Obligatory exercise 3a (INF4820, Fall 2013)

This is part 1/2 of the third obligatory exercise in INF4820 2013. You can obtain up to 10 points for 3a, and you need a minimum of 12 points for 3a + 3b in total. If you have any questions, send an email to `inf4820-help@ifi.uio.no`.

Answers must be submitted via Devilry by the end of the day (23:59) on Sunday, November 3rd. Please provide your code and answers (in the form of Lisp comments) in a single '.lisp' file.

Summary of goals for this exercise

- Understand fully the computation of probabilities in Hidden Markov Models (HMMs);
- Design a data structure and associated functions to train an HMM from tagged training data;
- Implement the Viterbi decoding algorithm; train and test a PoS tagger.

Files you'll need

For this exercise we provide four files—`eisner.tt`, `wsj.tt`, `test.tt`, and `hmm-eval.lsp`—which contain two sets of training data (one small, one large), a smaller amount of test data, and some Lisp evaluation code. You can copy the files to your IFI home directory with the following shell command:

```
cp ~rdridan/inf4820/{eisner.tt,wsj.tt,test.tt,hmm-eval.lsp} ~/
```

Note that `wsj.tt` and `test.tt` are parts of the Penn Treebank, for which UiO holds a license for unlimited research use at the university. Please do not re-distribute these files.

All files ending in '`.tt`' contain part-of-speech tagged 'sentences' (or sequences of observations of ice cream consumption in the case of `eisner.tt`). Each sentence consists of a sequence of lines, where each line provides one $\langle o_i, q_i \rangle$ pair, i.e. an observation and its label. On each line, the observation and label are separated by one tab character (which can be referenced by name in Common Lisp as `#\tab`). Sentences are separated by one blank line.

1 Building a Hidden Markov Model

1.1 Theory: Hidden Markov Models

Assume the following part-of-speech tagged training 'corpus'

the	eagle	flies	like	the	wind		fruit	bats	like	flies	as	food
D	N	V	P	D	N		N	N	V	N	P	N

(a) Assuming a bigram HMM with start and end states, calculate

(i) all transition probabilities *from* the start state, i.e. $P(D|\langle S \rangle), P(N|\langle S \rangle)$ etc.

(ii) the emission probabilities of *flies*, given each tag.

In order to help your implementation, we strongly recommend that you work out the full transition and emission matrices for this 'corpus' and make sure you understand how to use these matrices to calculate $P(S|O)$ where $O = \langle$fruit flies like fruit$\rangle$ and $S = \langle$N N V N$\rangle$ or $S = \langle$N V P N$\rangle$. You do not need to submit these additional calculations.

(b) In a few sentences, discuss the concept of smoothing and why it is important.

1.2 Defining our model

(a) To represent HMMs in our code, we will again implement an abstract data type using `defstruct`. Define a structure *hmm* that has at least:

- `states`: a structure containing legitimate states (tags) for our model;
- `n`: the number of legitimate tags (i.e. the number of elements in `states`);
- `transitions`: a structure to hold transition probabilities; and
- `emissions`: a structure to hold emission probabilities.

Choosing suitable data structures for `transitions` and `emissions` should be your main focus here. The `transitions` component will be indexed by a pairs of states. As in Assignment 2, we will assign numeric ids to states for efficient storage and manipulation. Include a sentence motivating your choice of data structure(s) for the transition matrix.

The `emissions` component is indexed by a state and an observation (word). Why might we not want to assign numeric ids to words? Comment on how this affected your choice of data structure(s) for storing emission probabilities.

(b) To hide the internals of our HMM implementation, write access functions `transition-probability()` and `emission-probability()`. Both take an *hmm* object as their first argument, plus either two state ids (`transition-probability()`) or a state id and a word string (`emission-probability()`) as additional arguments, and return the value stored in `transitions` or `emissions` as appropriate. Also write a `state2id()` function that takes an *hmm* and a state label (i.e. string) as arguments, and returns an integer id. For states not yet known to the model, this function should allocate a new id, adding the state to the `states` structure and incrementing `n`.

1.3 Reading the training data

(a) Write a function `read-corpus()` that takes two arguments, a file name corresponding to a labelled training corpus, and an integer, specifying the size of the state set (not counting the initial and final states of the HMM). The function should read the corpus, one line at a time, break each line into its observation and label parts (both represented as Lisp strings) and count state bi-gram and labelled observation frequencies. At this stage, store the counts in the `transition` and `emission` structures. In the next step, we will turn these counts into probabilities.

