CS 115: Functional Programming, Spring 2016

Assignment 3: A Touch of Class

Due: *Monday, May 2, 02:00:00*

Coverage

This assignment covers the material up to lecture 8.

Submitting your assignment

Write all of your code for the first two sections in a single text file called `Lab3ab.hs`. The sparse matrix code should be a separate file called `SparseMatrix.hs`. Submit both files to csman.

Writing and testing your code

All of your Haskell functions, even internal ones (functions defined inside other functions), should have explicit type signatures.

All of your code should be tested using `ghci` with the `-W` (warnings enabled) command-line option. Any warnings will be considered errors, so make sure your code does not give any warnings.

We are supplying automated test scripts for part A and part C. The test script for part A is located here and the one for part C is located here. You can run them from `ghci` by putting them in the same directory as your code (which must have the correct module names *i.e.* `Lab3ab.hs` for part A and `SparseMatrix.hs` for part C). Then do this:

```
Prelude> :l ./Lab3ab.hs
*Lab3ab> :l ./SignedNatTests.hs
*SignedNatTests> main
[output is printed...]
```

to test part A, or:

```
Prelude> :l ./SparseMatrix.hs
*SparseMatrix> :l ./SparseMatrixTests.hs
*SparseMatrixTests> main
[output is printed...]
```

to test part C. After doing this, the tests will be run and the output will be printed. As in assignment 2, the test code relies on the `HUnit` and `QuickCheck` modules for unit testing and randomized testing, respectively. The test script cannot test for everything, but it should catch the most serious problems.

One trick you might want to try in case you have a test failure which is difficult to debug is to use the `trace` function from the `Debug.Trace` module. This function takes a string and a value, prints the string and returns the value. The trick is that it does this without being in the `IO` monad, so it uses the dreaded `unsafePerformIO` function. This is OK since all it does is print output before returning a value. For a particular value in a computation, you can do this:

```
-- ... somewhere inside an expression, you want to print the value of x ...
-- instead of just (x), write:
(trace ("x = " ++ show x) x)
```

Then, when this code is run, you will get a printout indicating what the value of `x` was at that point in the program. This is the Haskell way to emulate the primitive style of debugging in other languages using *e.g.* `print` statements.

Part A: Basic exercises

This assignment is mainly concerned with algebraic data types and type classes. Here we will do some exercises involving the `Eq`, `Ord` and `Num` type classes. There is also a test script for this section (see below), which you should run to see if your code works correctly before submitting your assignment.

1. **[10]** In class, we saw the definition of a simple algebraic data type for natural numbers:

   ```
   data Nat = Zero | Succ Nat
   ```

   Write out manual definitions of instances of the `Eq` and `Show` type classes for `Nat`.

2. **[5]** Now redo the previous problem, but have Haskell derive the `Eq` and `Show` instances for `Nat` automatically.

   You should make sure that the output of the automatically-generated `Show` instance is the same as the output of your manually-written one from the last problem (correct the implementation in the last problem if this isn't the case).

   Once you are satisfied with your answer to this problem, comment out the manual instance definitions in the previous problem. This is necessary so that the test scripts can function properly.

3. **[20]** Write out an explicit instance definition of the `Ord` type class for the `Nat` type. This can be done with a single method definition (for the `<=` operator) since all other `Ord` methods can be defined in terms of `<=`.

   Also, note that if we wanted, we could have Haskell derive the `Ord` instance for us. Sometimes this will give us the definition we want, and sometimes it won't. Will it work in this case? Why or why not? (Include your answer in a comment.)

4. **[20]** Let's say we would like to represent negative numbers as well as natural numbers, while keeping a unary representation. I'm not sure why we would want this, but let's say we do. We define this datatype:

   ```
   data SignedNat =
     Neg Nat | Pos Nat
   ```

```
  deriving (Show)
```

Assume that `Eq` and `Ord` instances for `Nat` exist (since you just defined them). Write out the `Eq` and `Ord` instance definitions for the `SignedNat` type. Could you just use automatically-derived definitions for the `Eq` and `Ord` instances? Write the answer in a comment.

