Homework 3

15-210: Parallel and Sequential Data Structures and Algorithms (Fall'12)

`tag` — Version 1.0 **due:** *mon, sep 24 @ 11:59pm*

Disclaimer:
We will not grade non-compiling code.

1 Introduction

In this assignment you will program thousands of monkeys on typewriters to generate Shakespeare and pass spam filters.

With evil robots ubiquitous in our lives, it is ever more important to understand their attempts to thwart us. Thus you will play the role of the evil robots trying to pass Turing Tests. A Turing Test is an interaction with an agent designed to determine whether that agent is human or machine. CAPTCHAs are one such example using images, but suppose we limit ourselves to the medium of text. 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.

Similarly, you could specialize a Turing Test to determine *which* human generated the text; and on the opposite side, you can generate text to sound like someone specific, such as Shakespeare or Guy Blelloch. The same program can imitate a stylistic voice of your choosing by being predicated on the *input corpus*, or large body of text from which it generates its vocabulary.

One way to generate text from a corpus is based on *k-grams*: chunks of text k words long. After selecting k words, you make a random choice for the $k+1$st word weighted by the frequency of that $k+1$-gram in the input corpus.

Your task is to write a library for parsing text and storing k-gram information in a table. Then, you will use this library to write a short application for generating realistic, Turing Test-passing text, and you will test them on real corpora.

1.1 Input data

The handout will include three files for input: `shakespeare.txt`, `kennedy.txt` and `mobydick.txt`.

1.2 Submission

This assignment is distributed in a number of files in our `git` repository. Instructions on how to access that repository can be found at `http://www.cs.cmu.edu/~15210/resources/git.pdf`. This is how assignments will be distributed in this course.

This assignment requires that you submit both code and written answers.

Submit your solutions by placing you solution files in your handin directory, located at

```
/afs/andrew.cmu.edu/course/15/210/handin/<yourandrewid>/assn3/
```

Name the files exactly as specified below. You can run the check script located at

```
/afs/andrew.cmu.edu/course/15/210/bin/check/03/check.pl
```

to make sure that you've handed in appropriate files. Do not submit any other files.

Your written answers must be in a file called `hw03.pdf` and must be typeset. You do not have to use LaTeX, but if you do, we have provided the file `defs.tex` with some macros you may find helpful. This file should be included at the top of your `.tex` file with the command `\input{<path>/defs.tex}`.

For the programming part, the only files you're handing in are

```
parser.sml
parser-test.sml
kgram-stats.sml
kgram-test.sml
babble.sml
```

These files must contain all the code that you want to have graded for this assignment and must compile cleanly. If you have a function that happens to be named the same as one of the required functions but does not have the required type, it will not be graded.

1.3 Naming Modules

The questions below ask you to organized your solutions in a number of modules written from scratch. Your modules must be named exactly as stated in the handout. Correct code inside an incorrectly named structure, or a structure that does not ascribe to the specified signatures, *will not receive any credit*.

You may not modify any of the signatures or other library code that we give you. We will test your code against the signatures and libraries that we hand out, so if you modify the signatures your code will not compile and you will not receive credit.

1.4 The SML/NJ Build System

This assignment includes a substantial amount of library code spread over several files. The compilation of this is orchestrated by CM through the file `sources.cm`. Instructions on how to use CM can be found at on the website at `http://www.cs.cmu.edu/~15210/resources/cm.pdf`.

2 Parallel Text Parsing

In this task, you will write a parallel string *tokenizer* using the familiar *sequence* operations (`tabulate`, `filter`, `map2`,..).

Task 2.1 (10%). Implement a functor

```
Parser(Seq : SEQUENCE) : PARSER
```

in the file `parser.sml`. The `PARSER` signature is:

```
signature PARSER =
sig
  structure Seq : SEQUENCE
  val tokens : (char -> bool) -> string -> string Seq.seq
end
```

`tokens f s` should have the following behavior: let those characters `c` for which `f c` returns `true` be called *delimiting characters*. The result should be a sequence of maximal non-empty substrings (tokens) from the input string containing no delimiting characters. By maximal we mean that they cannot be extended on either side without including a delimiter. When the sequence of tokens is concatenated it must form the equivalent of `s` with all delimiting characters removed. It should be identical in behavior to `ArraySequence.tokens`. For our application, we'll use the function `fn c => not (Char.isAlphaNum c)`, but your `tokens` function should work correctly on other choices of delimiters as well.

