Part I

Expression Trees

Now we can use the ideas behind the definitions of List etc. to define EXPRESSIONS and functions that manipulate them.
Expression Trees

Arithmetic expressions first. Called expression TREES because they reflect
the tree-like structure of expressions.
Unlike (for example) arithmetic expressions represented using String.

```
data Exp = Lit Int  -- An Exp is either a Lit (LITERAL) integer
    | Add Exp Exp  -- or Add of two expressions
    | Mul Exp Exp  -- or Mul (Multiplication) of two expressions.
```

evalExp :: Exp -> Int  -- Evaluating expressions: given an Exp, return its value.
evalExp (Lit n)  =  n  -- Left-hand side pattern :: Exp.
evalExp (Add e f) = evalExp e + evalExp f
evalExp (Mul e f) = evalExp e * evalExp f

data Exp = Lit Int  -- An Exp is either a Lit (LITERAL) integer
    | Add Exp Exp  -- or Add of two expressions
    | Mul Exp Exp  -- or Mul (Multiplication) of two expressions.
```

evalExp :: Exp -> Int  -- Evaluating expressions: given an Exp, return its value.
evalExp (Lit n)  =  n  -- Left-hand side pattern :: Exp.
evalExp (Add e f) = evalExp e + evalExp f
evalExp (Mul e f) = evalExp e * evalExp f
```

showExp :: Exp -> String  -- Converting an Exp into a 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

Two expressions with same literals and operators but different structure.

e0, e1 :: Exp

\[
e0 = \text{Add } (\text{Lit } 2) (\text{Mul } (\text{Lit } 3) (\text{Lit } 3))
\]

\[
e1 = \text{Mul } (\text{Add } (\text{Lit } 2) (\text{Lit } 3)) (\text{Lit } 3)
\]

*Main> showExp e0
"(2+(3*3))"

*Main> evalExp e0
11

*Main> showExp e1
"((2+3)*3)"

*Main> evalExp e1
15

Trees in Computer Science are drawn upside-down
Same in Linguistics. Terminology: ROOT, BRANCH, LEAF
Expression Trees, Infix

\[ \text{data \ Exp} = \text{Lit \ Int} \]
\[ \quad | \text{Exp} \ 'Add' \ \text{Exp} \]
\[ \quad | \text{Exp} \ 'Mul' \ \text{Exp} \]

\[ \text{evalExp :: Exp} \to \text{Int} \]
\[ \text{evalExp \ (Lit \ n)} = n \]
\[ \text{evalExp \ (e} \ 'Add' \ f) = \text{evalExp \ e} + \text{evalExp \ f} \]
\[ \text{evalExp \ (e} \ 'Mul' \ f) = \text{evalExp \ e} * \text{evalExp \ f} \]

\[ \text{showExp :: Exp} \to \text{String} \]
\[ \text{showExp \ (Lit \ n)} = \text{show \ n} \]
\[ \text{showExp \ (e} \ 'Add' \ f) = \text{par} \ (\text{showExp \ e} ++ \ "\+" \ ++ \text{showExp \ f}) \]
\[ \text{showExp \ (e} \ 'Mul' \ f) = \text{par} \ (\text{showExp \ e} ++ \ "\*" \ ++ \text{showExp \ f}) \]

\[ \text{par :: String} \to \text{String} \]
\[ \text{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

```haskell
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 ++ " ) "
```

Or, we can use infix SYMBOLS. Remember, constructors always start with a capital letter. Symbols used as constructors need to start with : Here we put : at the end too, just for symmetry.
Expression Trees, Symbols

e0, e1 :: Exp
\n\n\n\n\n\n\ne0 = 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

We can do the same thing for propositions from Inf1-CL

type Name = String

data Prop = Var Name
| F
| T
| Not Prop
| Prop :|: Prop
| Prop :&: Prop

deriving (Eq, Ord) This part will be explained in a later lecture (type classes)

A Prop (proposition) is either a Var (variable) with a name
or F (False - using F to avoid reuse of Bool constructor)
or T (True)
or Not (negation) of a Prop
or :|: (disjunction) of two Props
or :&: (conjunction) of two Props
(we could add :=>: for implication etc.)