5. **[60]** Write out the `Num` instance definition for `SignedNat`.

   Hint: A clean way to do this for longer method definitions is to implement all the methods as separate stand-alone functions *e.g.* `addSignedNat`, `mulSignedNat` *etc.* and then write the instance definition by referring to those functions. Also note that you don't have to implement all 7 `Num` methods; specifically, you only have to implement one of (`(-)`, `negate`) because the other can be defined trivially in terms of it.

   Hint 2: Beware of subtle problems involving zero.

   Hint 3: It would be useful to define some helper functions to do arithmetic operation on `Nat`s.

6. **[10]** Write a function called `signedNatToInteger` which converts a `signedNat` value to an `Integer`, as the name suggests. This is used by the test suite.

7. **[20]** Is there anything about the `SignedNat` datatype that strikes you as a bit ugly and redundant? Come up with a different datatype to represent positive and negative integers called `UnaryInteger` in the spirit of `Nat` (*i.e.* a unary encoding) that doesn't have this redundancy. Does it have any other problems? Can you think of a way to fix even those problems, possibly at the cost of increased complexity?

   The purpose of this exercise is to illustrate the fact that designing datatypes involves tradeoffs, and it's not always possible to do a perfect job. In particular, there are invariants that you may want Haskell to enforce that Haskell can't enforce.

   For this problem, we're not demanding that you come up with the exact datatypes we did, just that you try some alternatives and think about the problems that come up.

   Also, note that you don't have to implement the new datatype's operations. Just write up the `data` declarations and describe them in words in a comment.

8. **[10]** Write a definition of the factorial function called `factorial` which will work for any instance of `Num` and `Ord`. Make sure that it reports an error when given a negative number. Verify that it works for the `SignedNat` datatype by computing the result of `factorial (Pos (Succ (Succ (Succ Zero))))`. (Write the answer in a comment.)

   Hint: Remember that integer literals can represent any type which is an instance of the `Num` type class.

Part B: Operators and fixities

1. **[30]** In mathematics, an *associative* operator `OP` is one that has the property that `x OP (y OP z) == (x OP y) OP z`. In other words, in a chained operator expression of the form `x OP y OP z`,

where you put the parentheses doesn't matter; you can interpret the expression as either `x OP (y OP z)` or `(x OP y) OP z` and it will mean the same thing. Associativity in programming languages like Haskell doesn't usually mean this kind of associativity, because most programming languages (including Haskell) aren't powerful enough to express the concept of mathematical associativity. Instead, associativity (more technically known as *fixity*) is just a syntactic property of an operator which allows the parser to correctly interpret expressions of the form `x OP y OP z`. If the operator `OP` is left-associative, this will be interpreted as `(x OP y) OP z` (put the parentheses around the leftmost operator subexpression); if it's right-associative, it will be interpreted as `x OP (y OP z)` (put the parentheses around the rightmost operator subexpression), and if it's non-associative, simply disallow the expression altogether (*i.e.* require the programmer to explicitly put the parentheses in to indicate what the intended meaning is).

Haskell allows you to define your own operators and to set both the precedence of the operators (from 1 to 10) and their associativities/fixities (`infix` for non-associative, `infixl` for left-associative, or `infixr` for right-associative). The default associativity (in case you don't set it) is left-associative. Some operators have type signatures which require a particular associativity to work in chained operator expressions, while with other operators, you have a choice, though sometimes one choice is much better than another. For instance, the "cons" operator (`:`) must be right-associative for expressions like `1 : 2 : [3]` to work, while the addition operator (`+`) can be either right- or left-associative (it's actually left-associative, but it would work correctly either way). Some operators like the division operator (`/`) could theoretically be either right- or left-associative but are left-associative to conform to mathematical practice (because the `/` operator is not associative in the mathematical sense, so right-associative and left-associative interpretations of chained operators give different results). Some operators like the equality operator (`==`) are non-associative; you might wish that an expression like `x == y == z` means the same thing as `(x == y) && (x == z)` but it doesn't (in fact, it's a syntax error). This makes sense, because `x == (y == z)` requires that `x` but not `y` or `z` be a `Bool`, while `(x == y) == z` requires that `z` but not `x` or `y` be a `Bool`. The point is, you sometimes have to think a bit in order to set the associativity of a user-defined operator correctly.

