Obligatory exercise 3b (INF4820, Fall 2013)

This is the second and final part of the third obligatory exercise in INF4820 2013. You can obtain up to 10 points for 3b, 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 Thursday, November 21st. Please provide your code as a '.lisp' file. Written answers can be included as comments, or in a separate file. Since this is the last exercise before the exam, there are more theory questions to give you practice with written answers. Try to answer them as you would in the exam.

Summary of goals for this exercise

- Understand how to read grammar rules from a treebank and estimate their conditional probabilities; implement a function to train a PCFG.
- Understand the structure of a generalised chart parser, and implement a version of the fundamental ruler.
- Understand how the Viterbi algorithm works over a packed parse forest.
- Understand various aspects of parser performance and how to evaluate them.

Files you'll need

For this exercise we provide four files—`chart.lsp`, `toy.mrg`, `wsj-train.mrg`, and `wsj-test.mrg`. The file `chart.lsp` is skeleton code, which you are expected to complete during this exercise. The `*.mrg` files are data for training and testing your parser and contain phrase structure trees in the form of Lisp s-expressions.

You can copy the files to your IFI home directory with the following shell command:

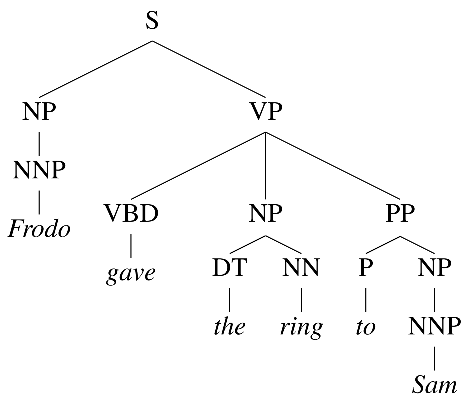
```
cp ~rdridan/inf4820/{chart.lsp,*.mrg} ~/
```

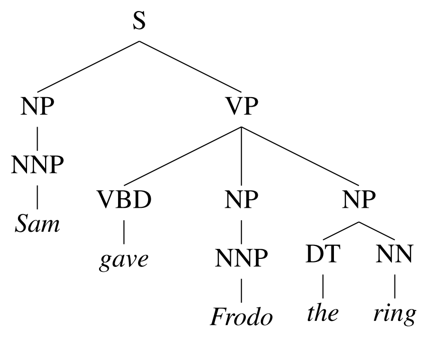
Note that `wsj-train.mrg` and `wsj-test.mrg` are parts of the Penn Treebank, for which UiO holds a license for unlimited research use at the university. Please do not re-distribute these files.

1 Estimating a PCFG from a treebank

1.1 Theory: Maximum likelihood estimation from a treebank

Given the following treebank, list the rules, their counts and the maximum likelihood estimation of their conditional probabilities. (Hint: you should find six lexical rules and six syntactic rules, not including START → S.)





1.2 Training a PCFG

In this section you will define a grammar structure and write a function to train the grammar from a treebank. You will also implement two accessor functions for the grammar that are required by our parser. Read the whole section before starting to code, since your grammar definition will be influenced by the accessor functions we need.

(a) The skeleton code contains structures for representing syntactic rules (*rules*) and lexical rules (*lexemes*), and a partially defined *grammar* structure. Using these structures (augmenting as necessary), implement the `read-grammar()` function, which takes a treebank file as an argument and returns a *grammar* object. This function should:

- Read in each tree (since the trees are represented as s-expressions, you might find Lisp's `read()` function useful here) and
  - recursively process each tree, extracting the rules it represents, and recording the counts in the probability slots.
    - For lexical rules, add a *lexeme* to the grammar (indexed by the word) if this rule has not been seen before. Otherwise increment the count of the appropriate lexeme.
    - For syntactic rules, add a *rule* to the grammar, if this rule has not been seen before **and** the rule is not a unary recursive rule (e.g., NP $\rightarrow$ NP). If the rule has already been added, just increment the count. (Note that in the next step you will write an accessor function that retrieves rules by the first element of the right hand side. This may influence how you store or index your rules.)
- Turn the *rule* and *lexeme* counts we just collected into estimates of conditional probabilities, using the standard relative frequency calculations. Store the probabilities as log probabilities.

(b) Our parser requires two accessor functions to retrieve the necessary rules from the grammar efficiently.

- `rules-starting-in()` takes a **category** and a **grammar** and returns a list of all rules in **grammar** which have **category** as the first element of the right hand side (i.e., the first non-terminal after the arrow).
- `get-lexemes()` takes a **word** and a **grammar**, and returns the list of lexemes relevant for **word** in **grammar**.

Implement these accessor functions. Consider if your *grammar* definition could be changed to make these functions more efficient.

