Version 1.2 ***due:*** *Mon, Sep 30 @ 11:59pm*

1 Introduction

A Turing Test is an interaction with an agent designed to determine whether that agent is human or machine. For this lab you will program thousands of monkeys on typewriters to generate text which passes Turing Test spam filters.

How can a machine fool a text-based Turing Test? It can't just deliver canned responses; there must be some generativity and randomness. But it must *sound like* human language. One way to generate text is based on k-grams, or chunks of text k words long. After selecting k words, a random choice for the $k+1^{th}$ word weighted by the frequency of that $k+1$-gram in the input body of text, known as the *input corpus*. You will write a library for parsing input to generate realistic, Turing Test-passing text.

2 Files

After downloading the assignment tarball from Autolab, extract the files from it by running `tar -xvf babblelab-handout.tgz` from a terminal window. You should see the following files in the newly created directory:

1. `babblelab.pdf`
2. `bobbfile.py`
3. `config.py`
4. `Makefile`
5. `lib/`
6. `sources.cm`
7. `data/`
8. `support/`
9. * `MkSeqUtil.sml`
10. * `MkTableKGramStats.sml`
11. * `MkBabble.sml`
12. * `Tests.sml`
13. * `corpus.txt`

You should only modify the last 5 files, denoted by *. Additionally, you should create a file called `written.pdf` which contains the answers to the written part of the assignment.

3 Submission

There are two ways to submit your solutions. Before submitting, ensure that `written.pdf` exists and is a valid PDF document.

The first, and easiest way to submit is by running `make` from within the `babblelab` directory. This will automatically package your submission into a tgz archive, and submit it to Autolab. You must be on the unix.andrew machines for this to work.

Important note: If you submit this way, you will still need to visit the Autolab website to view the results of the tests that Autolab runs against your code (which will contribute to part of your grade for this assignment). It is important to always do this for every lab to make sure that your code runs as you expect it to.

If you would rather submit through the Autolab website, run `make package` from the `babblelab` directory. This should produce a file called `handin.tgz`. Open the Autolab webpage and submit this file via the "Handin your work" link.

4 SEQUENCE_UTIL

This lab requires a few functions to extend the `SEQUENCE` library, which will be contained in the functor `MkSeqUitl` in `MkSeqUtil.sml`. You'll find these functions useful when programming your monkeys. Two of these have been written for you, but you will need to write a third before you start babbling.

4.1 String Tokens

Recall from recitation 4 the string parsing functions `String.fields` and `String.tokens`. We've implemented a parallel version of `String.tokens` for use in your `BABBLE` application.

```
val tokens :  (char -> bool) -> string -> string seq
```

`tokens cp s` evaluates to a sequence of maximal non-empty subtrings (tokens) of `s` containing no characters c for which $cp(c)$ evaluates to true. For this application, we'll use `(not o Char.isAlphaNum)` for cp, but `tokens` will work correctly with any choice of cp. `tokens cp s` has $O(|s|)$ work and $O(\log|s|)$ span, assuming the application of cp has constant work and span.

4.2 Histograms

A *histogram* estimates the probability density function of some variable. Specifically, we define

```
type 'a hist = ('a * int) seq
```

We associate each value with an integer which gives the frequency of that value in a given distribution. We've provided a function to build histograms:

```
val histogram :  'a ord -> 'a seq -> 'a hist
```

in `MkSeqUtil`. `histogram cmp s` evaluates to a sequence of (k, n) pairs, where k is a unique key in the input sequence `s`, and n is the number of occurences of k in `s`. `histogram cmp s` has $O(|s| \log |s|)$ work and $O(\log^2 |s|)$ span.

Task 4.1 (10%). Implement the function

```
val choose :  'a hist -> real -> 'a
```

in the functor `MkSeqUtil` in `MkSeqUtil.sml`. If $0 \leq r \leq 1$, `choose hist r` should evaluate to the value at r from the cumulative distribution that corresponds to the histogram $hist$. If r is not in the range $[0, 1]$ or $hist$ is empty, you should raise an exception. For full credit, `choose` should have $O(|hist|)$ work and $O(\log |hist|)$ span. Our solution is about 10 lines long. As an example, consider the histogram $h = \langle(\text{“a”}, 3), (\text{“b”}, 5), (\text{“c”}, 2)\rangle$. The cumulative distribution function table is:

“a”	0.3
“b”	0.8
“c”	1.0

Then `choose h 0.2` should return “a”, `choose h 0.8` should return “b”, and `choose h 0.95` should return “c”. That is, it should return the key with least value above or equal to r.