Note also that Haskell allows you to set invalid associativities for your operators! Any operator can be declared `infix`, which simply means that chained operator expressions using that operator become parse errors and are rejected. Or an operator which should be declared `infixr` could be declared `infixl` instead, which means that chained operator expressions will be interpreted incorrectly, leading to type errors in some cases.

Also, operators in Haskell (including user-defined operators) can only be made from "operator characters", which means symbolic characters (not `A-Z` or `a-z` or `0-9` or `_`) and not including some characters with special functions (like parentheses, square brackets *etc*.).

1. For the first part of this problem, we will show you some new operators we've defined, and you should indicate what the associativity (fixity) should be. Here are your choices:

    1. If neither `infixr` or `infixl` will work with chained operator expressions (*i.e.* both of them lead to type errors), declare it to be `infix` (non-associative).

    2. If the operator can be declared either `infixr` or `infixl` without resulting in a type error for chained operator expressions, declare it `infixl` but add a comment stating that it could be `infixr` as well.

3. If the operator can be declared as only one of `infixr` or `infixl` without resulting in a type error for chained operator expressions, declare it to be the correct associativity (either `infixr` or `infixl`; whichever works).

When figuring out if a "chained operator expression" works for either the `infixr` or `infixl` associativity, choose whichever values you like as arguments to the operator; if any choice of arguments works, it's valid for that associativity. Give one example of a chained operator expression in that case. No examples are needed if the correct associativity is `infix` (*i.e.* non-associative).

Here are the operators (just a description, not a definition):

1. The operator `>#<` compares two scores and tells you the winner. For example, `51 >#< 40 = "First Player"`, and `21 >#< 21 = "Tie"`. It has type `Integer -> Integer -> String`.

2. The operator `+|` adds two integers and takes the last digit of their sum. For example, `7 +| 6 = 3`. It has type `Integer -> Integer -> Integer`.

3. The operator `&<` appends an integer to the end of a list. For example, `[1, 2] &< 3 = [1, 2, 3]`. It has type `[Integer] -> Integer -> [Integer]`, where `[Integer]` means a list of integers.

4. An operator `>&&` that conses an integer twice to the beginning of a list. For example, `1 >&& [2, 3] = [1, 1, 2, 3]`. It has type `Integer -> [Integer] -> [Integer]`.

2. Now consider an operator `+#` that adds two integers and tells you how many digits long their sum is. For example, `2 +# 800 = 3`, since `802` is three digits long. It has type `Integer -> Integer -> Integer`. What *could* its associativity be in order to allow chained operators to type check (give all answers that work). What *should* its associativity be? (*Hint:* Try writing out some examples.)

Part C: Miniproject: Sparse Matrices

In this section, you will be implementing a Haskell module which includes datatypes and functions to represent and compute with sparse matrices. A sparse matrix is a matrix which contains a large number of zero elements, which are not represented explicitly. Typically, the vast majority of the elements in the matrix are zeros. Sparse matrices arise naturally in a wide variety of applications.

Because of the large number of zero elements, computing with sparse matrices requires different algorithms than with non-sparse ("dense") matrices, or else the resulting code will be very inefficient. (Note that here we use the term "matrix" to include not only square matrices but matrices with any number of rows or columns.) In this module, you will be implementing sparse matrix addition, subtraction, multiplication, and negation. **While doing so, it is very important that you take advantage of the sparseness of the matrices in your implementation.** If you implement any of the functions by treating sparse matrices as dense matrices, the code will be considered invalid and you will get a zero on this section!

In addition to the challenge of implementing efficient functions for computing with sparse matrices, this miniproject has some other goals. You will learn to work with some of the data structures from the Haskell libraries, and you will learn to use Hoogle to search through the APIs. The total amount of code you need to write for this section is not large; our code is about 100 lines of non-comment code. (Note that we do expect you to write a reasonable amount of comments in your code explaining what your functions do.)

