universität innsbruck

Advanced Functional Programming	WS 2025/2026	LVA 703139
Exercise Sheet 10, 10 points		Deadline: Tuesday, January 6, 2026, 4pm

- Solve the tasks in file `Exercise10.hs` and upload only this file in OLAT.
- Mark the solved exercises in OLAT.
- Your modified `Exercise10.hs` files must compile with `ghci` without error messages.
- Performance measurements will be performed via:

  ```
  cabal run Exercise10 -- numbers 5000000
  cabal run Exercise10 -- sort optimizedInput hybridSort +RTS -N4 -RTS
  ```

Task 1 *Hybrid Sorting* **4 p.**

Instead of performing parallelization of a single sorting algorithm such as quicksort, an alternative is to split the list into *nc* sublists (where *nc* is the number of cores), each sublist is sorted in parallel using sequential quicksort, and then the merge-operation of mergesort is applied.
Implement and evaluate this idea.
Hint: `parList` and `spine` might be useful.
For testing, the first commands generates a file with random numbers, and the second command invokes one of the selected algorithms using 4 cores.

```
cabal run Exercise10 -- numbers 5000000
cabal run Exercise10 -- sort seqInput {qsortSeq|qsortPar|hybridSort} +RTS -N4 -RTS
```

Task 2 *Parallel File Reading* **4 p.**

The current code for running a parallel sorting algorithm has a significant sequential phase, namely:
`input <- lines <$> readFile sortFile`
Figure out whether this part can be made more efficient by using parallelism, too. To this end, implement and evaluate some of the following ideas:

- reading the file is still done sequentially, but `lines` is re-implemented in a parallel way
- both reading and splitting the input into lines is done in parallel

If you want to perform the file-read operation in parallel, then the `ByteString` library might be useful, which is already imported in the template. It provides constant time operations to split (`BSC.splitAt`) a `ByteString` at any position, and there is also constant time random access to any character in the `ByteString` (`BSC.index`).
For testing, run

```
cabal run Exercise10 -- sort {seqInput|parListInput|optimizedInput} hybridSort +RTS -N4 -RTS
```

with different number of cores to test your input reader.

Task 3 *Concurrent Dictionaries* **2 p.**

We consider the task to create a concurrent dictionary, based on a standard immutable dictionary implementation. The aim is to gain efficiency by releasing `MVar`-locks early on.
In detail:

- Create a datatype for a concurrent map and implement `empty`, `insert`, and `lookup`. In the template, you already find the import of `Data.Map.Strict` to get access to a purely functional and strict implementation of maps.
- Test your implementation with the provided application code. To run the application, execute something like

  ```
  cabal run Exercise10 -- cmap 1000 100000 100 +RTS -N2 -s -RTS
  ```

  where 1000 is the number of elements that are inserted into the map per thread, 1000000 is the computation cost of each element, 100 is number of threads, and -s prints statistic information.

  In the statistics, the total time reports computation time and real time (elapsed).
- Also run

  ```
  cabal run Exercise10 -- cmap 10000 1 500 +RTS -N2 -s -RTS
  ```

  and look at the memory consumption. Can you observe some unwanted effect and integrate countermeasures?