExp (Lit n) = n

evalExp (e `Add` f) = evalExp e + evalExp f
evalExp (e `Mul` f) = evalExp e * evalExp f

showExp :: Exp -> String
showExp (Lit n) = show n
showExp (e `Add` f) = par (showExp e ++ "+" ++ showExp f)
showExp (e `Mul` f) = par (showExp e ++ "*" ++ showExp f)

par :: String -> String
par s = "("++s++")"
```

Expression Trees Infix

```haskell
data Exp = Lit Int
         | Exp :+: Exp
         | Exp :*: Exp

e0 = Lit 2 :+: (Lit 3 :*: Lit 3)

evalExp :: Exp -> Int
evalExp (Lit n)     =  n
evalExp (e :+: f)   =  evalExp e + evalExp f
evalExp (e :*: f)   =  evalExp e * evalExp f

showExp :: Exp -> String
showExp (Lit n)     =  show n
showExp (e :+: f)   =  par (showExp e ++ "+" ++ showExp f)
showExp (e :*: f)   =  par (showExp e ++ "*" ++ showExp f)

par :: String -> String pars = "("++s++")"
```

Propositions

Propositions

```
type Name = String

data Prp = Var Name
          | F
          | T
          | Not Prp
          | Prp :|: Prp
          | Prp :&: Prp
          deriving (Eq, Ord, Show)

type Names = [Name]
type Env = [(Name,Bool)]
```

Propositions

```
showPrp :: Prp -> String
showPrp (Var x)    =  x
showPrp (F)        =  "F"
showPrp (T)        =  "T"
showPrp (Not p)    =  par ("~" ++ showPrp p)
showPrp (p :|: q) =  par (showPrp p ++ "|" ++ showPrp q)
showPrp (p :&: q) =  par (showPrp p ++ "&" ++ showPrp q)

par :: String -> String
par s  =  "(" ++ s ++ ")"
```

Looking up a variable

```
lookUp :: Eq a => [(a,b)] -> a -> b
lookUp xys x  =  the [ y | (x',y) <- xys, x == x' ]
  where
  the [x]  =  x
```

Evaluation

```
eval :: Env -> Prp -> Bool
eval e (Var x)          =  lookUp e x
eval e (F)              =  False
eval e (T)              =  True
eval e (Not p)          =  not (eval e p)
eval e (p :|: q)        =  eval e p || eval e q
eval e (p :&: q)        =  eval e p && eval e q
```

Example

e0 = [("a",True)]

```
  eval e0 (Var "a" :&: Not (Var "a"))
=
  (eval e0 (Var "a")) && (eval e0 (Not (Var "a")))
=
  (lookup e0 "a") && (eval e0 (Not (Var "a")))
=
  True && (eval e0 (Not (Var "a")))
=
  True && (not (eval e0 (Var "a")))
= ... =
  True && False
=
False
```

Satisfiable

Extracting Variable Names

```
names :: Prp -> Names
names (Var x)     =  [x]
names (F)         =  []
names (T)         =  []
names (Not p)     =  names p
names (p :|: q)   =  nub (names p ++ names q)
names (p :&: q)   =  nub (names p ++ names q)
```

Generating Environments

```haskell
envs :: Names -> [Env]
envs []       =  [[]]
envs (x:xs)   =  [ (x,b):e | b <- bs, e <- envs xs ]
  where
  bs = [False,True]
```

Satisfiable

```haskell
satisfiable :: Prp -> Bool
satisfiable p  =  or [ eval e p | e <- envs (names p) ]
```

Lambda Calculus++
Haskell - - - -

Mini-Haskell

```
type Name = String

data Term = Var Name
          | Con Integer
          | Add Term Term
          | Lam Name Term
          | App Term Term
           deriving (Show,Eq)
```

Mini-Haskell

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

instance  Show Value where
    show Wrong     = "Wrong"
    show (Num i)   = "Int " ++ show i
    show (Fun _)   = "function “
```

```
eval :: Term -> Env -> Value
```

Environments

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

getVar :: Env -> Name -> Value
getVar []                _ = Wrong
getVar ((k1,v) : r)  k2
     | k1 == k2  = v
     | otherwise = getVar r k2
```

Apply

```haskell
add :: Value -> Value -> Value
add (Num x) (Num y) = Num (x + y)
add _       _       = Wrong

apply :: Value -> Value -> Value
apply (Fun f) t2 = f t2
apply _       _  = Wrong
```

Eval

