1 Introduction

The purpose of this assignment is to improve your skills writing Haskell functions over lists.

- For the purposes of this assignment, unless otherwise stated, don’t use any of the built-in library functions. — I want you to write all functions “from scratch”! Simple list manipulation functions (\texttt{head}, \texttt{tail}, :, ++, \texttt{length}) are OK to use, of course.
- Unless otherwise specified, all functions should use Haskell’s \texttt{pattern} or \texttt{guard} syntax, \textit{not} the \texttt{if-then-else} syntax!
- Unless otherwise specified, you don’t need to do any error checking or reporting.
- You may freely introduce auxiliary functions if that makes your program cleaner. Also, feel free to introduce local definitions (\texttt{where}-clauses) to make your code easier to read.
- You will be graded primarily on correctness and style, not on the execution efficiency of your code.
- Functions written for one problem may be freely used in subsequent problems. In fact, this is encouraged!
- All functions must be commented. At the very least, each function should start out with a description of what it does, what input parameters it takes, what output it produces, and an example of how it is invoked.
- All functions must have a function signature.
- You will need the set-manipulating functions in the next assignment!

2 Sorting

Our first task is to implement a sorting algorithm. We want our routines to be able to sort lists of arbitrary elements. We therefore provide a comparison function. The following ones can be used to sort integers (both ascending and descending order) and floats:

\begin{verbatim}
intCMP :: Int -> Int -> Ordering
intCMP a b | a == b = EQ
            | a < b = LT
            | otherwise = GT
\end{verbatim}
intCMPRev :: Int -> Int -> Ordering
intCMPRev a b | a == b = EQ
    | a < b = GT
    | otherwise = LT

floatCMP :: Float -> Float -> Ordering
floatCMP a b | a == b = EQ
    | a < b = LT
    | otherwise = GT

The constants EQ, LT, and GT are of type Ordering — an kind of “enumeration” type defined in the standard prelude.

1. Write a function sort3 that can sort lists of length 3 or less: [5 points]

   sort3 :: Ord a => (a -> a -> Ordering) -> [a] -> [a]
   sort3 cmp xs = ...

   > sort3 intCMP []
   []
   > sort3 intCMP [1]
   [1]
   > sort3 intCMP [3,2,1]
   [1,2,3]
   > sort3 intCMP [1,2,3]
   [1,2,3]
   > sort3 intCMP [1,1,1]
   [1,1,1]
   > sort3 intCMP [1,1,1,1]
   Program error: can't sort more than 3 elements!!!
   > sort3 floatCMP [3.0,2.0,1.0]
   [1.0,2.0,3.0]
   > sort3 intCMPRev [2,1,3]
   [3,2,1]

   For sort3 you should not use recursion! Instead, implement it by performing the minimal number of comparisons. For example, for a two element list you only need to make one or two comparisons, for a three element list you only need to make two or three comparisons. You can use any combination of guard syntax, if-then-else, or pattern syntax you like.

2. Implement a function pairCMP that compares two integer pairs: [5 points]

   type Pair = (Int, Int)

   pairCMP :: Pair -> Pair -> Ordering
   pairCMP (a,b) (c,d) = ...

   > pairCMP (1,0) (2,0)
   LT
   > pairCMP (3,0) (2,0)
   GT
Notice that you first compare the element of the two pairs. Only when these are equal do you also compare the second element of the pairs. You can use any combination of guard syntax, if-then-else, or pattern syntax you like.

Given `pairCMP` you’re now able to sort lists of pairs of integers:

```haskell
> sort3 pairCMP []
[]
> sort3 pairCMP [(1,2)]
[(1,2)]
> sort3 pairCMP [(3,1),(3,0),(2,1)]
[(2,1),(3,0),(3,1)]
```

3. Write a function `merge cmp xs ys` that merges two sorted lists into a sorted list: [5 points]

```haskell
merge :: Ord a => (a -> a -> Ordering) -> [a] -> [a] -> [a]
merge cmp xs ys = ...
```

```haskell
> merge intCMP [] []
[]
> merge intCMP [1,3,5] [2,4,6]
[1,2,3,4,5,6]
> merge intCMP [1,3,5] []
[1,3,5]
> merge intCMP [] [2,4,6]
[2,4,6]
> merge intCMP [1,2,2,3,4] [2,2,3,3,5,5]
[1,2,2,2,2,3,3,3,4,5,5]
```

