Lab 4 - *BabbleLab*

Due: *Mon, Feb 17th, 2014 @ 11:59 pm*

Parallel and Sequential Data Structures and Algorithms 15-210 (Spring '14)

1 Introduction

Often when we say "I love you" what we really mean is: "You're pretty close to the defense of advertising agencies", but it's no big deal. - Mark V Shaney

This week, you will be implementing a version of the **Dissociated Press** algorithm. Taking content from an input corpus, or body of text, the algorithm generates a string of facetious garbage, structured as if it were "babbled" by a monkey on a typewriter. This is a bit of a change of pace from previous labs – hopefully you will find it to be less algorithmically challenging, and a bit more fun. BabbleLab will also help you become familiar with the `TABLE` signature.

2 Files

After downloading the assignment tarball from Autolab, extract the files by running:

```
tar -xvf babblelab-handout.tgz
```

from a terminal window. Some of the files worth looking at are listed below. You should only modify the files denoted by *, as these will be the only ones handed in by the submission script.

1. `Makefile`
2. `data/`
3. * `MkSeqUtil.sml`
4. * `MkKGramData.sml`
5. * `MkBabble.sml`
6. `Tests.sml`
7. `corpus.txt`

Additionally, you should create a file called:

```
written.pdf
```

which contains the answers to the written parts of the assignment.

3 Submission

To submit your assignment to Autolab, open a terminal, `cd` to the `babblelab` folder, and run:

```
make
```

Alternatively, run `make package`, open the Autolab webpage and submit the `handin.tgz` file via the "*Handin your work*" link.

4 Sequence Utilities

This lab requires a few functions that extend the `SEQUENCE` library, which will be part of the functor `MkSeqUtil` in `MkSeqUtil.sml`. You'll find these functions useful when programming your monkeys.

4.1 Histograms

One concept you need to familiarize yourself with is that of a *histogram*. 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.

Task 4.1 (5%). In `MkSeqUtil.sml`, implement the function:

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

where `(histogram cmp S)` evaluates to a sequence of $(key, count)$ pairs. Each pair describes a unique key from `S` and the number of times it appears in `S`. The resulting histogram should be ordered with respect to `cmp`. For full credit, `(histogram cmp S)` should have $O(|S| \log |S|)$ work and $O(\log^2 |S|)$ span. Our solution almost fits in a single line ☺.

Task 4.2 (10%). In `MkSeqUtil.sml`, implement the function

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

If $0 \leq r \leq 1$, `(choose H r)` should evaluate to the value at r from the cumulative distribution that corresponds to the histogram H. You may assume r is in the range $[0, 1]$ and H is non-empty. For full credit, `(choose H r)` should have $O(|H|)$ work and $O(\log |H|)$ span. Our solution is about 10 lines long.

As an example, consider the histogram $H = \langle(\text{“}a\text{”}, 3), (\text{“}b\text{”}, 5), (\text{“}c\text{”}, 2)\rangle$. 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 **the least value above or equal to** r.

4.2 Testing

As in previous labs, you may use the `Tester` structure to test your implementations. `Tester` will pull test cases from the lists inside of `Tests.sml`. We have already provided a few test cases for you, but you can (and should!) add more tests to this file. To test your implementation:

```
$ smlnj
Standard ML of New Jersey v110.xx
- CM.make "sources.cm";
- Tester.testHist ();
- Tester.testChoose ();
```

Additionally, you may access your functions directly from within the `StuCharBabble : BABBLE_PACKAGE` structure, which contains a sub-structure `Util : SEQ_UTIL`. Here's an example:

```
- CM.make "sources.cm";
- val % = Tester.Seq.fromList;
- Tester.StuCharBabble.Util.choose (% [("hi",1),("there",2)]) 0.5;
- Tester.StuCharBabble.Util.histogram Int.compare (% [1,2,3,4]);
```