```
eval :: Term -> Env -> Value
eval (Var x)      env  = getVar env x
eval (Con x)      _    = Num x
eval (Add t1 t2) env  = add (eval t1 env) (eval t2 env)
eval (Lam n  b ) env  = Fun (\x -> eval b ((n,x) : env))
eval (App t1 t2) env  = apply (eval t1 env) (eval t2 env)
```

Zooming in on Lambda

```
eval :: Term -> Env -> Value
eval (Lam n  b ) env  = Fun (\x -> eval b ((n,x) : env))
```

```haskell
type Name = String

data Term = Var Name
     | Con Integer
     | Add Term Term
     | Lam Name Term
     | App Term Term
      deriving (Show,Eq)

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

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

```haskell
instance  (Show Value) where
    show Wrong     = "Wrong"
    show (Num i)   = "Int " ++ show i
    show (Fun _)   = "function “

getVar :: Env -> Name -> Value
getVar []             _ = Wrong
getVar ((k1,v) : r)  k2
    | k1 == k2   = v
    | otherwise = getVar r k2

add :: Value -> Value -> Value
add (Num x) (Num y) = Num (x + y)
add _       _       = Wrong
```

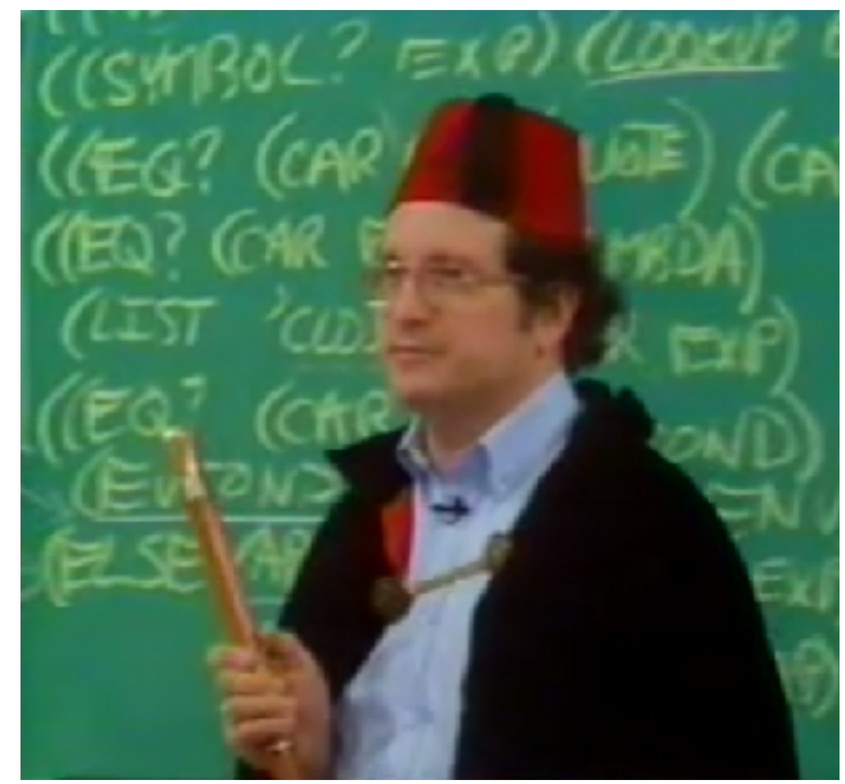
```haskell
apply :: Value -> Value -> Value
apply (Fun f) t2 = f t2
apply _       _  = Wrong
```

```haskell
eval :: Term -> Env -> Value
eval (Var x)     env  = getVar env x
eval (Con x)     _    = Num x
eval (Add t1 t2) env  = add (eval t1 env) (eval t2 env)
eval (Lam n  b ) env  = Fun (\x -> eval b ((n,x) : env))
eval (App t1 t2) env  = apply (eval t1 env) (eval t2 env)
```

Eval-Apply

Scheme

DrRacket



Sussman


(CADR EXP
ENV
ENV

Video Link

https://www.youtube.com/watch?v=0m6hoOelZH8&t=246s


THE End

λ Lambda Calculus

Functional Programming 2018-2019

Prof. Dr. Christophe Scholliers

Lambda Calculus

a very short introduction

Underpinning semantics
for functional languages

Alonzo Church

Semantics of Haskell is
basically a typed "extension" of LC

More material on:

http://www.inf.fu-berlin.de/lehre/WS03/alpi/lambda.pdf

Grammar

Variables Abstraction Application

$$M, N := x | (\lambda x. M) | (M\ N)$$

All functions are **anonymous**
and only take **one argument**

Left associative

$$x\ x\ x\ x\ x\ x$$

$$(((((x\ x)\ x)\ x)\ x)\ x)$$

Evaluation

$$\frac{}{(\lambda x.M)\ N \texttt{ -> } [x:=N]M}$$

$$\frac{M \texttt{ -> } M'}{(\lambda x.M) \texttt{ -> } (\lambda x.M')}$$

$$\frac{M \texttt{ -> } M'}{M\ N \texttt{ -> } M'\ N}$$

$$\frac{N \texttt{ -> } N'}{M\ N \texttt{ -> } M\ N'}$$

Evaluation

$$\frac{}{\texttt{(λx.M) N -> [x:=N]M}}$$

$$\frac{\texttt{M -> M'}}{\texttt{(λx.M)-> (λx.M')}}$$

$$\frac{\texttt{M -> M'}}{\texttt{M N -> M' N}}$$

$$\frac{\texttt{N -> N'}}{\texttt{M N -> M N'}}$$

Substitution Intuitive `[x:=N]`

$$(\lambda x.y)((\lambda z.z\ z)(\lambda w.w))$$

$$(\lambda x.y)\underline{((\lambda z.z\ z)(\lambda w.w))}$$

$$(\lambda x.y)\underline{((\lambda w.w)(\lambda w.w))}$$

$$\underline{(\lambda x.y)(\lambda w.w)}$$

$$y$$


COPY
PASTE

What could possibly go wrong ?

$$(\lambda x.\lambda x.x)(\lambda p.p)(\lambda q.q)$$

Substitution should not touch this x !

What are the characteristics x ?

Bound and Free Variables

$$\lambda x.\lambda y.\lambda z.x\ \ y\ \ q$$

Binder

Bound Free

Free Variables of a term

$$
\begin{aligned}
FV(x) &= x \\
FV(MN) &= FV(M) \cup FV(N) \\
FV(\lambda x.M) &= FV(M) \setminus \{x\}
\end{aligned}
$$

Substitution

$$
\begin{array}{lcll}
[x := N]x & = & N & \\
[x := N]z & = & z & \\
[x := P]M\ N & = & [x := P]M\ [x := P]N & \\
[x := P](\lambda x.M) & = & (\lambda x.M) & \\
[x := P](\lambda y.M) & = & (\lambda y.[x := P]M) & y \notin FV(M) \\
[x := P](\lambda y.M) & = & (\lambda y'.[x := P][y := y']M) & y \in FV(M), y'\ fresh, x \neq y
\end{array}
$$

Multi Argument Functions

$$\lambda x.\lambda y.x\ y$$

```
function(x,y) {
    return x(y)
}
```

Currying

$$\lambda x.\lambda y.x\ y$$

```
function(x,y) {
    return x(y)
}
```

```
function(x) {
    return function(y) {
        return x(y)
    }
}
```

Examples

$$(\lambda x.x)$$

$$(\lambda f.\lambda x.f\,f\,x)$$

Examples

$$(\lambda x.x)$$

Identity Function

$$(\lambda f.\lambda x.f\,f\,x)$$

Apply twice

Which one first ?

$$\frac{}{(\lambda x.M)\ N \rightarrow [x:=N]M}$$

$$\frac{M \rightarrow M'}{(\lambda x.M) \rightarrow (\lambda x.M')}$$

$$\frac{M \rightarrow M'}{M\ N \rightarrow M'\ N}$$

$$\frac{N \rightarrow N'}{M\ N \rightarrow M\ N'}$$

Call-by-value

$$\frac{}{\texttt{(λx.M) V -> [x:=V]M}}$$

$$\frac{\texttt{M -> M'}}{\texttt{M N -> M' N}} \qquad \frac{\texttt{N -> N'}}{\texttt{V N -> M N'}}$$