4. Write a function `msort cmp xs` that returns the list `xs` sorted: [5 points]

```haskell
msort :: Ord a => (a -> a -> Ordering) -> [a] -> [a]
msort cmp xs = ...
```

```haskell
> msort floatCMP [5.0,9.0,2.0,4.5,1.0]
[1.0,2.0,4.5,5.0,9.0]
> msort floatCMP []
[]
> msort pairCMP [(4,6),(4,1),(4,6),(4,0),(1,2),(2,3)]
[(1,2),(2,3),(4,0),(4,1),(4,6),(4,6)]
```

You should use the `mergesort` algorithm to implement `msort`. I.e. at every recursive step split `xs` into two halves (hint: use `take` and `drop`), sort them recursively using `msort`, and then use `merge` to construct a sorted list.
5. Construct a faster version of \texttt{msort} called \texttt{fsort} that uses the technique from \texttt{sort3} to treat lists of three or fewer elements specially. I.e. whenever \texttt{fsort} gets down to short lists it won’t call itself recursively but rather sort them directly by calling \texttt{sort3}. To measure the “time” (actually, number of operations) that \texttt{msort} and \texttt{fsort} take, start \texttt{hugs} with the +s option: \[
\texttt{> hugs +s} \\
\texttt{> :load ass.hs} \\
\texttt{> msort intCMP [10,9..1]} \\
\texttt{[1,2,3,4,5,6,7,8,9,10]} \\
\texttt{(1499 reductions, 2103 cells)} \\
\texttt{> fsort intCMP [10,9..1]} \\
\texttt{[1,2,3,4,5,6,7,8,9,10]} \\
\texttt{(997 reductions, 1372 cells)}
\]

3 \ Set manipulation

Use recursion to write functions to manipulate \textit{sets}, where each set is implemented as a sorted list without duplicate elements. You can use the \texttt{sort} functions from the previous section if you want. You should implement the following functions:

\begin{itemize}
\item \texttt{makeSet cmp xs} returns a new list \texttt{ys} containing all the unique elements from \texttt{xs}, sorted. As in the sort functions above, each function takes a comparison function as argument. \texttt{isSet cmp xs} returns \texttt{True} if \texttt{xs} is sorted and contains no duplicate elements.
\item \texttt{setUnion cmp xs ys} returns the set of all elements that occur in \texttt{either xs or ys}. \texttt{setSubtract cmp xs ys} returns the set of all elements that occur in \texttt{xs but not in ys}.
\item \texttt{setCrossproduct ord xs ys} is a function of two sets that returns all pairs of elements where the first element is drawn from the first set and the second element is drawn from the second set. \texttt{setIsSubset cmp xs ys} returns \texttt{True} if all the elements of \texttt{xs} occur in \texttt{ys}.
\end{itemize}

Sometimes it’s useful to compute how similar two sets are to each other, or how much of one set occurs in another. We use the following definitions:
Definition 1 (Similarity and Containment) The similarity \( \text{similarity}(p, q) \) between two sets \( p \) and \( q \) is defined as
\[
\text{similarity}(p, q) = \frac{|p \cap q|}{|p \cup q|}
\]
where \( |q| \) is the number of elements in \( q \). Similarly, the containment \( \text{containment}(p, q) \) of \( p \) within \( q \) is defined as
\[
\text{containment}(p, q) = \frac{|p \cap q|}{|p|}
\]
To illustrate, consider these two sets:
\[
p = \{1, 3, 7, 8, 9, 11\} \\
q = \{2, 7, 9, 11\}
\]
Their similarity is given by
\[
\text{similarity}(p, q) = \frac{|\{1, 3, 7, 8, 9, 11\} \cap \{2, 7, 9, 11\}|}{|\{1, 3, 7, 8, 9, 11\} \cup \{2, 7, 9, 11\}|} = \frac{|\{7, 9, 11\}|}{|\{1, 2, 3, 7, 8, 9, 11\}|} = \frac{3}{7}.
\]
The fraction of \( p \) that’s contained within \( q \) is given by
\[
\text{containment}(p, q) = \frac{|\{1, 3, 7, 8, 9, 11\} \cap \{2, 7, 9, 11\}|}{|\{1, 3, 7, 8, 9, 11\}|} = \frac{|\{7, 9, 11\}|}{|\{1, 3, 7, 8, 9, 11\}|} = \frac{3}{6}
\]
and the fraction of \( q \) that’s contained within \( p \) is given by
\[
\text{containment}(q, p) = \frac{|\{2, 7, 9, 11\} \cap \{1, 3, 7, 8, 9, 11\}|}{|\{2, 7, 9, 11\}|} = \frac{|\{7, 9, 11\}|}{|\{2, 7, 9, 11\}|} = \frac{3}{4}.
\]
Here are some more examples:

> setUnion intCMP [3,4,5,2] [1,2,3]
Program error: arguments must be sets
> makeSet intCMP [3,3,3,3,1,1,1,3,2,2] [1,2,3]
> isSet intCMP []
True
> isSet intCMP [1]
True
> isSet intCMP [1,1]
False
> isSet intCMP [1,2,1]
False
> setUnion intCMP [1,2,3] [3,4,5] [1,2,3,4,5]
> setIntersect intCMP [1,2,3] [3,4,5] [3]
> setSubtract intCMP [1,2,3] [3,4,5] [1,2]
> setUnion intCMP (makeSet intCMP [3,3,2,2,4]) (makeSet intCMP [3,3,5,5,1,1,1]) [1,2,3,4,5]
> makeSet intCMP [3,3,2,2,4] [2,3,4]
> setSimilarity intCMP (makeSet intCMP [1,3,7,8,9,11]) (makeSet intCMP [2,7,9,11]) 0.428571
> setContainment intCMP (makeSet intCMP [1,3,7,8,9,11]) (makeSet intCMP [2,7,9,11]) 0.5
> setUnion pairCMP (makeSet pairCMP [(1,2),(3,4),(1,2)]) (makeSet pairCMP [(1,2),(0,9)]) [(0,9),(1,2),(3,4)]
> setIsSubset intCMP (makeSet intCMP [3,3,2,2,4]) (makeSet intCMP [3,2,5,6]) False
> setIsSubset intCMP (makeSet intCMP [3,3,2,2,4]) (makeSet intCMP [3,2,5,6,4]) True
> setCrossproduct intCMP [] [4,5,6] []
> setCrossproduct intCMP [1,2,3] [] []
> setCrossproduct intCMP [1,2,3] [4] [(1,4),(2,4),(3,4)]
> setCrossproduct intCMP [1,2,3] [4,5] [(1,4),(1,5),(2,4),(2,5),(3,4),(3,5)]
> setCrossproduct intCMP [1,2,3] [4,5,6] [(1,4),(1,5),(1,6),(2,4),(2,5),(2,6),(3,4),(3,5),(3,6)]

HINT: Use fromIntegral x to convert from an integer to a floating-point number.

NOTE: All functions (except setIs and makeSet) should check that their arguments are indeed sets and throw an “exception” (using the built-in error function) if they’re not. An error message should be produced that says "arguments must be sets".

### 4 Computing Primes

Define a function sieve xs which computes a list of prime numbers using a Eratosthenes’ sieve. For example, sieve [2..10] should return the list [2,3,5,7].

The method is as follows:

1. start with a list of numbers beginning with 2, for example [2,3,4,5,6,7,8,9,10].
2. The first number in the list is prime. Remove all its multiples. In this case we get [3,5,7,9].
3. Repeat the previous step: 3 is prime, and after ‘sieving’ out the multiples we are left with [5,7].
4. Repeat the step again: 5 is prime, and sieving leaves [7].
5. Do it again: 7 is prime, and sieving leaves [].
6. When no numbers remain, we’ve found all the primes in the given range.

First implement a function sieve n xs which returns the elements of xs that are not multiples of n:
siever :: Int -> [Int] -> [Int]
siever n xs ...

> siever 2 [3..10]
[3,5,7,9]

Hint: use this helper function:

relprime p n = n `mod` p > 0

Using siever, implement the sieve function:

sieve :: [Int] -> [Int]
sieve xs ...

> sieve [2..10]
[2,3,5,7]
> sieve [2..20]
[2,3,5,7,11,13,17,19]
> sieve [2..]

5 Submission and Assessment

The deadline for this assignment is noon, Tue Sep 22. It is worth 5% of your final grade.

You should submit the assignment electronically using d2l. The file you upload should be named ass.hs.

| Don’t show your code to anyone, don’t read anyone else’s code, don’t discuss the details of your code with anyone. If you need help with the assignment see the instructor or the TA. |