Expression syntax

The AST is a non-recursive data type. To add recursion to psilo's syntax, I could use the Mu combinator, which is the type-level equivalent of the Y combinator:

```
newtype Mu f = Mu (f (Mu f))
type Expr = Mu AST
```

However, it so happens that the Free monad has a very similar definition:

```
data Free f a = Pure a | Free (f (Free f a))
```

Note the second constructor. So, instead, I use `Free`.

The `Free` monad constructor takes any `Functor` type and yields a monad for "free": you get generic instances of the `>>=` and `return` functions. These essentially build up values layer by layer and do not give your type any evaluation semantics.

Instead, it is on the programmer to write an interpreter function to unwrap and perform some computation on these values. This is exactly what the `interpreter` function from the `Evaluator` module does.

Thus, by using `Free`, not only do I get a recursive syntax definition with minimal complexity but I also get a suite of tools for building up expressions in my syntax and then tearing them down to yield a result, including `do` notation.

Not bad, huh?

```
{-# LANGUAGE DeriveFunctor #-}
{-# LANGUAGE DeriveFoldable #-}
{-# LANGUAGE DeriveTraversable #-}
{-# LANGUAGE StandaloneDeriving #-}
{-# LANGUAGE TypeSynonymInstances #-}
{-# LANGUAGE FlexibleInstances #-}
{-# LANGUAGE OverlappingInstances #-}

module Syntax where
```

```haskell
import Prelude hiding (sequence)
import Control.Monad.Free
import Data.Foldable (Foldable, fold)
import Data.Traversable (Traversable, sequence)

type Symbol = String

data AST a
    = AInteger Integer
    | ABoolean Bool
    | ASymbol { toSym :: Symbol }
    | ALambda [Symbol] a
    | AApply a a
    | AList [a]
    | ADefine Symbol a
    | AHylo Symbol [Symbol] a

deriving instance Show a => Show (AST a)
deriving instance Functor AST
deriving instance Foldable AST
deriving instance Traversable AST
deriving instance Eq a => Eq (AST a)
deriving instance Ord a => Ord (AST a)

type Expr = Free AST

instance Show a => Show (Expr a) where
    show (Pure _)   = ""
    show (Free x) = " ( " ++ show x ++ " ) "
```

The need occasionally arises to convert a list of `Free (ASymbol Symbol)` values to a list of `Symbol` values. This function serves that purpose:

```haskell
expr2symlist :: [Expr a] -> [Symbol]
expr2symlist ((Free (ASymbol x)):xs) = [x] ++ (expr2symlist xs)
expr2symlist _                       = []
```