1 To-do

- Convert to literate Haskell
- Make `SExpressable` on kind `*`
- Escape control characters in symbols
- Use gana, MonadFail for errors
- Rename define to value

2 Preamble

```
{-# LANGUAGE DeriveFoldable, DeriveFunctor, DeriveTraversable #-}

import Control.Arrow (first)
import Data.Char (isSpace)
import Data.Functor.Foldable (Fix(..), ana, cata)
import Data.List (intercalate)
import Text.ParserCombinators.ReadP (ReadP(..), (<++), readP_to_S, readS_to_P,
                                     skipSpaces, many, munch1, char, between)
```

3 S-Expressions

```
data SExpr = Symbol String
           | Sequence [SExpr]
           deriving (Eq)

instance Show SExpr where
  show (Symbol a)    = a
  show (Sequence xs) = "(" ++ intercalate " " (map show xs) ++ ")"
instance Read SExpr where
  readsPrec _ = readP_to_S grammar
    where
      grammar  = skipSpaces *> (sequence <++ symbol)
      sequence = Sequence <$> between (char '(') (skipSpaces *> char ')') (many grammar
      symbol   = Symbol <$> munch1 isSymChar

      isSymChar :: Char -> Bool
      isSymChar c = not (c `elem` "()" || isSpace c)

class SExpressable f where
  interpretLayer :: SExpr -> f SExpr
  expressLayer   :: f SExpr -> SExpr
```

4 ...

```
data Pattern a = PName String
               | PList String [Pattern a]
               deriving (Functor, Foldable, Traversable)

instance SExpressable Pattern where
  interpretLayer (Symbol var)                    = PName var
  interpretLayer (Sequence (Symbol head : tail)) = PList head $ map interpretLayer tail

  expressLayer (PName name) = Symbol name
  expressLayer (PList p ps) = Sequence $ Symbol p : map expressLayer ps


data DefineClause a = DefineClause String [Pattern a] a
                    deriving (Functor, Foldable, Traversable)

instance SExpressable DefineClause where
  interpretLayer (Sequence [Sequence (Symbol fname : args), expr]) = DefineClause fname

  expressLayer (DefineClause fname args expr) = Sequence [Sequence (Symbol fname : map

data Expr a = Reference String
            | Define a [DefineClause a]
            | Forall String a a
            | Apply a [a]
            deriving (Functor, Foldable, Traversable)

data DataConstructorDefinition a =
  DataConstructorDefinition String (Expr a)
  deriving (Functor, Foldable, Traversable)

instance SExpressable DataConstructorDefinition where
  interpretLayer (Sequence [(Symbol ctorName), ty])    = DataConstructorDefinition ctor
  expressLayer (DataConstructorDefinition ctorName ty) = Sequence [Symbol ctorName, exp

data TypeDefinition a =
  TypeDefinition String (Expr a) [DataConstructorDefinition a]
  deriving (Functor, Foldable, Traversable)

instance SExpressable TypeDefinition where
  interpretLayer (Sequence (Symbol "type" : Symbol typeCtor : ty : dataCtors)) =
    TypeDefinition typeCtor (interpretLayer ty) $ map interpretLayer dataCtors

  expressLayer (TypeDefinition typeCtor ty dataCtors) =
    Sequence $ Symbol "type" : Symbol typeCtor : expressLayer ty : map expressLayer dat

data TopLevel a = TypeDefinition a :+: Expr a

instance SExpressable Expr where
  interpretLayer (Symbol var)
= Reference var
```

```
  interpretLayer (Sequence (Symbol "define" : ty : clauses))
= Define ty $ map interpretLayer clauses
  interpretLayer (Sequence [Symbol "Forall", Symbol var, ty, expr]) = Forall var ty exp
  interpretLayer (Sequence (x : xs))
= Apply x xs

  expressLayer (Reference var)      = Symbol var
  expressLayer (Define ty clauses)  = Sequence (Symbol "define" : ty : map expressLayer
  expressLayer (Forall var ty expr) = Sequence [Symbol "Forall", Symbol var, ty, expr]
  expressLayer (Apply x xs)         = Sequence (x : xs)

interpret :: (SExpressable a, Functor a) => SExpr -> Fix a
interpret = ana interpretLayer

express :: (SExpressable a, Functor a) => Fix a -> SExpr
express = cata expressLayer

main :: IO ()
main = putStrLn "Hello, world!"
```