RANDOM TEXT GENERATION (413/413 points)

The goal of this project is to synthesize natural language sentences using information extracted from an existing text corpus.

word	→	next	freq
"START"	→	"I"	100%
"I"	→	"am"	100%
"am"	→	"a"	100%
"a"	→	"man"	25%
	|	"good"	75%
"man"	→	"and"	50%
	|	"STOP"	50%
"and"	→	"my"	50%
	|	"a"	50%
"my"	→	"dog"	100%
"dog"	→	"is"	33%
	|	"and"	33%
	|	"makes"	34%
"good"	→	"dog"	66%
	|	"man"	34%
"is"	→	"a"	100%
"makes"	→	"a"	100%

For this, given a text corpus as input, we will first compute the frequency of all sequences of two words in the original text; then we will use this information to produce new sentences by randomly collating these sequences with the same frequencies.

This method is known under the term of *Markov chain*. From the input text, we compute a transition table that associates to each word the list of words that may appear after it, with their relative frequencies.

For instance, if we examine *"I am a man and my dog is a good dog and a good dog makes a good man"*, delimiting it with `"START"` and `"STOP"` to identify the beginning and end of the sentence, we end up with the transition table on the right.

This table can then be used to generate new text that ressembles the input in the following way: starting from the `"START"` word, choose one of the words that may appear after it, with the probability found in the table, add it to the output, then iterate the process until the `"STOP"` word is found. Below are some example sentences produced using this table.

```
START I am a good  man STOP ; START I am a good dog  is a good dog and
my dog and my  dog is a man and my  dog and a man STOP ;  START I am a
good dog is a man and my dog makes a good man STOP ; START I am a good
dog makes a good dog is a good dog  and a good dog makes a good dog is
a man STOP ; START I am a good dog and a man and a good dog and a good
man and a good dog is a good dog  is a good man and a man STOP ; START
I am a good dog and a good dog and my dog is a man STOP ; START I am a
man STOP ;  START I am a good dog  is a good dog is a  good dog and my
dog is a man STOP ; START I am a good man STOP ; START I am a good dog
makes a good dog and a good dog is a good dog is a good man and my dog
is a good dog and  my dog and a good man and a good  dog is a good man
STOP ; START I am a man and my dog and my dog is a good dog and a good
dog makes a man STOP
```

This project is decomposed in three parts.

A. First, we will build a quick prototype, that goes from an input sentence to a randomly generated sentences via a distribution table as the ones above.

B. Then we will use better data structures to enhance the performance so that we can use larger texts, such as small books, as input.

C. After that, we will enhance the quality of input and output, by analysing in a smarter way the input text corpus, and by considering sequences of more than two words.

Note: this project may take more time to be graded, because it is longer than simple exercises, and because it is tested on large inputs. We suggest that you use the typecheck button and the toplevel extensively, so that you are reasonnably sure of your code before submitting it to the tests. Also, we added a function `grade_only : int list -> unit`, that you may call in your code to select the exercises to grade. All other exercises won't be graded at all, and considered failed. For instance, if you write `grade_only [ 3 ] ;;` at the beginning of your file, only exercise 3 will be tested.

PART A: A FIRST DRAFT

Our first goal will be to build such a table and generate sentences from it, quick and dirty style, using lists and their predefined operators. Consider using as much as possible the `List` module (`List.assoc` , `List.length` , `List.nth` , etc.) and don't think about efficiency.

In this exercise, we will use associative lists as the data structure that links each word to its possible suffixes. Associative lists are often used in prototypes or non critical programs because they are very easy to use and debug. Their major downfall is the complexity of searching for an element.
The type of an associative list that maps `string` keys to `'a` values is simply `(string * 'a) list` . The value associated with a key `"x"` is simply the right component of the first pair in the list whose left component is `"x"` . This lookup is already defined in the standard library as `List.assoc` . Hence, setting the value of `"x"` to `3` , for instance, is just adding `("x",3)` in front of the list. To remove an element, you can just use `List.filter` with the right predicate.

The type of lookup tables for this exercise is

```
type ltable = (string * string list) list
```

1. Write a function `words : string -> string list` that takes a sentence and returns the list of its words. As a first approximation, will work on single sentences in simple english, so you can consider sequences of roman letters and digits as words, and everything else as separators. If you want to build words bit by bit, you can experiment with the Buffer module. Beware, this preliminary function may not be as easy as it seems.
2. Write `build_ltable : string list -> ltable` that builds an associative list mapping each word present in the input text to all its possible successors (including duplicates). The table should also contain `"START"` that points to the first word and `"STOP"` that is pointed by the last word.
   For instance, a correct (and minimal) table for `"x y z y x y"` looks like:

   ```
   [ ("z", [ "y" ]);
     ("x", [ "y" ; "y" ]);
     ("START", [ "x" ]);
     ("y", [ "x" ; "z" ; "STOP" ]) ]
   ```

3. Write the random selection function `next_in_ltable : (string * string list) list -> string - > string` which takes a table, a given word and returns a valid successor of this word. Your function should respect the probability distribution (which should be trivially ensured by the presence of the duplicates in the successor lists).
4. Write the random generation function `walk_ltable : (string * string list) list -> string list` which takes a table and returns a sequence of words that form a valid random sentence (without the `"START"` and `"STOP"`).

