

## The Parser

This implements the basic s-expression syntax along with some sugar, like `let` bindings.

The result of one of the top-level parsing functions is a `Parser (Expr a)` value from which the `Expr` value may be extracted and given to the evaluator.

This is probably sub-optimal; `parsec` is a harsh master.

```
module Parser where

import Text.Parsec
import Text.Parsec.String (Parser, parseFromFile)
import Control.Applicative ((<$>))
import Control.Monad (mapAndUnzipM)
import Control.Monad.Free
import Data.List.Split (splitOn)

import qualified Text.Parsec.Expr as Ex
import qualified Text.Parsec.Token as Tok

import Syntax
import Lexer

parseNumber :: Parser (Expr a)
parseNumber = try ( do { n <- integer
                        ; return $ Free $ AInteger n
                        } )
```

Booleans are represented by the atoms `#t` and `#f`.

```
parseBoolean :: Parser (Expr a)
parseBoolean = do
  char '#'
  b <- char 't' <|> char 'f'
  case b of
    't' -> return $ Free $ ABoolean True
    'f' -> return $ Free $ ABoolean False
```

Symbols are like “atoms” in other lisps or Erlang. They are equivalent only to themselves and have no intrinsic value. They are mostly used to bind values in lambda abstractions.

```
parseSymbol :: Parser (Expr a)
parseSymbol = do
  sym <- operator <|> identifier
  sym' <- chomped sym
  return $ Free $ ASymbol sym'
  where chomped s = let s' = splitOn ":" s
                    in return $ s' !! 0
```

Lambda abstractions, or *functions*. A function definition is a list of symbols to bind to the elements of the argument list, and a psilo expression to evaluate in the context of the arguments.

```
parseFn :: Parser (Expr a)
parseFn = do
  reserved "\\\"
  optional whitespace
  (Free (AList arg)) <- parens parseQuotedList
  optional whitespace
  body <- parseExpr
  return $ Free $ ALambda (expr2symlist arg) body
```

let bindings are formally equivalent to wrapping an expression in an outer closure and immediately evaluating it, like so:

```
parseLetBinding :: Parser (Expr a)
parseLetBinding = do
  optional whitespace
  sym <- parseSymbol
  optional whitespace
  val <- parseExpr
  return $ Free . AList $ [sym, val]

parseLetBindings :: Parser (Expr a)
parseLetBindings = fmap (Free . AList) $ parens parseLetBinding `sepBy` whitespace

parseLet :: Parser (Expr a)
parseLet = do
  reserved "let"
  optional whitespace
  (Free (AList assns)) <- parens parseLetBindings
```

```

body <- parseExpr <|> return (Free (AList []))
(args,operands) <- (flip mapAndUnzipM) assns $ \(Free (AList (x:y:_))) -> return (x,y)
operands' <- return $ Free $ AList operands
fun <- return $ Free $ ALambda (expr2symlist args) body
return $ Free (AApply fun operands')

```

The application of a function to a list of arguments, a single symbol, or an arbitrary expression value.

```

parseApp :: Parser (Expr a)
parseApp = do
  try (reserved "apply") >> (do
    optional whitespace
    fun <- parseExpr
    optional whitespace
    body <- (try (char '\'') >> parens parseQuotedList)
      <|> (try (char '`') >> parens parseUnquotable)
    return $ Free (AApply fun body))
  <|> (do
    fst <-      try parseSymbol
      <|> try parseNumber
      <|> parseBoolean
      <|> try (parens parseFn)
      <|> parens parseApp

    optional whitespace
    rst <- fmap (Free . AList) $ parseExpr `sepBy` whitespace
    return $ Free (AApply fst rst))

```

Regular list created by the quote (') operator. Enters a state where everything is treated as a literal - no applications allowed.

```

parseQuotedList :: Parser (Expr a)
parseQuotedList = fmap (Free . AList) $ parseExprInQuote `sepBy` whitespace

```

Similar to a quoted list, except a comma operator (,) may be used to go back into a state where application is allowed.

```

parseUnquotable :: Parser (Expr a)
parseUnquotable = fmap (Free . AList) $ parseExprInQuasi `sepBy` whitespace

```

Many things may be quoted, not just lists.

```

parseQuote :: Parser (Expr a)
parseQuote = do
  x <- parseSymbol <|> parseNumber <|> parens parseQuotedList
  return $ (Free . AList) [(Free . ASymbol) "quote", x]

```

You can quasi-quote anything you can quote, though this is of dubious utility.

```

parseQuasi :: Parser (Expr a)
parseQuasi = do
  x <- parseSymbol <|> parseNumber <|> parens parseUnquotable
  return $ (Free . AList) [(Free . ASymbol) "quasi", x]

```

```

parseComma :: Parser (Expr a)
parseComma = do
  x <- parseSymbol <|> parseExpr
  return $ (Free . AList) [(Free . ASymbol) "comma", x]

```

Definitions - that is, permanent additions to the environment and store - are treated especially as strictly speaking they are not expressions.

```

parseDefn :: Parser (Expr a)
parseDefn = do
  optional whitespace
  reserved "="
  optional whitespace
  Free (ASymbol sym) <- parseSymbol
  optional whitespace
  body <- parseFunDef <|> parseSimpleDef
  return $ Free $ ADefine sym body

```

```

parseSimpleDef = parseExpr

```

```

parseFunDef = try $ do
  Free (AList args) <- parens parseQuotedList
  optional whitespace
  body <- parseExpr
  return $ Free $ ALambda (expr2symlist args) body

```

Top level expression parser

```

parseExpr :: Parser (Expr a)
parseExpr = parseBoolean
  <|> parseSymbol

```

```

<|> parseNumber
<|> (try (char '\\' ) >> parseQuote)
<|> (try (char '`' ) >> parseQuasi)
<|> parens ( parseDefn <|> parseFn <|> parseLet <|> parseApp )

```

Expression parser inside a quoted list

```

parseExprInQuote :: Parser (Expr a)
parseExprInQuote = parseBoolean
    <|> parseSymbol
    <|> parseNumber
    <|> (try (char '\\' ) >> parseQuote)
    <|> parens ( parseQuotedList )

```

Expression parser inside a quasiquoted list

```

parseExprInQuasi :: Parser (Expr a)
parseExprInQuasi = parseBoolean
    <|> parseSymbol
    <|> parseNumber
    <|> (try (reserved "\" ) >> parseQuote)
    <|> (try (reserved "`" ) >> parseQuasi)
    <|> (try (char ',') >> parseComma)
    <|> parens ( parseUnquotable )

```

```

contents :: Parser a -> Parser a
contents p = do
    whitespace
    r <- p
    eof
    return r

```

```

topLevel :: Parser [Expr a]
topLevel = many $ do
    x <- parseExpr
    return x

```

```

type Parsed a = Either ParseError [Expr a]

```

```

parseFile :: String -> IO (Parsed a)
parseFile fname = parseFromFile (contents topLevel) fname

```

```

parseTopLevel :: String -> Parsed a
parseTopLevel s = parse (contents topLevel) "<stdin>" s

```