To test your grammar implementation, first train a grammar on the `toy.mrg` file.

```
? (setf toy (read-grammar "~/toy.mrg"))
? (rules-starting-in 'NP toy)
→ (#S(RULE :LHS START :RHS (NP) :PROBABILITY -1.5404451)
 #S(RULE :LHS S :RHS (NP VP) :PROBABILITY 0.0))
? (get-lexemes "flies" toy)
→ (#S(LEXEME :CATEGORY VBZ :PROBABILITY -1.9459101)
 #S(LEXEME :CATEGORY NNS :PROBABILITY -2.0794415))
```

Once you are getting the right outputs, try the larger file, `wsj-train.mrg`. This will take longer, but should still finish in under a minute. (Remember to compile your code first.) If it takes significantly longer, re-work your code to make it more efficient. You should end up with just under 15000 rules and a little over 50000 lexemes. If we check the usage of "flies" in this grammar, you should see that The Wall Street Journal rarely writes about insects:

```
? (setf wsj (read-grammar "~/wsj-train.mrg"))
? (get-lexemes "flies" wsj)
→ (#S(LEXEME :CATEGORY VBZ :PROBABILITY -8.192017))
```

However, the Penn Treebank analyses have many more rules with a noun phrase as the first element on the right hand side:

```
? (length (rules-starting-in 'NP wsj))
→ 1783
```

2 Generalised chart parsing

We will cover this material in the lecture on November 6th.

The second part of the skeleton code implements a version of the generalised chart parser we discussed in lectures. While it may appear complicated, most of it maps fairly clearly to the algorithm we went through. There are the three structures—*chart*, *edge* and *agenda*—and the main `parse()` function. Look at the `parse()` function and identify our three stages: initialisation, main loop, termination. Also look at the `pack-edge()` function and try to understand how this helps us deal with ambiguity.

2.1 Theory: Understand the algorithm

When does a new edge get added to the agenda? For each of the situations where a new edge is pushed on to the agenda, briefly describe the conditions that must be true (if any) and the features of the new edge.

2.2 The fundamental rule

Implement the `fundamental-rule()` function. In this implementation, the function is called with an active and passive edge which we already know to be adjacent. If your implementation is correct, you should see:

```
? (setf toy (read-grammar "~/toy.mrg"))
? (print (edge-to-tree (parse '("Frodo" "lives") toy)))
→ (START (S (NP (NNP "Frodo")) (VP (VBZ "lives"))))
```

3 Viterbi over a PCFG parse forest

The `parse()` function produces a packed edge, which could represent multiple trees. Our previous example only has one complete analysis according to our toy grammar. Now we can try an ambiguous input:

```
? (print (edge-to-tree (parse '("Frodo" "adores" "the" "ring" "in" "Oslo") toy)))
→ (START
  (S (NP (NNP "Frodo"))
   (VP (VBZ "adores")
      (NP (DT "the") (NN "ring") (PP (P "in") (NP (NNP "Oslo")))))))
```

Depending on how the rules are stored in your grammar, you may see a different tree, since only the first tree found will be printed. The `viterbi()` function runs the Viterbi algorithm over the packed forest represented by an edge and returns an edge representing the most likely tree:

```
? (print (edge-to-tree (viterbi (parse '("Frodo" "adores" "the" "ring" "in" "Oslo") toy))))
→ (START
  (S (NP (NNP "Frodo"))
    (VP (VBZ "adores") (NP (DT "the") (NN "ring"))
     (PP (P "in") (NP (NNP "Oslo"))))))
```

(a) Study the `viterbi()` function in our skeleton code. In a few sentences, describe what the function is doing.

(b) Compare how the Viterbi algorithm works over a packed forest and an HMM trellis. What is the same? What is different?

4 Testing the parser

To test our Viterbi algorithm, we evaluate how well the parser output compares to gold standard annotations in our test set. The function `pcfg-eval()` reads in gold standard trees from a test file, extracts the leaves of the tree and provides them as input to our `parse()` function, and then compares the output against the original tree, using the ParsEval metric. A keyword argument **:baseline** allows us to evaluate how well the parser would do without Viterbi, i.e., just returning the first tree found. Run the evaluation function with and without the **:baseline** option. On an average laptop, it might take 1–2 minutes each time.

```
? (setf wsj (read-grammar "~/wsj-train.mrg"))
? (pcfg-eval "~/wsj-test.mrg" wsj :baseline t)
? (pcfg-eval "~/wsj-test.mrg" wsj)
```

(a) What does the output mean? Is Viterbi making a difference to the parser accuracy? Even if you can't run the commands, briefly explain what ParsEval measures, and also what is meant by coverage.

(b) The `pcfg-eval()` function has another keyword argument **:maxlength**, which defaults to 10. Only sentences with a length less than or equal to **:maxlength** are evaluated. Using `time()`, compare the difference between using a **:maxlength** of 9 or 10.

```
? (time (pcfg-eval "~/wsj-test.mrg" wsj :maxlength 9))
? (time (pcfg-eval "~/wsj-test.mrg" wsj :maxlength 10))
```

How many inputs are parsed each time? What is the total evaluation time for each evaluation run?

(c) In a few sentences, discuss factors that can affect parsing time. What could we do during training to improve parsing time?

Happy coding!