You can use `display_quote: string list -> unit` to display the generated texts.

PART B: PERFORMANCE IMPROVEMENTS

Now, we want to use more efficient data structures, so that we can take larger inputs and build bigger transition tables.

In this exercise, we will use hash tables, predefined in OCaml in the Hashtbl module. Used correctly, hash table provide both fast insertion and extraction. Have a look at the documentation of the module. In particular, don't miss the difference between `Hashtbl.add` and `Hashtbl.replace` (you'll probably want to use the latter most of the time).

The types for this exercise are:

```
type distribution =
  { total : int ;
    amounts : (string * int) list }
type htable = (string, distribution) Hashtbl.t
```

5. In the simple version, we stored for each word the complete list of suffixes, including duplicates. This is a valid data structure to use when building the table since adding a new suffix in front of the list is fast. But when generating, it means computing the length of this list each time, and accessing its random `nth` element, which is slow if the list is long.
   Write `compute_distribution : string list -> distribution` that takes a list of strings and returns a pair containing its length and an association between each string present in the original list and its number of occurrences.
   For instance, `compute_distribution ["a";"b";"c";"b";"c";"a";"b";"c";"c";"c"]` should give `{ total = 10 ; amounts = [("c", 5); ("b", 3); ("a", 2)] }`.
   Hint: a first step that simplifies the problem is to sort the list.
6. Write a new version of `build_htable : string list -> htable` that creates a hash table instead of an associative list, so that both table building and sentence generation will be faster.
   Like the associative list, the table is indexed by the words, each word being associated to its successors. But instead of just storing the list of successors, it will use the format of the previous question.
   Hint: You can first define an intermediate table of type `(string, string list) Hashtbl.t` that stores the lists of successors with duplicates. Then you traverse this intermediate table with `Hashtbl.iter`, and for each word, you add the result of `compute_distribution` in the final table.
7. Define `next_in_htable : htable -> string -> string` that does the same thing as `next_in_ltable` for the new table format.
8. Finally, define `walk_htable : htable -> string list`

PART C: QUALITY IMPROVEMENTS

9. If we want to generate sentences from larger corpuses, such as the ones of the `ebooks_corpus` given in the prelude, we cannot just ignore the punctuation. We also want to generate text using not only the beginning of the original text, but the start of any sentence in the text.
   Define `sentences : string -> string list list` that splits a string into a list of sentences such as:
   - uninterrupted sequences of roman letters, numbers, and non ASCII characters (in the range `'\128'..'\255'`) are words;
   - single punctuation characters `';'`, `','`, `':'`, `'-'`, `'"'`, `'\''`, `'?'`, `'!'` and `'.'` are words;
   - punctuation characters `'?'`, `'!'` and `'.'` terminate sentences;
   - everything else is a separator;
   - and your function should not return any empty sentence.

Now, we will drastically improve the results by matching sequences of more than two words. We will thus update the format of our tables again, and use the following `ptable` type (which looks a lot like the previous one).

```
type ptable =
  { prefix_length : int ;
    table : (string list, distribution) Hashtbl.t }
```

So let's say we want to identify sequences of N words in the text. The `prefix_length` field contains $N-1$. The `table` field associates each list of $N-1$ words from the text with the distribution of its possible successors.

The table on the right gives the lookup table for the example given at the beginning of the project: *"I am a man and my dog is a good dog and a good dog makes a good man"*, and a size of 2. You can see that the branching points are fewer and make a bit more sense.

prefix	→	next	freq
["START"; "START"]	→	"I"	100%
["START"; "I"]	→	"am"	100%
["I"; "am"]	→	"a"	100%

As you can see, we will use `"STOP"` as an end marker as before. But instead of a single `"START"` we will use as a start marker a prefix of the same size as the others, filled with `"START"`.

```
     ["am"; "a"] → "man"    100%
   ["man"; "and"] → "my"    100%
      ["is"; "a"] → "good"  100%
    ["and"; "my"] → "dog"   100%
    ["my"; "dog"] → "is"    100%
  ["makes"; "a"] → "good"   100%
   ["a"; "good"] → "man"    33%
                 | "dog"    66%
   ["dog"; "is"] → "a"      100%
    ["and"; "a"] → "good"   100%
 ["good"; "dog"] → "makes"  50%
                 | "and"    50%
  ["dog"; "and"] → "a"      100%
    ["a"; "man"] → "and"    100%
 ["good"; "man"] → "STOP"   100%
["dog"; "makes"] → "a"      100%
```