Sparse matrix representation

A particular element of each sparse matrix is identified by its row index and its column index. Row and column indices start at 1 (not 0!). Therefore, the basic representation for sparse matrices will be a map between pairs of `Integer`s (don't use `Int`s) and values, where the integers must be at least 1 and values can be any Haskell type which instantiates the `Num` type class. To represent the map you will use the `Data.Map` module and the `Map` datatype. Browse through the Hoogle documentation on `Data.Map` to learn how to use `Map`s. Note that you don't have to know how `Map`s are *implemented*, though you can browse the source code if you're curious; what's important here is that you know how to use them.

In addition to the map between pairs of `Integer`s and values, we need some extra bookkeeping information in each sparse matrix. Specifically, we have to store:

- The bounds of the matrix (number of rows and columns), as a pair of `Integer`s.
- A set of row indices, representing all rows that have nonzero elements.
- A set of column indices, representing all columns that have nonzero elements.

The row and column indices are stored to allow us to skip rows and columns that contain only zeros. This is particularly important when implementing multiplication. To implement sets, use the `Set` datatype contained in the `Data.Set` module. Again, see Hoogle for details.

Here is the definition of the sparse matrix datatype you should use:

```
import qualified Data.Map as M
import qualified Data.Set as S

data SparseMatrix a =
  SM { bounds     :: (Integer, Integer),  -- number of rows, columns
       rowIndices :: S.Set Integer,       -- row indices with nonzeros
       colIndices :: S.Set Integer,       -- column indices with nonzeros
       vals       :: (M.Map (Integer, Integer) a) }  -- values
  deriving (Eq, Show)
```

Note that this is a polymorphic type where the type `a` must be an instance of the `Num` type class. (We don't include that as a datatype context because Haskell deprecates adding contexts to datatypes, so this must be enforced by functions that act on the datatype.) You are not allowed to choose a different sparse matrix representation.

The `import qualified ... as ...` lines indicate that you should import the indicated modules and give a one-letter prefix to names in those modules. (It doesn't have to be one-letter, but that's what we're doing here.) So the `Set` datatype in `Data.Set` becomes `S.Set`, and the `Map` datatype in `Data.Map` becomes `M.Map`. This allows you to keep different namespaces separate without having to do too much

extra typing. This is important here because the `Set` and `Map` modules contain a number of functions with the same names.

One important point about the data representation is that after any sparse matrix operation (including the function to construct a sparse matrix!), you must make sure that:

- there are no zeros stored in the matrix (*i.e.* in the `vals` map)
- the `rowIndices` and `colIndices` fields are updated so that they contain all (and only!) the rows/columns with nonzero elements

Structure of the code

The Haskell code for the sparse matrix module will be called `SparseMatrix.hs` and will implement the `SparseMatrix` module. The first few lines will be as follows (add comments as you see fit):

```
module SparseMatrix where

import qualified Data.Map as M
import qualified Data.Set as S

data SparseMatrix a =
  SM { bounds     :: (Integer, Integer),  -- number of rows, columns
       rowIndices :: S.Set Integer,       -- row indices with nonzeros
       colIndices :: S.Set Integer,       -- column indices with nonzeros
       vals       :: (M.Map (Integer, Integer) a) }  -- values
  deriving (Eq, Show)
```

Following this, write the code for your function/operator definitions, along with any helper functions you may want to write.

Useful library functions

Here are the names of some library functions you may find useful. Look them up in Hoogle to find out how they work. You aren't required to use them if you don't need to. You may also find other functions in these modules to be useful; use whichever ones you like.

From `Data.Map`

- `elems`
- `filter`
- `filterWithKey`
- `findWithDefault`
- `fold`
- `fromList`
- `intersectionWith`
- `keys`
- `lookup`
- `map`
- `mapKeys`
- `toList`
- `unionWith`

Note that the functions called `map` and `filter` in this module are not the same as those called `map` and `filter` in the Prelude (*i.e.* that work on lists). This is another reason why we use `import qualified` when importing this module.

