Autocorrect Suggestion Using Levenshtein Automata (ANNOTATED)

We are using the Levenshtein distance algorithm in conjunction with Finite State Machines (Nondeterministic Finite-state Automata and Deterministic Finite-state Automata) to create a typo corrector and autocorrect suggestion tool, much like the concept of iOS's autocorrect feature. We will use ocaml. It will run in the command-line interface.

Features:

1. Basic Levenshtein Automata core (Successfully implemented!)
   - The Levenshtein distance between two strings s1 and s2 is the number of single-character insertions, deletions, or substitutions that are required to transform s1 into s2. The Levenshtein automaton takes as inputs string s and integer n, and then outputs the set all strings whose Levenshtein distance from *s i*s at most *n*. This has a useful application to spell checking as it can return a list of words that are "close" to the incorrectly typed word.
2. Keyboard ranking algorithm (Successfully implemented!)
   - Based on the proximity of keys on the keyboard, we can re-rank the results from the Levenshtein automata, in order of likely typos and accidental/mistake keystrokes. i.e. F-O-O-E, should suggest 'food' above 'foot', because the D-key is closer to E than T is. So we can start with a basic representation of the physical keyboard, by keeping a dictionary of distances between 2 given keys on the keyboard. This will allow us to give more reasonable typo suggestions based on the idea of an "off by one" error and a "double character" error are much more likely than other types of errors. (Works as intended!)
   - In the ideal implementation, we would like to use this keyboard ranking lookup within the building of our Levenshtein Automaton. Incorporate the "unlikeliness scores" by scoring certain edits based on which keys are substituted, inserted, or deleted. However, we understand that after the basic implementation of the Levenshtein Automaton, it could be more suitable to run this ranking process on the output of the Automaton. (Works as intended! We decided to run the ranking process on the output of the automaton.)
   - In addition to the keyboard distance score described above, we also decided to use an index that measures how frequent a word occurs in the English language, and we use a combination of the two scores (Keyboard distance and Frequency index) to suggest words. For example, an input word of 'snile' suggests 'smile' as the top choice over 'anile' even though they have the same score based on keyboard distances, because 'smile' is used much more frequently than 'anile'. In our test cases, we found that this greatly improved the quality of the suggestions.

Tech Spec:

Interface (ocaml):

Original specs that changed a lot!

```
module type NFA =
sig
	type transition = Epsilon | Any | Correct of char
	(*
	  we may also break transition down into further sub_types: insert | delete | swap | correct of char
	*)
```

We did have two Any types so as not to have to deal with any transition pointing to more than one state

```
	type state = int * int
```

The type of an nfa is now a tuple of the start state, dictionary representing transitions, and a set of final states

```
	type nfa = state * ((transition * state) list)

	val initialize:  nfa
	val add_state: nfa -> state -> transition -> state -> nfa
```

The to_dfa function is now in the Lev module that is a functor on both Dfa and Nfa. In this way it can easily convert between the two using the functionality of both modules

```
	val to_dfa: nfa -> dfa
end

module type DFA =
sig
```

Transitions are just a correct of a letter or an Other.

```
	type transition = Correct of char | Incorrect of (char list)
```

A state is defined to be a set of nfa states (int*int)

```
	type state = int * int
```

There was no need for this

```
	type powerset = state list
```

Similarly, this was defined as the nfa type

```
	type dfa = start * ((transition * powerset) list)

	val next_valid string: dfa -> string -> string
	val next_edge -> dfa ->
end
```

In our final implementation, NFA and DFA are both factored out into an ‘Automata’ module which contains logic that exists in both NFA and DFA, such as states and transitions. This common module is a functor that takes in a lot of modules that define a set over the states, dictionary over the states, dictionary over the transitions, along with the type of the transitions and states. Using a functor allowed for both Nfa and Dfa to use these function by passing in the appropriate modules. This is different from our original plan which was to implement DFA as a functor of NFA, and we think this was a good decision because it takes advantage of the overlap of logic between NFA and DFA while also allowing us to write logic that is different between NFA and DFA. In the NFA and DFA interfaces, we only needed to expose any methods that were utilized in the functions to_dfa and find_matches.

```
module Make (N : NFA) : DFA =
struct
	...
end

let levenshtein_automata (term: string) (e_distance: int) : NFA.nfa =
	(*produce the nfa for a word *)
;;

let find_matches (word: string) (e_distance: int) (lookup_fun : string -> string): unit =
	(* print the resulting matches*)
;;

(* following functions for comparing automata results against English dictionary *)

let extract_dictionary () : string list =
	(* extract dictionary from a file, e.g. the unix dictionary *)
;;

let lookup_dictionary (table: string list) (query: string) : string =
	(* give back
;;
```

As is evident, the structure of our code changed significantly. The code now uses a lot of abstraction in terms of types in order to factor out common code and make the code easy to change and expand. The code is split up into several different files so it is obvious and hopefully clear where to go to change certain aspects.

Timeline

By **April 21** we will have the standard Levenshtein Automata completed.
(This was successfully completed a day or two after we had planned.)
By **April 28** we will do either of two things:

1.) Treat the Levenshtein Automata algorithm we implemented as a black box and develop a method to rank the results from the Automaton using keyboard distances

2.) Incorporate our system of ranking the results directly into the automaton.

(This was also successfully completed a day or two after we had planned. We used Levenshtein as a black box and implemented a ranking algorithm.)

By **May 5** we will have the final version completed. Hopefully actually May 4 because May 5 is my birthday.

(We worked on improving user interaction, improving the scoring system by experimenting with the weights given to the scoring mechanisms, and enabling additional features such as autocorrect on an entire file.)