CSE P505, Autumn 2016, Homework 5
Due: Wednesday 7 December 2016, 11:00PM

- This assignment covers Haskell, laziness, the IO monad, and typeclasses.
- We have provided a (very, very simple) Makefile that creates four programs: `prob12` (problems 1 and 2), `sort` (problems 3 and 4), `sortML` (problem 3), and `typeclass` (problem 5). It is also fine to compile directly at the command line.
- Instructions regarding `sortML.ml` apply to `sortML.fs` as well.
- Turn in your solution via the "Turn-in" link on the course website. Include `sortML.ml` (or `sortML.fs`), `prob12.hs`, `sort.hs`, and `typeclass.hs`. Do not zip, tar, or otherwise archive your submission.
- Understand the course policies on academic integrity (see the syllabus) and challenge problems.

1. (Haskell warmup) Two types, `InttreeT` and `InttreeC` have been provided, along with `insertT`, `fromListT`, `insertC`, and `fromListC`. The types `InttreeT` and `InttreeC` have nearly identical representations, except `InttreeT` uses tupled data constructors (like OCaml/F#'s) and `InttreeC` uses Haskell curried data constructors.

   Your solutions should closely match analogous functions you wrote in OCaml/F# in Homework 1.

   (a) In an English comment in the code, explain why changing the lines in the provided code `insertT (NodeT(j,l,r)) i =` and `insertC (NodeC j l r) i =` to `insertT NodeT(j,l,r) i =` and `insertC NodeC j l r i =` respectively parses but leads to type-checking errors.

   (b) Implement `prodT :: InttreeT -> Int` to return the product of all the numbers in its argument.

   (c) Implement `mapT :: (Int -> Int) -> InttreeT -> IntTreeT` to produce a tree with the same shape as its second argument with the int at each position the result of applying the first argument to the int at the same position in the second argument.

   (d) Implement `negateAllT :: IntTreeT -> IntTreeT` using `mapT`. It produces a tree of the same shape where each int is replaced with its negation. Note the result is therefore not properly sorted, but do not do anything about that.

   (e) Implement `prodC`, `mapC`, and `negateAllC` in corresponding fashion for `IntTreeC`. (Just write similar code; do not worry about sharing code as these functions are short.)

2. (The IO Monad) In `prob12.hs`, implement the `until_pair` function to have type `(a -> a -> Bool) -> IO a -> IO [a]`. This function produces an `IO [a]` that, when performed, repeatedly performs the `IO a` action and puts the results in a list in order. That is, the first element is the result of the first action execution, the second element is the result of the second action execution, and so on. The action stops when the most recent two results (of type `a`) produce `True` via the `a -> a -> Bool` argument. The resulting list will therefore always have at least two elements in it. The most recent result should be the first argument to the filtering function, and the second-most recent result the second.

   Hints:

   - The reference solution is only 7 lines.
   - You can test your code using the `getList` and commented out lines of the `main` function in `prob12.hs`. This main function uses the `until_pair` combinator to read a series of numbers (one per line) from standard input until the two most recent numbers are identical.
   - It is relatively expensive to access the last element of a list. We recommend instead to use an inner loop that builds a list in reverse order and then reverse the final list once at the end (similar to your implementation of `interpSmall` in Homework 2).

3. (Laziness and sorting and performance) You will implement two sorting algorithms in Haskell and OCaml (four total implementations but it's just sorting :-)) and use them to answer a "first-k-elements" query.

   (a) Implement `selectionSort` and `mergeSort` in `sort.hs`.

   (b) Implement `selectionSort` and `mergeSort` in `sortML.ml`.

   (c) Run `sort random_numbers_10k` and note the times for each sorting algorithm. Next, run `sortML random_numbers_10k` and note the result times. The ML and Haskell implementations should have very different performance characteristics. Why is the fastest implementation the fastest and why is the slowest implementation the slowest? Do you expect to see the same performance difference for a last-k element query? Put your answer in a comment in `sort.hs`. (Depending on your computer, you may need to adjust the list lengths to "see" non-trivial timings, but you also need to avoid OCaml stack overflow — see the first challenge problem.)

   (d) In a comment in `sort.hs`, explain why sorting, even for a first-k elements query, would never work on an infinite list or data structure.