For full credit your implementation must use $O(n)$ work and $O(l + \log n)$ span, where n is the length of the input string and l is the length of the largest token returned. You can assume that `String.substring(s,i,l)` takes $O(l)$ work and span.

Remark: For parsing you will probably end up evaluating large amount of conditionals. We found in our test, that SML/NJ compiler has a bug when compiling "case x of" statements, leading to very slow running time and even a memory leak. If you experience same problem, you might need to change your "case" to if-statements.

Task 2.2 (5%). Test your implentation in `parser-test.sml`. It should be easy to test your implementation against a benchmark.

Task 2.3 (5%). Explain informally why your implementation has work $O(n)$ and span of $O(l + \log n)$.

3 K-Gram Statistics

The next task is to implement a data structure to support a KGRAM_STATS abstract data type for storing statistics for a corpus. The functions of the data type build the data structure from the corpus (represented as sequence of tokens) and then can answer queries about any given "k-gram" (k consecutive tokens). For a given K, the kgram-stats table needs to store information for all k-grams that appear in the corpus for $k = 0$ (the empty k-gram) up to K. All code from this section should go in the file `kgram-stats.sml`

> **Assumptions:** Throughout this problem, we'll assume that K is constant and that all words (tokens) have length at most a constant L_0.

Task 3.1 (10%). As a preliminary task, you will implement a function that creates a histogram. As an input it will take an ordering operator (such as String.compare) and a sequence of values. As an output it gives a list of tuples (key, n) for each unique key in the input sequence, where n is the number of occurrences of the key in the input.

```
signature HISTOGRAM =
sig
    structure Seq : SEQUENCE
    val histogram :  ('a * 'a -> order) -> 'a Seq.seq -> ('a * int) Seq.seq
end
```

You will probably find this function useful in the main task. The cost of histogram, assuming the ordering operator has constant cost, should be $O(n \log n)$ work and $O(\log^2 n)$ span.

Task 3.2 (30%). Create a structure `KgramStatsTable : KGRAM_STATS`. Use appropriate internal data representation to achieve the cost specifications.

For full credit you need to match the following cost bounds: For an input corpus with n tokens the costs of `make_stats` must be bounded by $O(n \log n)$ work and $O(\log^2 n)$ span, and the cost of both `lookup_freq` and `lookup_extensions` bounded by $O(\log n)$ work and span.

The `KGRAM_STATS` signature follows:

```
(* This is the signature for the kgram-statistics Abstract Data Type *)
signature KGRAM_STATS =
sig
  structure Seq : SEQUENCE
  type kgramstats (* self type *)
  type kgram = token Seq.seq
  type token = string

  (* make_stats tokens K
     Construct the underlying data structure given the sequence tokens
     representing the corpus and a maximum k-gram size K. *)
  val make_stats: token Seq.seq -> int -> kgramstats

  (* lookup_freq corpus prefix token
     For the corpus and a "prefix" of length at most K returns a pair
     consisting of:
      1) the number of times "token" appeared immediately after "prefix"
         for |prefix|=0 this is the total # of times token appears
      2) the total number of tokens that appear after "prefix"
         for |prefix|=0 this is the total size of the corpus *)
  val lookup_freq : kgramstats -> kgram -> token -> (int * int)

  (* lookup_extensions corpus prefix
     For the corpus and a "prefix" of length at most K returns a
     sequence of pairs each consisting of
      1) a token that appears at least once immediately after "prefix", and
      2) a count of how many times that token appears
     Every token that appears after "prefix" must appear in the seq *)
  val lookup_extensions : kgramstats -> kgram -> ((token * int) Seq.seq)
end
```

Hint: In the library directory the file `defaults.sml` contains a structure definition for Tables with string sequence as a key `StringSeqTable`. You might find it helpful. You can access it as `Default.StringSeqTable`.

