Language and Speech Processing

Final Project

PCFG Parser

Roberto Valenti, 0493198
Rómulo Gonçalves, 0536601

Introduction and Problem Description

The use of statistical methods in areas such as language processing, speech recognition and grammar learning, switched from being virtually unknown to being a fundamental approach in the last ten years [1]. Thanks to this, we now have available a big number of corpora which we can use to extract statistical information and can help us to understand the underlying structures of the languages. In fact, these data sets are usually annotated and we can use these annotations, together with statistics, in order to spot certain regularities or frequent characteristics of a language.

Within the multiple tasks that can be performed with such knowledge, the target of this project is the machine reconstruction of the syntax (parse) of a sentence. We would like to understand which rules generated a sentence, in order to understand which class of sentence we are dealing with.

To better explain the task, we can use the noisy channel model [8]: In this model (represented in figure 1), we assume that **A** has a concept and wants **B** to receive this concept. In our case this concept is the correct parse for a sentence. **B** does not receive the full parse, but just the sentence, so he needs to reconstruct the correct parse that **A** had in mind starting from the observations he gets through the noisy channel (in figure 1, this is represented by the dashes without the parse).

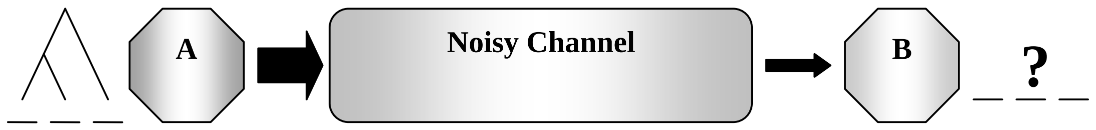


Figure 1*: The noisy-channel model*

B will have to use the sentence in a parser to generate all possible parses for the given sentence. The disambiguator will then decide which one of the proposed parses is the most likely, basing its assumption some kind of model. This is shown in figure 2.

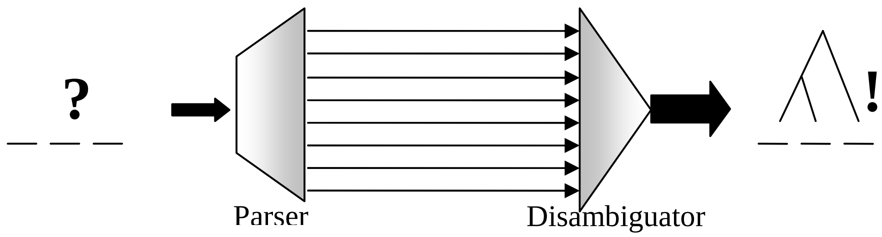


Figure 2*: A typical disambiguation approach*

In a brute force approach, the parser will have to enumerate the possible parses by combining possible tags. The disambiguator will then have to pick the most likely one. This approach is clearly inadequate for medium and for big grammars. In this project we will deal with this problem.

The next sections of the report are structured as follows: we will first describe the required steps for the implementation of the parser, giving some background theory for the given tasks together with some pseudo codes of the relevant implemented functions; we will then discuss some of the implementation choices and details together with the description of the simple interface we implemented for this project. Furthermore, we will illustrate and evaluate some experiments we performed with the final parser. Finally, before come to a conclusion, we will start a discussion about the results obtained and about the decisions we took during the developing phases.

Formalization of the Solution

The aim of this project is a full implementation of a probabilistic context free grammar parser from scratches. To tackle this goal, the following main steps are required:

- Extract a CFG from the Treebank
- Transform the CFG in PCFG
- Transform the PCFG in CNF grammar
- Use the CNF grammar in a CYK parser.

In this session we will explain each one of the step involved, together with some theory behind each step.

Context Free Grammars

A good model that approximates the natural language syntax is the context free grammars (CFG). A context-free grammar G can be defined as a 4-tuple:

G = (V_t,V_n,P,S) where
- V_t is a finite set of terminals
- V_n is a finite set of non-terminals
- P is a finite set of productions rules
- S is an element of V_n, and represents the start symbol.
- elements of P are of the form $V_n \rightarrow (V_n \cup V_t)^*$

The term "context-free" comes from the fact that the non-terminal V can always be replaced by an element of (V_n U V_t), regardless of the context in which it occurs.
Context-free grammars are simple enough to allow the construction of efficient parsing algorithms which, for a given string, determine whether and how it can be generated from the grammar. The problem of this grammar is that it misses the probabilistic model which is needed in order to disambiguate between parses. Let us introduce the probabilistic version of the context free grammars (PCFG), which defines the language as a probability over strings [6]. This is basically a CFG with probabilities associated with each rule, indicating how probable a production rule is.

A production rules is a way to represent the grammar using a condition-action pair of the form if (condition) then (action). In our case, the condition is the root and the action is the children of this root. For example, if we have the rule "*A -> B C*", this means that every time we find the symbol A, it can be substituted with the symbol B and C.

The simplest way to gather statistical information about a CFG is to count the number of times each production rule is used in a corpus containing parsed sentences and to use this in order to estimate the probability of each rule being used. In our case, we estimate the rules probabilities using the relative frequency of the rule in the training set.
For a generic rule "*A -> B C*", its conditional probability is defined as [5]:

$$P(\text{A->BC} \mid \text{A}) = \frac{\text{Frequency (A->BC)}}{\text{Frequency (A)}}$$

Once we have the probability of the production rules in a PCFG, the probability of a parse tree can easily be calculated by multiplying the probabilities of the rules that built its sub-trees [2]. A derivation of a sentence S is defined as sequence of one or more context free rewritings starting from TOP until we arrive at the sequence of terminals S.