   Hints:

   - As a refresher, the Wikipedia pages for merge and selection sort have pseudo-code descriptions of these algorithms (`https://en.wikipedia.org/wiki/Merge_sort` and `https://en.wikipedia.org/wiki/Selection_sort`). In particular, do *not* implement insertion sort.
   - Your implementations do *not* have to be stable.
   - In principle, selection sort can look for the minimal or maximal element in the unsorted portion of the list. You should look for the minimal element in your implementation.
   - For merge-sort over a linked list, it is easiest when "splitting" to, via simple recursion, put the odd-position elements in one list and the even-position elements in another list.
   - The reference Haskell solution uses 9 lines for `selectionSort`, and 12 lines for `mergeSort`.
   - For rapid debugging, test your code using the `random_numbers_small`.
   - As a last resort, you can use the `Debug.Trace` module (`https://hackage.haskell.org/package/base-4.9.0.0/docs/Debug-Trace.html`) for "printf debugging" in Haskell. Be sure to delete any debug code you add before submission.

4. (More IO) Implement `doAndPrintSorted :: IO()` in `sort.hs`. The IO action should prompt the user for a number, output a sorted list of all the numbers entered so far, and then prompt the user for another number. This loop ends when the user enters anything that is not a number (including an empty line).

   Hints:

   - You may use one of the sorting methods implemented for problem 3.
   - Use the `rInt` function we provided to read numbers from standard input. Its type signature is `IO (Maybe Int)`, i.e., an action that produces a `Maybe Int`. `Maybe` is analogous to OCaml's `option` type constructor: `Nothing` is like `None` and indicates non-numeric input, and `Just n` is like `Some`, and contains the number entered on the command line.
   - The reference solution is 7 lines.
   - `putStrLn (show l)` will print a string representation of the list `l` to standard output. This expression has type `IO ()`.
   - You can test your solution by editing `main` to be `main = doAndPrintSorted`.

5. (Typeclasses) This problem explores using typeclasses to overload operators and better reuse algorithms in a typesafe manner. In most languages (including Haskell), tuples are sorted lexicographically by tuple elements. That is, `(1,5) < (5,1)` and `(1,2) < (1,3)`. This is similar to how we sort English words. In `typeclass.hs`, we have defined the type `PairRL` which is a pair that we wish to order *reverse lexicographically* by tuple elements. That is, `(5,1) < (1,6)` and `(1,3) > (1,2)`.

   (a) We will use typeclasses to overload the Haskell comparison operators to respect this ordering. To do this, uncomment the two typeclass instance stubs and complete the where clauses. For `Eq` typeclass, provide an implementation of the equality operator (`==`) and in `Ord` the `<=` operator. (Optional language design question: with just these two operators Haskell can derive *all* of the comparison operators. Why? Can you think of other possible operators?)

   After implementing these typeclasses, remove the `undefined` token in the main function and uncomment the code beginning with `let a = ...`. If you implemented your operators correctly, you should see `[(1,1),(3,1),(1,2),(3,5),(6,11)]` when you run `typeclass`.

   Hints:

   - Both operators are very simple: do not overthink things.
   - Operators may be defined using infix syntax: `a op b = expr` is equivalent to `let (op) a b = expr`.
   - The documentation for the `Eq` and `Ord` typeclasses may be helpful: `https://hackage.haskell.org/package/base-4.9.0.0/docs/Data-Eq.html` `https://hackage.haskell.org/package/base-4.9.0.0/docs/Data-Ord.html`

   (b) To keep matters simple in the earlier sorting problems, we provided explicit type annotations of type `[Int] -> [Int]`. If you remove these annotations, do your sorting algorithms work for any ordered type? Could you use your algorithms over lists of reverse-lexicographic pairs? Answer in a brief comment in `typeclass.hs`.

6. (**Challenge Problem:** Don't Blow Your Stack) Extend your ML implementation of `selectionSort` to handle at least 1 million element lists without a stack overflow. As before, sorting does not have to be stable. You'll need to make all passes over the list of elements tail recursive and the main algorithm must remain selection sort and be $O(n^2)$.

   Hints:

   - One solution is to use a pair of lists in one of the passes.
   - Alternatively, port the entire sorting algorithm to CPS.

7. (**Challenge Problem:** Monads Go Nuts)

   - Read the classic paper, *The Essence of Functional Programming*, available at `http://www.eliza.ch/doc/wadler92essence_of_FP.pdf`.
   - Implement the monadic interpreter described in the paper. Make sure your interpreter is generic over "which monad" it is run in.
   - Create a new instance of the monad typeclass such that when your interpreter is run "in" that monad, it behaves like a normal lambda-calculus interpreter *except* that if any subexpression evaluates to 42 (e.g., there is a constant holding 42 or the result of an addition is 42), then the result of the entire program is 42.