5 K-Gram Data

5.1 Implementation

The `KGRAM_DATA` signature defines an ADT for storing information about k-grams. We define a k-gram as any consecutive subsequence of tokens of length k. To gather information about the k-grams in a particular corpus, we must first tokenize it. For example, below is a corpus followed by its tokenization, assuming whitespace is used as a delimiter:

"I spent an interesting evening recently with a grain of salt."
$\langle$I, spent, an, interesting, evening, recently, with, a, grain, of, salt.$\rangle$

For $k = 6$, all k-grams of this tokenized corpus would be:

$\langle$I, spent, an, interesting, evening, recently$\rangle$
$\langle$spent, an, interesting, evening, recently, with$\rangle$
$\langle$an, interesting, evening, recently, with, a$\rangle$
$\langle$interesting, evening, recently, with, a, grain$\rangle$
$\langle$evening, recently, with, a, grain, of$\rangle$
$\langle$recently, with, a, grain, of, salt.$\rangle$

You will define the `kgramdata` type to store information about every k-gram which appeared in the input corpus. Specifically, if given a valid $(k-1)$-gram, you should be able to retrieve the information necessary to construct a valid k-gram. Make sure you read and understand all functions which are dependent on the `kgramdata` type before attempting to implement it.

Also, note that the `MkKGramData` functor is parameterized on a structure `Tok : TOKEN`, which implements some representation of tokens and tokenization. As you will see later, this allows us to implement multiple styles of Babbling for some crazy results.

Task 5.1 (4%). Define the `kgramdata` type, and write a short comment describing the representation that you chose.
Hint: You should make use of the functor argument

```
structure Table : TABLE where type Key.t = Tok.token Tok.Seq.seq
```

Specifically, an `'a table` defines a table with keys of type `token seq` and values of type `'a`.

Task 5.2 (24%). In `MkKGramData.sml`, implement the function

```
val makeData : string -> int -> kgramdata
```

where `(makeData corpus k)` evaluates to a value of type `kgramdata` containing information for all k-grams in the input corpus. You may assume $k > 0$.

For full credit, `makeData` should have $O(n \log n)$ work and $O(\log^2 n)$ span, where n is the number of tokens in the input corpus. Our solution is about 15 lines long.

You should split the input corpus into tokens using the function `Tok.tokenize`. You might also find `Seq.collate` useful. For asymptotic analysis, you may assume that k is a small constant.

Task 5.3 (4%). In `MkKGramData.sml` implement the function

```
val lookupHist : kgramdata -> kgram -> token hist option
```

where `(lookupHist data gram)` evaluates to `SOME hist`, where `hist` is a histogram of tokens which followed `gram` in the input corpus. If `gram` was the last $(k-1)$-gram in corpus (and therefore no tokens followed it), then `(lookupHist data gram)` should evaluate to `NONE`. For full credit, `lookupHist` should have $O(\log n)$ work and span, where $n = |data|$.

Note that `data` will only contain information for k-grams of a specific length. Therefore, if $|gram| \neq k-1$, where `data` was created with the call `(makeData corpus k)`, then `(lookupHist data gram)` should evaluate to `NONE`.

For example, suppose we perform the following, where tokenization uses both whitespace and punctuation as delimiters:

```
- val corpus = "My name is Harold; his name is John; his name is Harold,
also; but her name was Jennifer.";
- val data = makeData corpus 3;
- val h1 = lookupHist data (Tok.tokenize "name is")
- val h2 = lookupHist data (Tok.tokenize "his name")
- val h3 = lookupHist data (Tok.tokenize "My")
- val h4 = lookupHist data (Tok.tokenize "Excalibur")
```

Then we will have

$$h1 = SOME\,\langle(\text{Harold}, 2), (\text{John}, 1)\rangle$$
$$h2 = SOME\,\langle(\text{is}, 2)\rangle$$
$$h3 = NONE$$
$$h4 = NONE$$

Task 5.4 (4%). In `MkKGramData.sml`, implement the function

```
val getK : kgramdata -> int
```

where `(getK data)` evaluates to the integer argument `k` of the call to `makeData` that was used to create `data`. For full credit, `getK` should have $O(1)$ work and span.