Note, the first case is Var Name, not just Name - we need the constructor to distinguish between cases.

type Names = [Name] Names - will be used for sets of names
type Env = [(Name,Bool)] Env (environments) - will be used to map names to values
Showing a proposition

```haskell
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 ++ ")"
```

Converting a Prop to a string. Notice how recursion is essential for this definition.
Notice how the structure of the definition follows exactly the structure of the type definition:
- there is one equation for each constructor
- the clauses for Var, F and T aren't recursive
- the clauses for Not, :+:, :&: are recursive, in just the same way that the type definition is recursive
You can read off the "shape" of the function definition from the form of the type definition.
You've seen this before for recursive definitions over lists.

You can write function definitions on algebraic types that don't follow the shape of the type definition, for instance

```haskell
falsify :: Prop -> Prop
falsify p = F
```
but following the shape is common and a good starting point.
Names in a proposition

```haskell
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)
```

Computing the set of all of the variable names in a Prop.
Important if you want to build a truth table for a proposition.
The built-in function `nub` removes duplicates from a list.
Evaluating a proposition in an environment

\[
\begin{align*}
eval & : \text{Env} \rightarrow \text{Prop} \rightarrow \text{Bool} \\
eval e (\text{Var } x) &= \text{lookUp } e \ x \\
eval e \ F &= \text{False} \\
eval e \ T &= \text{True} \\
eval e (\text{Not } p) &= \text{not } (\eval e \ p) \\
eval e (p :\mid: q) &= \eval e \ p \mid\mid \eval e q \\
eval e (p :\&: q) &= \eval e \ p \&\& \eval e q
\end{align*}
\]

\[
\begin{align*}
\text{lookUp} & : \text{Eq } a \Rightarrow [(a,b)] \rightarrow a \rightarrow b \\
\text{lookUp } xys \ x &= \text{the } [ y \mid (x',y) \leftarrow xys, x == x' ] \\
\text{where} \\
\text{the } [x] &= x
\end{align*}
\]

Evaluating a Prop tells us if it's true or false. Only makes sense if we provide an ENVIRONMENT which gives the values of the variables.

Evaluation is along similar lines to evaluation of arithmetic expressions. Left-hand pattern :: Prop. Right-hand side :: Bool

lookUp is for looking up the value of a variable in an Env. An Env is a list of (variable name, value) pairs. The comprehension gives a list, which should contain one value. The returns that value.
Propositions

\[ p_0 :: \text{Prop} \]
\[ p_0 = (\text{Var} \ "a" :\&: \text{Not} \ (\text{Var} \ "a")) \]

\[ e_0 :: \text{Env} \]
\[ e_0 = [("a", \text{True})] \]

*Main> showProp p0
(a&(¬a))

*Main> names p0
["a"]

*Main> eval e0 p0
False

*Main> lookUp e0 "a"
True
How eval works

\[
\begin{align*}
\text{eval } e \ (\text{Var } x) &\ = \ \text{lookUp } e \ x \\
\text{eval } e \ F &\ = \ False \\
\text{eval } e \ T &\ = \ True \\
\text{eval } e \ (\text{Not } p) &\ = \ \text{not} \ (\text{eval } e \ p) \\
\text{eval } e \ (p :|: q) &\ = \ \text{eval } e \ p \ || \ \text{eval } e \ q \\
\text{eval } e \ (p :&: q) &\ = \ \text{eval } e \ p \ && \text{eval } e \ q
\end{align*}
\]

\[
\begin{align*}
\text{eval } e0 \ (\text{Var } "a" :&: \text{Not } (\text{Var } "a")) \\
= \ (\text{eval } e0 \ (\text{Var } "a")) \ && \ (\text{eval } e0 \ (\text{Not } (\text{Var } "a"))) \\
= \ (\text{lookup } e0 \ "a") \ && \ (\text{eval } e0 \ (\text{Not } (\text{Var } "a"))) \\
= \ True \ && \ (\text{eval } e0 \ (\text{Not } (\text{Var } "a"))) \\
= \ True \ && \ (\text{not} \ (\text{eval } e0 \ (\text{Var } "a"))) \\
= \ ... \ = \ True \ && \ False \\
= \ False
\end{align*}
\]