4.3 Testing

As in previous labs, you may use the `Tester` structure to test your implementations. `Tester` will pull test cases from the `testsChoose` list inside of `Tests.sml`:

```
Tester.testChoose ();
```

Task 4.2 (5%). Add test cases for `choose` in `Tests.sml`. Be sure to consider edge cases, as well as to include some longer, more complex tests. Using comments , briefly explain the motivation behind each test – why is the test useful?

5 K-Gram Stats

Read over the `KGRAM_STATS` signature defined in `support/KGRAM_STATS.sig`. This defines an ADT for storing a certain piece of information about k-grams in a corpus (exactly what information this is will be described below), and three functions:

- `val makeStats : string -> int -> kgramstats`
  `makeStats corpus maxK` evaluates to a value of type `kgramstats` containing information for all k-grams of length up to the given $maxK$ in the input corpus.
- `val lookupExts : kgramstats -> kgram -> token hist`
  `lookupExts stats kgram` evaluates to a histogram containing all of the tokens tok that appear immediately after the input $kgram$ along with the number of times this occurs in the corpus (i.e. it is a sequence of (tok, n) pairs where tok is a unique token and n is the number of times tok appears immediately after the input $kgram$.
- `val maxK : kgramstats -> int`
  `maxK stats` should return the integer argument (`maxK`) of the call to `makeStats` that was used to create `stats`.

You will now implement a data structure to support the `KGRAM_STATS` ADT for storing statistics on a corpus, which you can then use to answer queries about any given k-gram. For a given constant $maxK$, the `kgramstats` type must store information for all k-grams that appear in the corpus for $0 \leq k \leq maxK$.

Task 5.1 (2%). Define the abstract `kgramstats` type and explain in a comment why you chose the type you did. *Hint:* You should make use of the functor argument `structure T : TABLE where type Key.t = string Util.Seq.seq`. Specifically, `'a T.table` defines a table with keys of type `string seq` and values of type `'a`.

Task 5.2 (22%). Implement the function `makeStats` (described above) in the functor `MkTableKGramStats` in `MkTableKGramStats.sml`. For full credit, `makeStats corpus maxK` should have $O(n \log n)$ work and $O(\log^2 n)$ span, where n is the number of tokens in $corpus$, and assuming the constant $maxK$ is small.

You should split the input corpus into tokens using the delimeter function `(not o Char.isAlphaNum)`. You might also find `SEQUENCE collate` useful. For asymptotic analysis, you may assume that string comparisons are constant. As a reference, our solution is about 15 lines long.

Task 5.3 (4%). Implement the function `lookupExts` (described above) in the functor `MkTableKGramStats` in `MkTableKGramStats.sml`. For full credit, `lookupExts` should have $O(\log n)$ work and span, where n is the the size of the `kgramstats` type.

Task 5.4 (2%). Implement the function `maxK` (described above) in the functor `MkTableKGramStats` in `MkTableKGramStats.sml`. For full credit, `maxK` should have $O(1)$ work and span.

5.1 Testing

Again, you should use `Tester` to test your code. This time, `Tester` will test your implementation of `makeStats` and `lookupExts` in one go (because you have defined your own `kgramstats` type, it is abstract, so we cannot test either of these functions independently):

```
Tester.testKGramStats ();
```

Task 5.5 (5%). Write test cases for `MkTableKGramStats` in `Tests.sml`. You should see some existing tests which use the corpus in the file `corpus.txt`. `corpus.txt` will get handed in, so feel free to change it, but any tests dealing with any other corpus will not be graded. You may test with a corpus in a different test file, but we will just ignore those test cases. Be sure to consider edge cases, as well as to include some longer, more complex tests. Using comments, briefly explain the motivation behind each test – why is the test useful?

6 Babble

Using the `KGRAM_STATS` abstract data type, it is easy to write an algorithm that generates text that is statistically similar to an input corpus. For example, to write pseudo-Shakespeare, you would compute statistics of Shakespeare's texts and use it to generate new masterpieces. We call this the babble problem.

To help with this problem, we present to you the `RANDOM210` signature. In particular, you will find the following functions exported by the signature useful:

```
val randomRealSeq :  rand -> ((real * real) option) -> int -> real Seq.seq
val randomIntSeq :  rand -> ((int * int) option) -> int -> int Seq.seq
```

Given a seed r, `(randomRealSeq r NONE n)` generates a sequence of n random `real` values $\in_{\mathbb{R}} [0,1]$. `(randomIntSeq seed (SOME (i, j)) n)` generates a sequence of n random `int` values $\in_{\mathbb{Z}} [i,j)$.