Task 5.5 (4%). In `MkKGramData.sml`, implement the function

```
val chooseStarter : kgramdata -> real -> kgram
```

where `(chooseStarter data r)` should evaluate to some gram from the input corpus of length $k-1$. You may assume $0 \leq r \leq 1$. For full credit, `chooseStarter` should have $O(n)$ work and $O(\log n)$ span, where $n = |data|$.

You may choose to implement this function however you like. However, we recommend augmenting your `kgramdata` type with a histogram of all $(k-1)$-grams from the input corpus, then using `Util.choose` to pick one.

This function will allow you to pick a few starting tokens when generating sentences in the next section. Ultimately, then, your choice of implementation will only affect the quality of the *beginning* of the sentences which you generate.

5.2 Testing

Again, you should use `Tester` to test your code. This time, `Tester` will test your implementation of `makeData` and `lookupHist` in one go. This is because you have defined your own `kgramdata` type, so we cannot test either of these functions independently. Remember to add your own tests in `Tests.sml`! In the SMLNJ REPL, run:

```
- Tester.testKGrams ();
```

To access your functions directly, see the following:

```
- val corpus = "red leather yellow leather mellow leather yellow leather";
- val data1 = Tester.StuStringBabble.Data.makeData corpus 2;
- val hist1 = Tester.StuStringBabble.Data.lookupHist data
(Tester.StringTok.tokenize "leather");

- val data2 = Tester.StuCharBabble.Data.makeData corpus 5;
- val hist2 = Tester.StuCharBabble.Data.lookupHist data
(Tester.CharTok.tokenize "ello");
```

6 Babble

Using the `KGRAM_DATA` abstract data type, it is easy to write an algorithm that generates text which is statistically similar to an input corpus. The general idea is to iteratively look up $(k-1)$-grams which have already been generated, each time randomly choosing the next token.

For example, suppose $k = 3$, and you call `Data.chooseStarter` which returns $\langle$Hello, my$\rangle$. You could then call `lookupHist` on this $(k-1)$-gram to get a histogram of possible next tokens, from which you should choose one at random using `Util.choose`. Let's say the next token you chose was "name" – so far, you have assembled the sequence $\langle$Hello, my, name$\rangle$. You could then use `lookupHist` on the gram $\langle$my, name$\rangle$ in order to be able to choose the next token, and so on.

Text generated this way has a special property, which we will call the "k-gram property": **every contiguous subsequence of k tokens appeared somewhere in the input corpus.** Because of this, your generated text will be statistically similar to the input text, and might even sound like it was written by a somewhat grammatically-challenged human.

Before we get to babbling, though, let's talk about randomness.

6.1 Functional, Deterministic Randomness

Perhaps it sounds counter-intuitive, but deterministic randomness is actually very useful – primarily because it allows us to repeat simulations. (It also fits in nicely with the functional paradigm!)

The most important aspect to remember about deterministic randomness is that, if given the same seed, a random number generator will always spit out the same number. Because of this, if you want to generate 10 random numbers, **you first need to generate 10 seeds.**

We've made it a little easier for you with the `RANDOM210` signature. Some specific functions of interest would be

```
fromInt : int -> rand
randomInt : rand -> ((int * int) option) -> (int * rand)
randomReal : rand -> ((real * real) option) -> (real * rand)
randomIntSeq : rand -> ((int * int) option) -> int -> (int seq * rand)
randomRealSeq : rand -> ((real * real) option) -> int -> (real seq * rand)
```

In the above, values of type `rand` are seeds. Each function returns a tuple containing the result, as well as a new seed. Remember: **if you use the same seed, you will get the same result.**

In terms of functionality, `(randomRealSeq seed NONE n)` generates n random `real` values $\in_{\mathbb{R}} [0,1]$. Similarly, `(randomIntSeq seed (SOME (i,j)) n)` generates n random `int`s in the range $[i,j)$.