Here's how eval works for this example.
The result will also be False if the environment says that a is False.
So this proposition is a CONTRADICTION.
Propositions

\[ p_1 :: \text{Prop} \]
\[ p_1 = (\text{Var } "a" :\&: \text{Var } "b") :|:\]
\[ (\text{Not} (\text{Var } "a") :\&: \text{Not} (\text{Var } "b")) \]

\[ e_1 :: \text{Env} \]
\[ e_1 = [("a", \text{False}), ("b", \text{False})] \]

*Main> showProp p1
p1 is :|
((a&b)|((¬a)&(¬b))

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

*Main> eval e1 p1
True

*Main> lookUp e1 "a"
False
All possible environments

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

Alternative

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

To write functions for checking whether a Prop is a tautology, a contradiction, satisfiable etc. we need to compute all of the possible environments over a set of variables. Consider `envs (x:xs)` - combine the possible choices for `x` with all the possible choices for the other variables. Careful with `envs []` - there is one environment over the empty set of variables, namely the empty environment. If you define `envs [] = []`, then `envs anything = []`
All possible environments

```haskell
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)]]
```

Here's an example.
Satisfiable

\[
\text{satisfiable} :: \text{Prop} \rightarrow \text{Bool} \\
satisfiable \ p \ = \ \text{or} \ [ \ \text{eval} \ e \ p \ | \ e \ \leftarrow \ \text{envs} \ (\text{names} \ p) \ ]
\]

A Prop is satisfiable if there is at least one environment that makes it evaluate to True.
Combining:
- getting the set of variables in a Prop (names)
- getting all the environments over those names (envs)
- evaluating Prop in each of those environments (eval)
Apply or to the list of results to get the answer - False only if Prop evaluates to False in all those environments.
Propositions

\[ p_1 :: \text{Prop} \]
\[ p_1 = (\text{Var "a" \&\& \text{Var "b"}) :|: (\text{Not (Var "a") \&\& \text{Not (Var "b")}) \]

\[ \text{*Main>} \quad \text{envs (names p1)} \]
\[ \text{[["a",False), ("b",False)]}, \]
\[ \text{[["a",False), ("b",True)]}, \]
\[ \text{[["a",True), ("b",False)]}, \]
\[ \text{[["a",True), ("b",True)]]} \]

\[ \text{*Main>} \quad \text{[ eval e p1 | e <- envs (names p1) ]} \]
\[ [\text{True,} \]
\[ \text{False,} \]
\[ \text{False,} \]
\[ \text{True}] \]

\[ \text{*Main>} \quad \text{satisfiable p1} \]
\[ \text{True} \]

Here's an example.
Part III

Maybe, Maybe Not
Optional Data

Maybe a is a built-in type that is handy when a value of type a may be missing.

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

A value of type Maybe a is either Nothing (value missing) or a value of type a "wrapped up" with the constructor Just.

Optional argument

Useful for the situation where there is an optional argument, with a DEFAULT when it is not supplied.

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

Optional result

Useful when a function may have no result.

```haskell
divide :: Int -> Int -> Maybe Int
divide n 0  =  Nothing
divide n m  =  Just (n \text{`div`} m)
```
Using an Optional Result

\[ \text{divide :: Int} \rightarrow \text{Int} \rightarrow \text{Maybe Int} \]
\[ \text{divide } n \ 0 \ = \ \text{Nothing} \]
\[ \text{divide } n \ m \ = \ \text{Just } (n \ \text{`div` } m) \]

Using an optional result requires "unwrapping" it from the constructor.

\[ \text{wrong :: Int} \rightarrow \text{Int} \rightarrow \text{Int} \]
\[ \text{wrong } n \ m \ = \ \text{divide } n \ m \ + \ 3 \]