Since we wish to model start and terminate transitions, your `read-corpus()` function needs to pay attention to the empty lines that indicate sentence breaks, and include counts of $(\langle s \rangle, t)$ and $(t, \langle /s \rangle)$ bi-grams to model these.

```
? (setf eisner (read-corpus "~/eisner.tt" 2))
→ #S(HMM :STATES ("<s>" "H" "C" "</s>") :N 4 :TRANSITIONS ??? :EMISSIONS ???)
```

It will be important to have the correct counts, hence test your implementation of `read-corpus()` and access functions on our sample file '`eisner.tt`' carefully. Here is what we would expect at this point:

```
? (transition-probability eisner (state2id eisner "<s>") (state2id eisner "H"))
→ 77
? (transition-probability eisner (state2id eisner "C") (state2id eisner "H"))
→ 26
? (transition-probability eisner (state2id eisner "C") (state2id eisner "</s>"))
→ 44
? (emission-probability eisner (state2id eisner "C") "3")
→ 20
```

(b) Once you have all the (correct) counts, what remains to be done is to alter the values in the `transitions` and `emissions` components to be relative frequencies, which are estimates of the true probabilities $P(s_i|s_{i-1})$ and $P(o_i|s_i)$. Write a function `train-hmm()` that, given an HMM recording frequency counts, destructively modifies the transitions and emissions components. For each transition and emission probability, `train-hmm()` should retrieve the corresponding counts, compute the probability, and change the relevant entry in the `transitions` and `emissions` data structures. Test to make sure your probabilities look correct:

```
? (setf eisner (train-hmm (read-corpus "~/eisner.tt" 2)))
? (transition-probability eisner (state2id eisner "C") (state2id eisner "H"))
→ 13/68
? (emission-probability eisner (state2id eisner "C") "3")
→ 5/34
```

2 Tagging with Viterbi

We cover this material in the lecture on October 23rd, so don't worry if it looks unfamiliar at first. Just focus on Section 1 until we go over this in class.

2.1 Theory: Viterbi

(a) Using the training corpus in section 1.1, and the input "flies like fruit", show the calculations used to fill in the first column of the trellis, and at least one state in the second column. (Again, you might find it helpful to do more calculations by hand, but you only need to show the first bit.)

(b) In a few sentences, summarize the key points of the Viterbi algorithm. What is the interpretation of each cell in the trellis? What is the complexity of the algorithm, i.e. the number of computations performed in relation to (i) the length of the observation sequence and (ii) the size of the label set? Discuss the naïve method of computing the most probable state sequence s_1^n, given an input string o_1^n very briefly; state how the Viterbi algorithm improves over this approach.

2.2 Implement Viterbi

(a) Implement the Viterbi algorithm we discussed in lectures as a function `viterbi()` that takes two arguments, an HMM and an input sequence, and returns the most probable sequence of state labels. You may find Chapters 5 and 6 in Jurafsky & Martin a useful additional reference.

Internally, the function should make use of two two-dimensional matrices: the so-called Viterbi *trellis* to record, for each state s and each time point (i.e. input position) t, the maximum probability of being in state s at time t; and the *backpointer* matrix that records the best path to each state. Ensure your implementation can handle unseen data. Since we are not implementing a smoothing algorithm, a simple option is to return a small default value for unseen data (e.g. `1/1000000`). Again using the eisner corpus, test to see that your function does the right thing:

```
? (viterbi eisner '("1" "1" "3" "3" "3" "3" "1" "1" "1" "1"))
→ ("H" "H" "H" "H" "H" "H" "C" "C" "C" "C")
```

(b) Now that it looks like your code works, it's time to test on a real data set. Create an *hmm* using the `wsj.tt` file (this is a large file – remember to compile your code!) and test your `viterbi()` function on a single sentence:

```
? (setf wsj (train-hmm (read-corpus "~/wsj.tt" 45)))
? (viterbi wsj '("No" "," "it" "was" "n't" "Black" "Monday" "."))
→ ("UH" "," "PRP" "VBD" "RB" "NNP" "NNP" ".")
```

The file `hmm-eval.lsp` contains a function `evaluate-hmm()` that takes an HMM and the name of a file containing tagged test data, and calls a `viterbi()` function on the observation sequences in the test data. Use it to test your implementation:

```
? (evaluate-hmm wsj "~/test.tt")
→ 0.95744324
```

What accuracy does it return? If your accuracy is below 95%, this may indicate remaining problems in your implementation. We've used standard probabilities here to make debugging easier, but a practical system would use log probabilities. In a couple of sentences explain why log probabilities are better, and what changes you would need to make to your implementation in order to use them. (You can actually implement these changes if you like.)

Happy coding!