You could generate n seeds by mapping `fromInt` across the sequence returned by `(randomIntSeq seed NONE n)`.

6.2 Implementation

Task 6.1 (10%). In MkBabble.sml, implement the function

```
val makeSentence : kgramdata -> int -> Rand.rand -> sentence
```

where `(makeSentence data n seed)` generates a token sequence of length $\leq n$, depending on the given `seed`. You may only return a sequence of length less than n if the generated sentence ends with the final token of the input corpus. You may assume that $n >$ `(getK data) - 1`.

Your sentence should have the *k-gram property* for the corpus that was used to create `data`. You should use `chooseStarter` (which you defined in `MkKGramData.sml`) to pick the starting $(k-1)$-gram.

For full credit, `makeSentence` must have $O(|data| + n \log |data|)$ work and $O(n \log |data|)$ span, ignoring the cost of token comparisons.

Task 6.2 (10%). In MkBabble.sml, implement the function

```
val makeParagraph : kgramdata -> int -> (int * int) -> Rand.rand
                    -> sentence seq
```

where `(makeParagraph data n (low, high) seed)` generates n sentences, dependent on `seed`. Each sentence should be generated with a call to `(makeSentence data` $\texttt{len}_i$ $\texttt{seed}_i$`)`, where for each i, j, `low` $\leq \texttt{len}_i <$ `high` and $\texttt{seed}_i \neq \texttt{seed}_j$. (This last piece is especially important: if you used the same seed every time, you would get n identical sentences!)

For full credit, `makeParagraph` must have $O(n \cdot W_{\texttt{makeSentence}}(x))$ work and $O(S_{\texttt{makeSentence}}(x))$ span, where $x =$ `high`.

6.3 Testing (Babbling)

Because your functions make use of randomness, we will not be comparing them against a reference solution. However, in `Tests.sml`, you can still write inputs for babbling. To run them, type following in the SMLNJ REPL:

```
- Tester.testCharBabble ();
- Tester.testStringBabble ();
```

You should notice dramatically different results. The first uses individual characters as tokens, whereas the second uses whole words as tokens.

Note that babbling requires you to have completed all sections of the lab (using our reference implementations is far too slow.)

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

7 Fun with Sets

You are so touched by the stories generated by your babbling monkeys that you have decided to publish them in a book. However, your simian friends smartly babbled in parallel, leaving you with the task of merging their sets of babbled documents. Before getting to work, you of course analyze the costs with the 15-210 library documentation in hand.

Task 7.1 (5%). Based on the cost specification for Sets, what is the asymptotic work and span for taking the union of two sets one of size n and the other of size $\sqrt{n}$? You can assume the comparison for sets takes constant work. Please simplify as much as possible.

Task 7.2 (5%). In a functional setting, taking $X = A \cup B$ results in a new value X, the union of A and B. However, we still have the original A and B (since everything is persistent).

It seems that since X is at least as large as $n = \max\{|A|, |B|\}$, it would take at least $\Omega(n)$ work to generate a new X, especially since we can't modify A or B. Does this break your analysis above? Explain in a couple sentences how this could be possible (i.e. why our lower bound is not the case).

Task 7.3 (15%). Satisfied, you write the following code to take the union of a sequence of sets:

$$UnionSets(S) = Seq.reduce\ Set.Union\ \ S$$

For example

$$UnionSets(\langle\{5,7\},\{3,1,5,2\},\{2,5,8\}\rangle)$$

would return the set $\{1,2,3,5,7,8\}$. Derive (tight) asymptotic upper bounds for the work and span for UnionSets in terms of $n = |S|$ and $m = \sum_{x \in S} |x|$. You can assume the comparison used for the sets takes constant work. Please keep your analysis to under a page. We do not need a formal proof. An explanation based on the tree-method is sufficient.