10. First, define `start: int -> string list` that makes the start prefix for a given size (`start 0 = []`, `start 1 = [ "START" ]`, `start 2 = [ "START" ; "START" ]`, etc.).
11. Define `shift: string list -> string -> string list`. It removes the front element of the list and puts the new element at the end.
    (`shift [ "A" ; "B" ; "C" ] "D" = [ "B" ; "C" ; "D" ]`,
    `shift [ "B" ; "C" ; "D" ] "E" = [ "C" ; "D" ; "E" ]`, etc.).
12. Define `build_ptable : string list -> int -> ptable` that builds a table for a given prefix length, using the two previous functions.
13. Define `walk_ptable : ptable -> string list` that generates a sentence from a given `ptable`. Unless you put specific annotations, `next_in_htable` should be polymorphic enough to work on the field `table` of a `ptable`, so you don't have to rewrite one. If you want, since we now have proper sentence splitting, you can generate multi-sentence texts, by choosing randomly to continue from the start after encountering a `"STOP"`.

Finally, the most funny texts are generated when mixing various kinds of inputs together (pirate stories, history books, recipes, political news, etc.).

14. Define `merge_ptables: ptable list -> ptable` that combines several tables together (you may fail with an exception if the prefix sizes are inconsistent).

Now you can try and generate some texts using larger inputs, such as short novels! The prelude provides a few samples, otherwise Project Gutemberg is a good source. You can use `display_quote: string list -> unit` to display the generated texts.

```
let sauce_ptable =
   merge_ptables
     (List.map
        (fun s -> build_ptable s 2)
        (sentences some_cookbook_sauce_chapter)) ;;
display_quote (walk_ptable sauce_ptable) ;;
```

THE GIVEN PRELUDE

```
type ltable = (string * string list) list
type distribution =
  { total : int ;
    amounts : (string * int) list }
type htable = (string, distribution) Hashtbl.t
type ptable =
  { prefix_length : int ;
    table : (string list, distribution) Hashtbl.t }

let simple_0 =
  "I am a man and my dog is a good dog and a good dog makes a good man"

let simple_1 =
  "a good dad is proud of his son and a good son is proud of his dad"

let simple_2 =
  "a good woman is proud of her daughter and a good daughter is proud of her mom"

let simple_3 =
  "there is a beer in a fridge in a kitchen in a house in a land where \
   there is a man who has a house where there is no beer in the kitchen"

let multi_1 =
  "A good dad is proud of his son. \
   A good son is proud of his dad."

let multi_2 =
  "A good woman is proud of her daughter. \
   A good daughter is proud of her mom."

let multi_3 =
  "In a land of myths, and a time of magic, \
   the destiny of a great kingdom rests \
   on the shoulders of a young man."

let grimms_travelling_musicians =
  "An honest farmer had once an ass that had been a faithful servant ..."

let grimms_cat_and_mouse_in_partnership =
  "A certain cat had made the acquaintance of a mouse, and ..."

let the_war_of_the_worlds_chapter_one =
  "No one would have believed in the last years ..."

let some_cookbook_sauce_chapter =
  "Wine Chaudeau: Into a lined saucepan put ½ bottle Rhine ..."

let history_of_ocaml =
  "“Caml” was originally an acronym for Categorical ..."
```

YOUR OCAML ENVIRONMENT

```
(* -- Part A ------------------------------------------------------------ *)
grade_only [1; 2; 3; 4; 5 ; 6 ; 7; 8 ; 9 ; 10 ; 11 ; 12 ; 13 ; 14] ;;

let rec words str =
  let sep = ' ' in
  try
    let i = String.index str sep in
    String.sub str 0 i ::
    words (String.sub str (i+1) (String.length str - i - 1))
  with Not_found ->
    [str]

let update_table x y table =
  try
    let pair = List.assoc x table in
    let lst = List.remove_assoc x table in
    (x, y::pair)::lst
  with Not_found ->
    (x, [y])::table;;

let build_ltable words =
  let rec aux_build table l =
    match l with
    | [] -> table
    | x::[] -> aux_build (update_table x "STOP" table) []
    | x::y::xs -> aux_build (update_table x y table) (List.tl l) in
  aux_build ["START", [List.hd words]] words;;
```

Evaluate >>

Switch >>

Typecheck

Reset Template

Full-screen [+]

Check & Save

Exercise incomplete (click for details) 413 pts

About...

Help

Contact

Terms of use

Usage policy

Privacy policy

Terms and conditions