Problem Set #4

due October 1

Problem 1. In this problem, you will define interpretations on an expression tree. The following ADT represents represents the expressions:

```
data ExprT = Val Int
           | Add ExprT ExprT
           | Mul ExprT ExprT
           deriving (Eq)
```

For example `Add (Mul (Val 1) (Val 2)) (Val 3)` represents `1 * 2 + 3`.

(1) The first task is to define `eval` which computes the value of the expression represented by the tree. For example:

```
> x = Add (Mul (Val 1) (Val 2)) (Val 3)
> eval x
5
```

(2) Next, define `prettyPrint` which prints the expression in its unevaluated form, adding parentheses around each addition and each multiplication expression. For example:

```
> x = Add (Mul (Val 1) (Val 2)) (Val 3)
> prettyPrint x
"((1 * 2) + 3)"
```

(3) Finally, revise `prettyPrint` to print the expression *without redundant parentheses*. That is, you will use the common conventions for precedence and associativity and keep parentheses only when they are needed to preserve the order of operations defined by the tree structure. For example:

```
> x = Add (Mul (Val 1) (Val 2)) (Val 3)
> prettyPrint' x
"1 * 2 + 3"

> x = Mul (Add (Val 1) (Val 2)) (Val 3)
> prettyPrint' x
"(1 + 2) * 3"

> x = Add (Val 1) (Add (Val 2) (Val 3))
> prettyPrint' x
"1 + (2 + 3)"
```

(4) Finally, add an instance of `Show` to the `ExprT` data type (notice that I did not include an automatic derivation for `Show`). Your instance sets `show` in such a way, so that an expression is printed in its prettiest form, i.e., in infix form without redundant parentheses:

```
> x = Mul (Add (Val 1) (Val 2)) (Val 3)
> x
(1 + 2) * 3
```

Note that the point here is not only to code the three interpretations, but to code them in a general way. Define your solution in such a way so that it captures and reuses common traversal functionality, and it is relatively easy to add new interpretations, e.g., prefix print, size, etc.

Problem 2. This goes back to the `LogMessage` parsing you did a couple of homeworks ago, but now, we'll try to parse in a more principled way.

We will make use of the following ADTs to represent messages. Note that I've changed the data types. Most notably, I got rid of the `Unknown` summand in `LogMessage` replacing it with a `Maybe` type, and I introduced wrappers around `Int` for the severity level and time stamp fields (i.e., the `newtype` declarations). In addition, there are two type classes, `ParseField` and `ParseRecord`, defining self-explanatory interfaces `parseField` and `parseRecord` respectively.

```
data MessageType = Info
                 | Warning
                 | Error Level
                 deriving (Show,Eq)

data LogMessage = LogMessage {
    mType :: MessageType, timeStamp :: TimeStamp, message :: String
} deriving (Show,Eq)

newtype Level = Level Int deriving (Eq, Show)

newtype TimeStamp = TimeStamp Int deriving (Eq, Show)

class ParseRecord a where
  parseRecord :: [String] -> Maybe a

class ParseField a where
  parseField :: String -> Maybe a
```

The task is (not very surprisingly) to define instances of `ParseField` for the `Level` and `TimeStamp` types, followed by an instance of `ParseRecord` for `LogMessage`. Use the respective `parseField` functions when parsing a log message record, then use `parseMessage` accordingly to parse the entire file. Starter code indicates where you'll need to add function definitions. The autograder will test functions `parseMessage` and `parse`. For example:

```
> parseMessage "I 2669 pci_hci"
Just (LogMessage {mType = Info, timeStamp = TimeStamp 2669, message = "pci_hci"})

> parseMessage "ABCDE"
Nothing
```

Note: Starter code for the first two problems is in `Ps4.hs`.

Problem 3. The final task is to implement `mergeSort` using a *foldable* representation of a list (`DivideList` in our case) and an appropriate *monoid* type (`MergesortList`). `MergesortList` is a wrapper around regular lists and all `MergesortList`s are *sorted lists*.

(1) Add appropriate implementation for `mappend` and `mempty` for `MergesortList`.

The following functions take advantage of the sortedness of `MergesortLists`:

(2) `minimum` returns the minimum of the list.

```
> minimum (ML [1,3,5])
Just 1

> minimum (ML [])
Nothing
```

(3) `numDistinct` counts the number of distinct values in the list:

```
> numDistinct (ML [1::Int,1,3,3,5])
3
```

(4) `count` counts the number of occurrences of distinct values:

```
> count (ML "abbcccdddd")
ML [('a', 1),('b',2),('c',3),('d',4)]
```

The following functions implement the mergesort functionality:

(5) `divide` takes a `DL` list and breaks it into two `DL` lists appropriately:

```
> divide (DL "abbcccdddd")
(DL {getDivideList = "abbcc"},DL {getDivideList = "cdddd"})
> divide (DL "abcde") == (DL  "ab",  DL "cde")
True
```

(6) `foldMap` defines the foldable instance of `DivideList`. Use `divide` to break the list into two parts, then process and combine accordingly.

```
No examples. You need to figure this out.
```

(7) Define `mergesort` using `foldMap`.

```
> mergeSort  [10,9..1]
[1,2,3,4,5,6,7,8,9,10]
```

Note: Starter code for Problem 3 is in `Ps4MergeSort.hs`. Submit `Ps4.hs` and `Ps4MergeSort.hs` to the Submitty autograder.

Haskell Style Guide. Adapted from Stephanie Weirich.

- **Write a type signature for every function.** We will be strict about this when grading. Hint: try writing the signature *before* writing the function. If you do write the function first, try deducing the signature and if this doesn't work, there is always the `:t` command.
- Make sure that your code produces no errors or warnings. Code with errors receives 0 on Submitty and we will mark down warnings during TA grading.
- Use consistent indentation.
- Do not use tab characters, use space for indentation. GHC should be flagging tabs, but nevertheless, be careful.
- No line should have more than **80 characters**.

- Use whitespace to make your code readable. Add whitespace on either side of binary operators, e.g., write `3 * n + 1` instead of `3*n+1`.
- Use descriptive names.
- Follow standard Haskell naming conversions: (1) use camelCase for compound names, and (2) use `x` and `xs` when you pattern-match lists.
- Use comments. Each function definition should be preceded by a comment.
- Comment should say what the function does, not how.
- Comments should be concise. Do not overcomment.
- Use full English sentences.
- Do not leave incomplete pattern matches. They will be marked down.
- Tuples, records and datatypes can be decomposed. You can also use the `@` operator if you need a reference to both the object and its components.
  For example, do not use this:

```
f arg1 arg2 = ...  where
   x = fst arg1
   y = snd arg1
   z = fst arg2
```

  Use this instead:

```
f (x,y) (z,_) = ...
```

- Combine nested case expressions.
  For example, do not use this:

```
case x of
      Red -> case y of
                Red  -> True
                Blue -> False
      Blue -> case y of
                Red  -> False
                Blue -> True
```

  Use this instead:

```
case (x,y) of
      (Red,   Red) -> True
      (Blue, Blue) -> True
      (   _,    _) -> False
```

- Use library functions, unless the assignment explicitly forbids them. Use Haskell's search engine Hoogle to look up library functions.