Example

```
(λy. y) ((λu. u) (λv. v))
(λy. y) (λv. v))
λv. v
```

Call by name

$$\frac{}{\texttt{(}\lambda\texttt{x.M) N -> [x:=N]M}}$$

$$\frac{\texttt{M -> M'}}{\texttt{M N -> M' N}}$$

Call by name

```
(λy. y) ((λu. u) (λv. v))
((λu. u) (λv. v))
λv. v
```

Lazy evaluation

```
(λy. y y) ((λu. u) (λv. v))
```

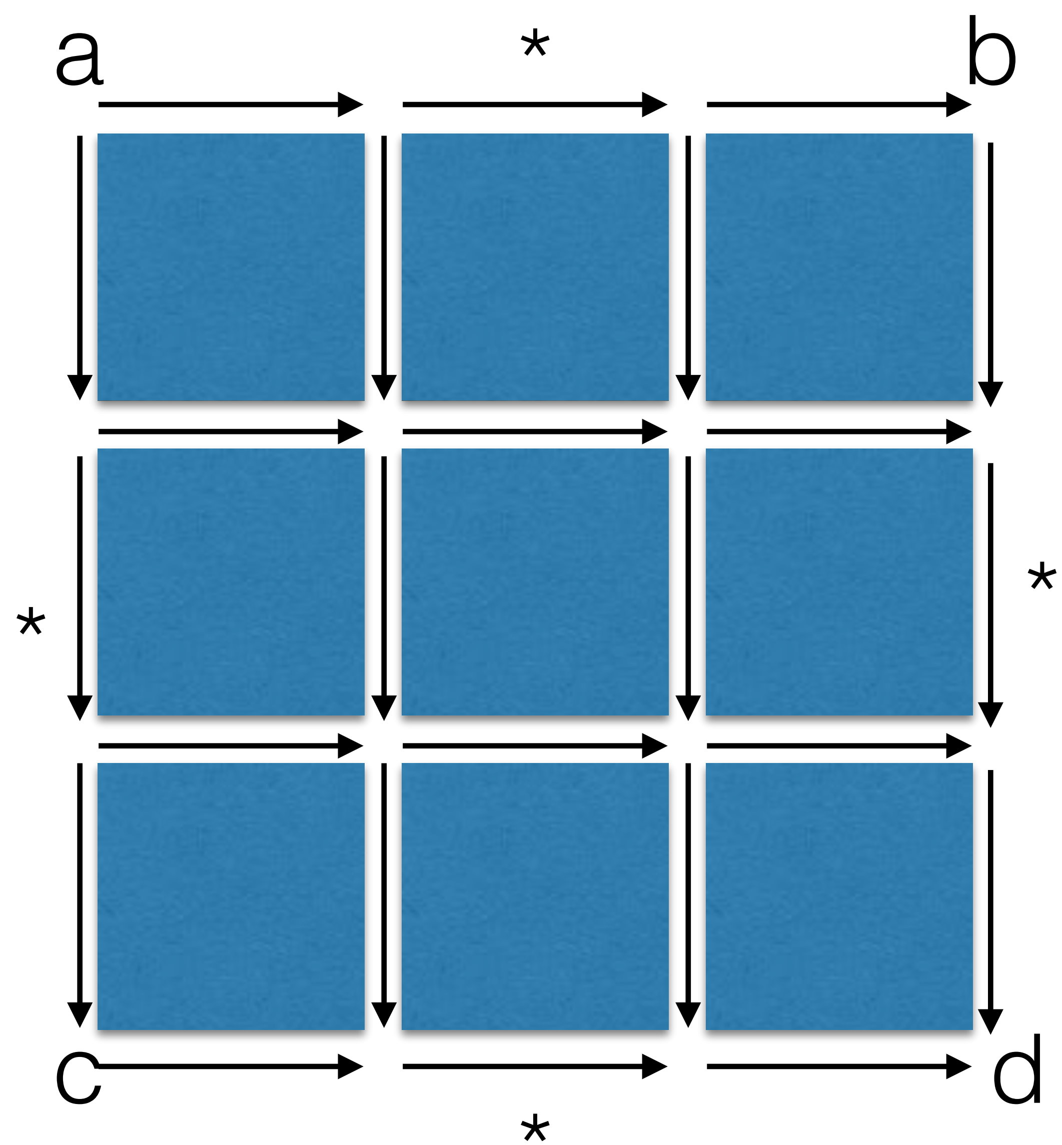
```
“((λu. u) (λv. v)) ((λu. u) (λv. v))”
```

```
((λv. v) (λv. v))
```

```
(λv. v)
```

Reduce **when needed** and **only once** !

Diamand property
Church Rosser
Confluence



Functional programming "assembly"

Encoding Booleans

$$\begin{array}{lcl} True & = & (\lambda x.\lambda y.x) \\ False & = & (\lambda x.\lambda y.y) \end{array}$$

And

$$And \quad = \quad (\lambda b1.\lambda b2.b1\ b2\ b1)$$

Or

$$Or \quad = \quad (\lambda b1.\lambda b2.b1\ b1\ b2)$$

Natural numbers

$$
\begin{aligned}
0 &= (\lambda f.\lambda x.x) \\
1 &= (\lambda f.\lambda x.fx) \\
2 &= (\lambda f.\lambda x.ffx) \\
\ldots \\
n &= (\lambda f.\lambda x.f^n x)
\end{aligned}
$$