From `Data.Set`

- `fromList`
- `toList`

Note that these two functions are not the same as the functions with the same name from the `Data.Map` module. Again, this is why we use `import qualified` jhen importing these modules.

From `Data.List`

- `all`

Problems

1. **[30]** Implement a function called `sparseMatrix` which creates a sparse matrix from a list of index/value pairs and the array bounds.

   The type signature of this function will be:

   ```
   sparseMatrix :: (Eq a, Num a) =>
     [((Integer, Integer), a)] -> (Integer, Integer) -> SparseMatrix a
   -- sparseMatrix <list of index/element pairs> <bounds> -> sparse matrix
   ```

   Note that the indices in the index/value pairs are themselves pairs of integers. The function must check that the given bounds are valid (*i.e.* at least 1 each), and that all the index pairs given are within those bounds. If not, an error must be signalled.

   The function must not put zero values into the sparse matrix, even if some of the values in the input index/value pair list are zeros. Similarly, the function must store the row indices and column indices of only the rows/columns that have nonzero values.

   You can assume that the list of index/value pairs will not have duplicated indices. If there are duplicated indices, you can do whatever is most convenient.

   Note: The number `0` is a valid value of any type that is an instance of the `Num` type class.

2. **[30]** Implement a function called `addSM` which adds two compatible sparse matrices.

   The type signature of this function will be:

   ```
   addSM :: (Eq a, Num a) => SparseMatrix a -> SparseMatrix a -> SparseMatrix a
   ```

   If the two sparse matrices cannot be added (*i.e.* they don't have the same number of rows or columns) an error must be signalled. Be sure to adjust the `rowIndices` and `colIndices` after adding, and make sure that there are no zero elements in the matrix after adding.

3. **[10]** Implement a function called `negateSM` which negates a sparse matrix.

4. **[10]** Implement a function called `subSM` which subtracts two compatible sparse matrices.

   Hint: This function can trivially be defined in terms of the previous two.

5. **[60]** Implement a function called `mulSM` which multiplies two compatible sparse matrices.

   Recall that for two matrices to be compatible for multiplication, the number of columns in the first matrix must equal the number of rows in the second matrix. Your function must check for this and signal an error if the two matrices cannot be multiplied.

   Notes: This function is by far the trickiest one to implement of all the sparse matrix functions in this miniproject. We recommend that you write a helper function to multiply a row of one matrix (a row vector) by a column of the other (a column vector), and use that to construct the product matrix. Use the `rowIndices` and `colIndices` fields of the matrices to avoid multiplying empty rows or columns.

6. **[20]** Define three accessor functions for sparse matrices: `getSM`, which retrieves a value from a sparse matrix given the row and column, `rowsSM`, which returns the number of rows in a sparse matrix, and `colsSM`, which returns the number of columns in a sparse matrix.

   `getSM` has this type signature:

   ```
   getSM :: Num a => SparseMatrix a -> (Integer, Integer) -> a
   ```

   It retrieves the value at the index represented by the pair of integers. If this row/column location is invalid (out of bounds of the matrix), signal an error. If it's in bounds but there is no value stored in the matrix at that location, return a zero.

7. **[10]** Define operator shortcuts for the `addSM`, `subSM`, `mulSM`, and `getSM` functions defined previously.

   Use `<|+|>` as the operator for `addSM`, `<|-|>` for `subSM`, `<|*|>` for `mulSM` and `<!>` for `getSM`. The type signatures of these operators are identical to those of the corresponding functions (but you should write them out anyway!).

8. **[10]** Why doesn't it make sense to define the `SparseMatrix` datatype as an instance of the `Num` type class? Write your answer in a comment.

Things to watch out for

By far the biggest pitfall when writing sparse matrix code is writing code that inadvertently breaks the sparse matrix invariants *i.e.* where explicit zero elements are stored in the sparse matrix, or where the row/column indices stored are invalid (*e.g.* by containing indices for rows with no nonzero elements). Pay special attention to this when writing the code, especially the `sparseMatrix` function and the addition and multiplication functions.

Also, don't forget to run the test suite on your code (see above).