

Informatics 1

Functional Programming Lecture 9

Monday 27 and Tuesday 28 October 2014

Expression Trees  
as Algebraic Data Types

Don Sannella

University of Edinburgh

Part I

# 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
```

```
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, e1 :: Exp
```

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

```
e1 = Mul (Add (Lit 2) (Lit 3)) (Lit 3)
```

```
*Main> showExp e0
```

```
" (2+(3*3)) "
```

```
*Main> evalExp e0
```

```
11
```

```
*Main> showExp e1
```

```
" ((2+3)*3) "
```

```
*Main> evalExp e1
```

```
15
```

# 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

```
e0, e1 :: Exp
```

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

```
e1 = (Lit 2 `Add` Lit 3) `Mul` Lit 3
```

```
*Main> showExp e0
```

```
" (2+(3*3)) "
```

```
*Main> evalExp e0
```

```
11
```

```
*Main> showExp e1
```

```
" ((2+3)*3) "
```

```
*Main> evalExp e1
```

```
15
```

# Expression Trees, Symbols

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

```
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
par s = "(" ++ s ++ ")"
```

# Expression Trees, Symbols

```
e0, e1 :: Exp
```

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

```
e1 = (Lit 2 :+: Lit 3) :* Lit 3
```

```
*Main> showExp e0
```

```
" (2+(3*3)) "
```

```
*Main> evalExp e0
```

```
11
```

```
*Main> showExp e1
```

```
" ((2+3)*3) "
```

```
*Main> evalExp e1
```

```
15
```



Part II

Propositions

# Propositions

```
type Name = String
data Prop = Var Name
          | F
          | T
          | Not Prop
          | Prop :|: Prop
          | Prop :&: Prop
          deriving (Eq, Ord)
```

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

## Showing a proposition

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

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

# Names in a proposition

```
names :: Prop -> 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)
```

# Evaluating a proposition in an environment

```
eval :: Env -> Prop -> 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
```

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

# Propositions

```
p0 :: Prop
p0 = (Var "a" :&: Not (Var "a"))
```

```
e0 :: Env
e0 = [ ("a", True) ]
```

```
*Main> showProp p0
(a & (~a))
```

```
*Main> names p0
["a"]
```

```
*Main> eval e0 p0
False
```

```
*Main> lookUp e0 "a"
True
```

## How eval works

```
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
```

```
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
```

# Propositions

```
p1 :: Prop
p1 = (Var "a" :&: Var "b") :|:
      (Not (Var "a") :&: Not (Var "b"))
```

```
e1 :: Env
e1 = [("a", False), ("b", False)]
```

```
*Main> showProp p1
((a&b) | ((~a)&(~b)))
```

```
*Main> names p1
["a", "b"]
```

```
*Main> eval e1 p1
True
```

```
*Main> lookUp e1 "a"
False
```



# All possible environments

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

## Alternative

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

# All possible environments

```
envs []  
= [[]]
```

```
envs ["b"]  
= [("b",False):[]] ++ [("b",True ):[]]  
= [{"b",False}],  
  [{"b",True )}]
```

```
envs ["a", "b"]  
= [("a",False):e | e <- envs ["b"] ] ++  
  [("a",True ):e | e <- envs ["b"] ]  
= [("a",False):[("b",False)], ("a",False):[("b",True )]] ++  
  [("a",True ):[("b",False)], ("a",True ):[("b",True )]]  
= [{"a",False}, {"b",False}],  
  [{"a",False}, {"b",True )}],  
  [{"a",True }, {"b",False}],  
  [{"a",True }, {"b",True )}]
```

# Satisfiable

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

# Propositions

```
p1 :: Prop
p1 = (Var "a" :&: Var "b") :|:
      (Not (Var "a") :&: Not (Var "b"))
```

```
*Main> envs (names p1)
[[("a",False), ("b",False)],
  ("a",False), ("b",True)],
  ("a",True ), ("b",False)],
  ("a",True ), ("b",True )]]
```

```
*Main> [ eval e p1 | e <- envs (names p1) ]
[True,
 False,
 False,
 True]
```

```
*Main> satisfiable p1
True
```

## Part III

# Maybe, Maybe Not

# Optional Data

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

## Optional argument

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

## Optional result

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

## Using an Optional Result

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