Task 3.3 (5%). Write a test structure in `kgram-test.sml` that tests your implementation of `KGRAM_STATS` using $K = 3$ and a hard-coded test corpus (of at least 50 tokens). Test that it returns correct values for prefixes of length $0, 1, 2$ and 3 using prefixes from beginning, middle and end of the input corpus. Test that it works properly with prefixes that do NOT occur in the test corpus.

4 Babble

Using the K-gram statistics 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. You need to implement the following signature consisting of the babble problem applied to one sentence and a document, and a helper function `choose_random`.

```
signature BABBLE = sig
  structure KS : KGRAM_STATS

  (* Function for selecting a value from a histogram.
     [choose_random f hist] returns a value from the histogram hist
     corresponding to the cumulative distribution at f, where
     f is a (random) number from 0 to 1. *)
   val choose_random : real ->  (KS.token * int) KS.Seq.seq -> KS.token

  (* generate_sentence corpus n seed
     Generates a sentence consisting of n words (tokens) of babble.
     The words are output as a string with spaces between each word
     and ending with a period.
     Each word should be selected on a random basis weighted by its
     likelyhood to follow the previous k tokens and using the
     random seed.   See the description in the text. *)
  val generate_sentence : KS.kgramstats -> int -> int -> string

  (* generate_document corpus n seed
     Generates n sentences (in parallel) with random lengths between
     5 and 10.   Each should be generated with a different seed
     based on the input "seed" (e.g. seed+1, seed+2, ...).
     Sentences should be appended together into a string *)
  val generate_document : KS.kgramstats -> int -> int -> string
end
```

Task 4.1 (10%). Implement `choose_random`. Note that instead of creating a random number itself, the function is passed a random real value from 0 to 1. If this value is not in this range, you can raise an exception.

As an example, consider a histogram: `h = < ("a", 3), ("b", 5), ("c", 2) >`. The cumulative distribution function table is:

a	0.3
b	0.8
c	1.0

Then `choose_random 0.2 h` would return "a", and `choose_random 0.75` should return "b" and `choose_random 0.95` should return "c". That is, it should return the key with least value above f.

Your solution should have linear work in the length of the histogram.

Task 4.2 (3%). Describe how would you implement `choose_random` in linear work and $O(\log n)$ span. You don't need to code it.

Task 4.3 (2%). Describe how you could implement `choose_random` in $O(\log n)$ work and span if you could preprocess the histogram.

Task 4.4 (20%). Implement `generate_sentence` and `generate_document`. We supply a structure `Random210` in the library to generate random numbers. You should use it to generate a sequence of n random numbers from a seed `rseed` as follows:

```
val seed = (Random210.fromInt rseed)
val randomVals = Seq.tabulate (fn i => Random210.randomReal seed i) n
```

Here is how to generate each word in a sentence sequentially with the appropriate probability. Assume you have already generated l words. Let `prefix` be the last (most recent) $\min(l,k)$ words. If this has appeared in the corpus then you should select one of the words that appear after it using `choose_random` (i.e. the selection should be weighted by the number of times each word appears). If `prefix` has not occurred in the input corpus, then try with a prefix that is one shorter (i.e. the previous $\min(l,k)-1$ words). Repeat until you find the kgram in the corpus.

As an example, assume that you have generated words *"dog barked at the cat in the chimney"*, and that you have computed K-gram statistics with $K = 3$. Now to generate the next word, you should first check for any words that followed *"in the chimney"*. If there are none, you should check for any words that followed *"the chimney"*. If that yields no results, then use *"chimney"*. And as a last resort, use an empty-prefix, i.e choose the next word from all words in the corpus (weighted by their frequency).

4.1 Help for testing your code

In the file `babble-test.sml` you will find code for reading an input file (corpus) and for generating an output file of 50 sentences of "babble" for each of three sample files: `kennedy.txt`, `shakespeare.txt`, and `mobydick.txt`.