Increment

$$inc \quad = \quad (\lambda n.\lambda f.\lambda x.f(nfx)$$

Add

$$add \quad = \quad (\lambda n.\lambda m.\lambda f.\lambda x.mf(nfx))$$

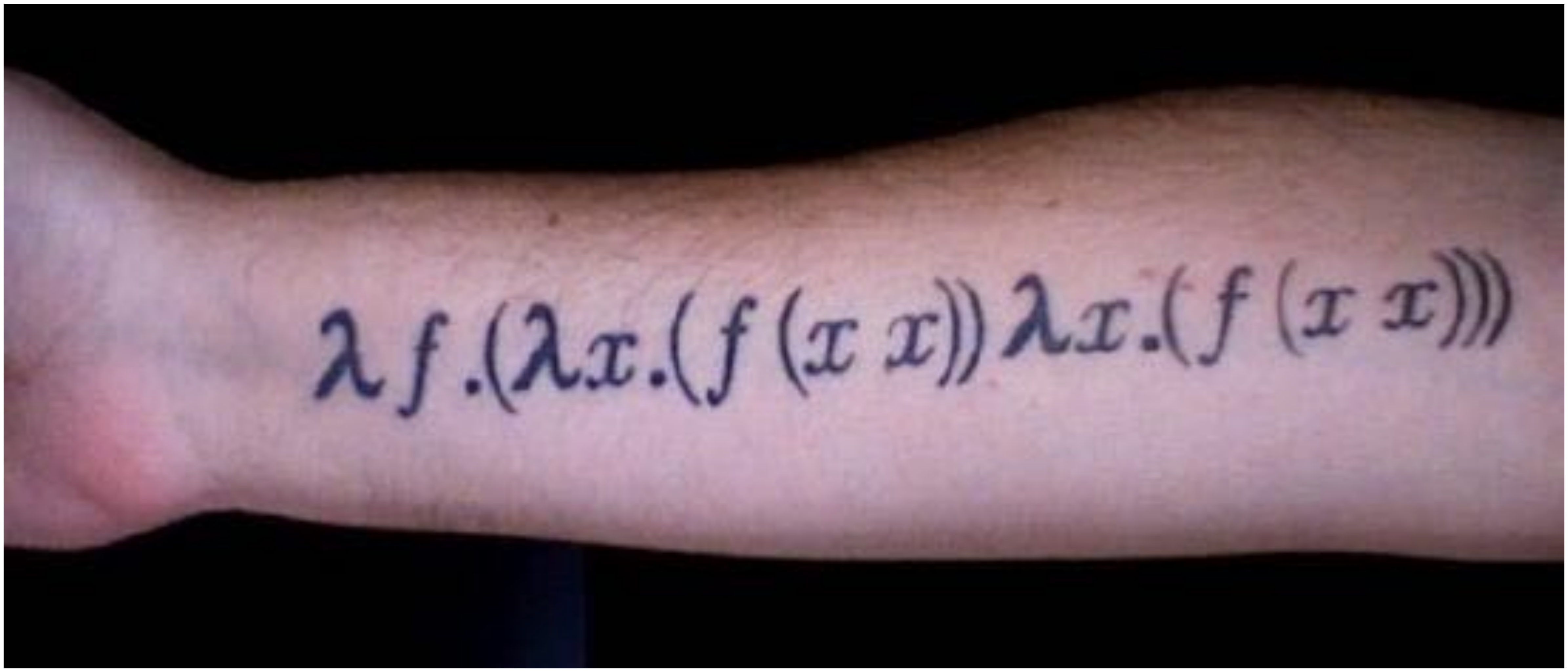
Multiply

$$mul \quad = \quad (\lambda n.\lambda m.\lambda f.m(nf))$$

Conditionals

$$\lambda b.\lambda t.\lambda f.b\ t\ f$$

Recursion



Recursion

```
λ f . (λx.  f  (x  x)) (λx.  f  (x  x))

λ f . (λx.  f  (x  x)) (λx.  f  (x  x))  g

(λx.  g  (x  x)) (λx.  g  (x  x))

(λx.  g  (x  x)) (λx.  g  (x  x))

g (λx.  g  (x  x) (λx.  g  (x  x))

g g g …
```

Delayed

$$\mathtt{Z}_f \quad = \quad \lambda\mathtt{y.}\ (\lambda\mathtt{x.}\ \mathtt{f}\ (\lambda\mathtt{y.}\ \mathtt{x\ x\ y}))\ (\lambda\mathtt{x.}\ \mathtt{f}\ (\lambda\mathtt{y.}\ \mathtt{x\ x\ y}))\ \mathtt{y}$$

Delayed

```
Z = λf. (λy.  (λx.  f  (λy.  x  x  y))
              (λx.  f  (λy.  x  x  y))  y)
```

Possible exam question

Write fibonacci in the lambda calculus