Using "normal" division: \( n \ \text{`div` } m + 3 \)

Doesn't work: divide \( n \ m :: \text{Maybe Int} \) and + adds an Int to an Int

\[ \text{right :: Int} \rightarrow \text{Int} \rightarrow \text{Int} \]
\[ \text{right } n \ m \ = \ \text{case } \text{divide } n \ m \ \text{of} \]
\[ \text{Nothing} \rightarrow 3 \]
\[ \text{Just } r \rightarrow r + 3 \]

Just \( r :: \text{Maybe Int} \), so \( r :: \text{Int} \)

case syntax is new: case expr of pat1 -> exp1 ... pat n -> expn
Part IV

Union of Two Types
Either a or b

This built-in type can be used to get lists with values of different types.
For instance, [Either Int Bool]

data Either a b = Left a | Right b

A value of type Either a b is either a value of type a, wrapped up using Left, or a value of type b, wrapped up using Right.

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

A function to add all of the integers in a list of type [Either Int String], ignoring the strings.

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

The same function, written using comprehension. Notice the use of the pattern Left n to select only the values of this form.
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

A function to concatenate all of the strings in a list of type [Either Int String], ignoring the integers.

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

The same function written using comprehension.

These examples haven't shown:
- types with multiple parameters
- mutually recursive types
- functional types as arguments of constructors
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

\[
\begin{align*}
\text{subs } [] & \quad = \quad [ [] ] \\
\text{subs } ["b"] & \quad = \quad \text{subs } [] + ["b":ys \mid ys \leftarrow \text{subs } [] ] \\
& \quad = \quad [ [] ] + ["b":[ [] ] ] \\
& \quad = \quad [ [], [ "b" ] ] \\
\text{subs } ["a","b"] & \quad = \quad \text{subs } ["b"] + ["a":ys \mid ys \leftarrow \text{subs } ["b"] ] \\
& \quad = \quad [ [], [ "b" ] ] + ["a":[ [] ], "a":[ "b" ] ] \\
& \quad = \quad [ [], [ "b" ], [ "a" ], [ "a","b" ] ]
\end{align*}
\]
Part VI

The Universal Type and Micro-Haskell

Optional material: an interpreter for a tiny subset of Haskell
The Universal Type and Micro-Haskell

```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

\[
\begin{align*}
\text{showUniv} & : \text{Univ} \rightarrow \text{String} \\
\text{showUniv} \ (\text{UBool} \ b) & = \ \text{show} \ b \\
\text{showUniv} \ (\text{UInt} \ i) & = \ \text{show} \ i \\
\text{showUniv} \ (\text{UList} \ us) & = \\
& \quad \quad \text{"[" ++ concat (intersperse "," (map showUniv us)) ++ "]"}
\end{align*}
\]

\[
\begin{align*}
\text{eqUniv} & : \text{Univ} \rightarrow \text{Univ} \rightarrow \text{Bool} \\
\text{eqUniv} \ (\text{UBool} \ b) \ (\text{UBool} \ c) & = b == c \\
\text{eqUniv} \ (\text{UInt} \ i) \ (\text{UInt} \ j) & = i == j \\
\text{eqUniv} \ (\text{UList} \ us) \ (\text{UList} \ vs) & = \\
& \quad \quad \text{and} \ [ \ \text{eqUniv} \ u \ v \ | \ (u,v) \leftarrow \text{zip} \ us \ vs \ ] \\
\text{eqUniv} \ u \ v & = \ \text{False}
\end{align*}
\]

Can’t show functions or test them for equality.
Micro-Haskell in Haskell

```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

\[
\begin{align*}
\text{h0} &= \text{HApp} \\
&= \text{HApp} \\
&= \text{HLam "x" (HLam "y" (HAdd (HVar "x") (HVar "y")))} \\
&= \text{HLit 3)} \\
&= \text{HLit 4)} \\

\text{test\_h0} &= \text{eqUniv (hEval h0 []) (UInt 7)}
\end{align*}
\]