6.1 Implementation

Task 6.1 (10%). Implement the function

```
val randomSentence :  kgramstats -> int -> Rand.rand -> string
```

in the functor `MkBabble` in MkBabble.sml. `randomSentence stats n seed` should generate a sentence with $n > 0$ words given the `stats` of some corpus and a random `seed`. For full credit, `randomSentence` should have $O(n(W_{\texttt{lookupExts}} + W_{\texttt{choose}}))$ work and $O(n(S_{\texttt{lookupExts}} + S_{\texttt{choose}}))$ span, assuming that the words in the corpus have constant length. The output string should end with a period and not have any leading spaces. You might find `String.concatWith : string -> string list -> string` to be useful.

Task 6.2 (5%). Implement the function

```
val randomDocument :  kgramstats -> int -> Rand.rand -> string
```

in the functor `MkBabble` in MkBabble.sml. `randomDocument stats n seed` should generate a document of $n > 0$ sentences given the `stats` of some corpus and a random `seed`. Each sentence should have a random length between 5 and 10 words. For full credit, `randomDocument` should have $O(nW_{\texttt{randomSentence}})$ work and $O(n + S_{\texttt{randomSentence}})$ span, assuming that each sentence has constant length. Each sentence should be separated with a space, and there should be no leading or trailing spaces. Again, you should use `String.concatWith` which you may assume to have work and span linear in the length of the input `string list`.

6.2 Babbling

For this section, you will not write test cases or test your solution against a reference solution. However, if you have correctly implemented the `MkTableKGramStats` and `MkBabble` functors, you are now ready to launch your monkeys and generate the complete works of Shakespeare (among other texts)!

The `Tester` structure provides a function that takes an `inputFile : string` (the corpus) and prints 10 sentences of babble to the screen (albeit in a not-so-nicely formatted way). Note that this will require your implementation of `kgramstats` and `choose` to be correct (using our reference implementations causes babble to be far too slow):

```
Tester.babbleFromFile "data/shakespeare.txt";
```

There are sample corpus files located in `data/` which you should use to test your implementation. Feel free to come up with your own corpora. Sharing is caring, so please do post them on Piazza!

7 SkylineLab Reloaded

After you did such a good job recreating the Pittsburgh skyline recently, we're back with a related task: we want you to find its local maxima (e.g. the US Steel Tower, BNY Mellon Center, Cathedral of Learning) and local minima (e.g. Neville Street, poor grad students' flooded basements.)

Given a sequence s of integers, an element at position i is a *local minimum* if $s_i < s_{i-1}$ and $s_i < s_{i+1}$. Similarly, an element at position i is a *local maximum* if $s_i > s_{i-1}$ and $s_i > s_{i+1}$. The first and last elements are local minima or maxima if they are less or greater than their only neighbor (respectively).

For example, in the sequence, the local maxima are marked with a plus and local minima are marked with a minus sign: $\langle 5^+, 4, 3^-, 4, 5, 6, 7^+, 6, 5, 2^-, 3, 4^+ \rangle$. **For this question, assume no two adjacent elements are equal.**

We want a function `extrema` that computes local minima and maxima of a sequence of integers with

optimal work and span [1]. `extrema` should map an element k to `SOME k` if k is a local minimum or maximum, and to `NONE` otherwise. For example, `extrema` $\langle 5^+, 4, 3^-, 4, 5, 6, 7^+, 6, 5, 2^-, 3, 4^+\rangle$ should evaluate to

$\langle SOME(5), NONE, SOME(3), NONE, NONE, NONE, SOME(7), NONE, NONE, SOME(2), NONE, SOME(4)\rangle$.

As an added challenge, we are not giving you cost bounds. We want you to think about the problem at hand and reason about how efficiently it can be solved in parallel.

Task 7.1 (5%). Estimate the work and span (in big-Θ notation) of an optimal parallel implementation of the function `extrema`.

Task 7.2 (10%). Give pseudocode for a parallel implementation of `extrema` that meets the cost bounds you gave in **7.1**. You need not prove that the implementation meets the cost bounds. You may assume that you have access to the function `cmp`, given below.

```
fun cmp f (i,j) =
     (j < 0) orelse (j > length s - 1)   (* first/last true *)
     orelse f (nth s i, nth s j)
```

Task 7.3 (5%). Informally justify why you believe no more efficient algorithm exists. Proving this formally can be quite difficult, but you can often give an informal but convincing argument by appealing to intuition about the problem. For example, a justification might take the form of "If a better solution existed, it would let us do X, which is known to be impossible" or "Any algorithm that solves this problem must at least do Y, which takes $O(Z)$."

[1] In this class, we prioritize low work over low span. This means that we want you to write an algorithm with the best possible span *among those with the best possible work*.