```
wrong :: Int -> Int -> Int
wrong n m = divide n m + 3
```

```
right :: Int -> Int -> Int
right n m = case divide n m of
              Nothing -> 3
              Just r -> r + 3
```

## Part IV

# Union of Two Types



## Either a or b

```
data Either a b = Left a | Right b
```

```
mylist :: [Either Int String]
mylist = [Left 4, Left 1, Right "hello", Left 2,
          Right " ", Right "world", Left 17]
```

```
addints :: [Either Int String] -> Int
addints [] = 0
addints (Left n : xs) = n + addints xs
addints (Right s : xs) = addints xs
```

```
addints' :: [Either Int String] -> Int
addints' xs = sum [n | Left n <- xs]
```

## Either a or b

```
data Either a b = Left a | Right b
```

```
mylist :: [Either Int String]
mylist  = [Left 4, Left 1, Right "hello", Left 2,
          Right " ", Right "world", Left 17]
```

```
addstrs :: [Either Int String] -> String
addstrs [] = ""
addstrs (Left n : xs) = addstrs xs
addstrs (Right s : xs) = s ++ addstrs xs
```

```
addstrs' :: [Either Int String] -> String
addstrs' xs = concat [s | Right s <- xs]
```

## Part V

Aside:

All sublists of a list

## All sublists of a list

```
subs :: [a] -> [[a]]
```

```
subs [] = [[]]
```

```
subs (x:xs) = subs xs ++ [ x:ys | ys <- subs xs ]
```

## All sublists of a list

```
subs []  
= [[]]
```

```
subs ["b"]  
= subs [] ++ ["b":ys | ys <- subs []]  
= [[]] ++ ["b":[]]  
= [[], ["b"]]
```

```
subs ["a", "b"]  
= subs ["b"] ++ ["a":ys | ys <- subs ["b"]]  
= [[], ["b"]] ++ ["a":[], "a":["b"]]  
= [[], ["b"], ["a"], ["a", "b"]]
```

## Part VI

# The Universal Type and Micro-Haskell

# The Universal Type and Micro-Haskell

```
data Univ = UBool Bool
          | UInt Int
          | UList [Univ]
          | UFun (Univ -> Univ)
```

```
data Hask = HTrue
          | HFalse
          | HIf Hask Hask Hask
          | HLit Int
          | HEq Hask Hask
          | HAdd Hask Hask
          | HVar Name
          | HLam Name Hask
          | HApp Hask Hask
```

```
type HEnv = [(Name, Univ)]
```

# Show and Equality for Universal Type

```
showUniv :: Univ -> String
showUniv (UBool b)      = show b
showUniv (UInt i)       = show i
showUniv (UList us)     =
  "[" ++ concat (intersperse "," (map showUniv us)) ++ "]"
```

```
eqUniv :: Univ -> Univ -> Bool
eqUniv (UBool b) (UBool c)      = b == c
eqUniv (UInt i) (UInt j)        = i == j
eqUniv (UList us) (UList vs)    =
  and [ eqUniv u v | (u,v) <- zip us vs ]
eqUniv u v                       = False
```

Can't show functions or test them for equality.



# Micro-Haskell in Haskell

```
hEval :: Hask -> HEnv -> Univ
hEval HTrue r          = UBool True
hEval HFalse r         = UBool False
hEval (HIf c d e) r    =
  hif (hEval c r) (hEval d r) (hEval e r)
  where hif (UBool b) v w = if b then v else w
hEval (HLit i) r       = UInt i
hEval (HEq d e) r      = heq (hEval d r) (hEval e r)
  where heq (UInt i) (UInt j) = UBool (i == j)
hEval (HAdd d e) r     = hadd (hEval d r) (hEval e r)
  where hadd (UInt i) (UInt j) = UInt (i + j)
hEval (HVar x) r       = lookUp r x
hEval (HLam x e) r     = UFun (\ v -> hEval e ((x,v):r))
hEval (HApp d e) r     = happ (hEval d r) (hEval e r)
  where happ (UFun f) v = f v

lookUp :: HEnv -> Name -> Univ
lookUp x r = head [ v | (y,v) <- r, x == y ]
```

## Test data

```
h0 =
  (HApp
    (HApp
      (HLam "x" (HLam "y" (HAdd (HVar "x") (HVar "y"))))
      (HLit 3))
    (HLit 4))

test_h0 = eqUniv (hEval h0 []) (UInt 7)
```