Parsing the Treebank in a PCFG

For this project we used an excerpt from the Openbaar Vervoer Informatie Systeem (OVIS) treebank: In this treebank, a tree is represented as a recursive structure "*(A,[comma-list])*", where "*A*" is the label of the root node and "*comma-list*" is a comma-separated list of subtrees representing the ordered (left-to-right) list of child nodes of the root, each dominating a tree or a leaf-node " (*terminal_,[])*". The original Dutch treebank consists of 10,000 trees, so is larger than the English ATIS corpus. The mean sentence length in this treebank is 3.5 words per sentence, which is much shorter than the average in the ATIS corpus. Apart from a few questions, the sentences in this corpus are all imperatives or answers to questions.

The following algorithm is used by the implemented parser to parse the training Treebank in a CFG and to subsequently calculate the probabilities to transform it to a PCFG.

```
while (there are lines in the file) {
   Read the line and add the start symbol
   Parse(line)
   Calculate probabilities
   Create the PCFG
}
Parse (String tree) {
      Extract the root
      Extract the children chunk
      for (all the characters in the children chunk) {
            count the number of opened and closed brackets
            if (number of open and closed brackets==0){
                  recursive step:
                  children[++]= Parse (content between brackets) )
            }
      }
      Remove unary production
      Add the new rule to the structures
      return root
}
```

In few words, the recursive part of the algorithm passes sequentially through the string and, after it has extracted the root of the tree (which correspond to the first encountered element), calls itself on each of the found children in the children chunk. The parse function returns the root of the current string, so that the calling function can use this information to give the name to new child.
It is important to note that, on each tree in the Treebank, we added the "TOP" symbol as main tree root, which is the start symbol of our CFG grammar. In this way we can infer statistical information about how a sentence starts. The TOP symbol will later help to disambiguate between possible parses of a sentence.
The parsed rules are then collected in two Hashtables, each of which respectively counts the frequency of the rule's root in the corpus and the frequency of the rule itself;
The function will successively use this information to calculate the probabilities of each rule an in [2]. Finally, the function transforms the content of the Hashtable in an array containing all the rules of the newly generated PCFG;

Converting the PCFG to Chomsky Normal Form

After the rules are extracted from the Treebank and a probability is assigned to them, the next necessary step, in order to be handily used in further operations, is to transform the grammar in Chomsky Normal Form (CNF).
A grammar is in CNF only if its rules are in the form:

- A -> a
- A -> B C

where A,B,C are non-terminals and a is a terminal symbol.
By transforming a grammar in CNF, rules of the form "*A -> B C D E (p)*", where p is the probability, are replaced by a set of rules:

- A -> B C1 (p)
- C1-> C D1 (1)
- D1-> D E (1)

The steps in converting a standard PCFG into CNF are the following:

1. Remove Empty Productions
2. Remove Unit Productions
3. Change grammar so all rules are in CNF

The dataset we used was lacking of empty productions so we didn't worry about them, the second step is performed during the extraction of the grammar from the Treebank, while the third step is performed by the function "toCNF", which can be summarized in the following pseudocode:

```
toCNF(rules) {
    for all rules {
        if there are more than 2 children{
                save the root
                for all children -1 as j{
                        child 1= j
                        child 2=j+1
                        if is not the last child {
                                while the child is unique, add a postfix to the child
                        }
                        add the new rule, with probability 1 if is not the first;
                        root = child 2
                }
        } else add the rule;
    }
    Update the rules;
}
```

Once the function was implemented, we noticed that a lot newly generated rules could have been removed since similar rules were already in the grammar.
One valid example a rule repetition can be given by analyzing the transformation of the following rules:

- A -> B C D E(p)
- F -> B C D E(p2)

Applying the algorithm, the rules are transformed in:

- A -> B C1(p)
- C1 -> C D1 (1)
- D1 -> D E (1)

- F -> B C2(p2)
- C2 -> C D2 (1)
- D2 -> D E (1)

It is clear that the first of the second rule set could be transformed to “F -> B C1(p2)” without loosing any information in the grammar and reducing the number of generated rules of 50%

Given the small Treebank on which we trained our parser, this problem is irrelevant, but for the sake of learning we decided to include some modifies to the algorithm in such a way that it was possible to reduce the grammar size to its lower bound.
The first attempt we did to tackle our goal was to implement stand-alone function which spotted these duplication in the transformed rules to later fix them. The problem of this first implementation was that the advantages of using this function were not worth its computational complexity. We then decided to reduce the grammar “on the fly” by using an additional structure in the proposed algorithm while transforming the grammar in CNF. The modified algorithm stores all visited sequences of children and, in case a new rule has an already visited sequence, it links the rule to the known sequence. With the discussed additions, the normalized grammar was reduced of 72 rules without relevant additional computational costs. For further implementation details, please refer to the source code.
Appendix A shows the CNF transformation (using the described algorithm) of the toy CFG 12.36 on page 458 of [2], previously transformed to a PCFG.

The CYK algorithm

The Cocke-Younger-Kasami (CYK) algorithm determines whether or not a string can be generated by a given context-free grammar and, if so, how it can be generated. This process is known as "parsing" [6]. The standard version of CYK recognizes languages defined by context-free grammars written in Chomsky normal form (CNF). Since any context-free grammar can easily be converted to CNF, CYK can be used to recognize any context-free language. It is also possible to extend the CYK algorithm to handle some context-free grammars which are not written in CNF, but this increases the difficulties that can be encountered in the algorithm implementation.
The computational complexity of the CYK is $O(n^3)$, where n is the length of the parsed string, This, even if it seems expansive, makes the CYK one of the most efficient algorithms for recognizing any context-free language if compared with previously used algorithms.
The pseudo code of the used CYK algorithm is shown below, implemented as described in [4] and explained in [8] with some implementation additions and optimizations.

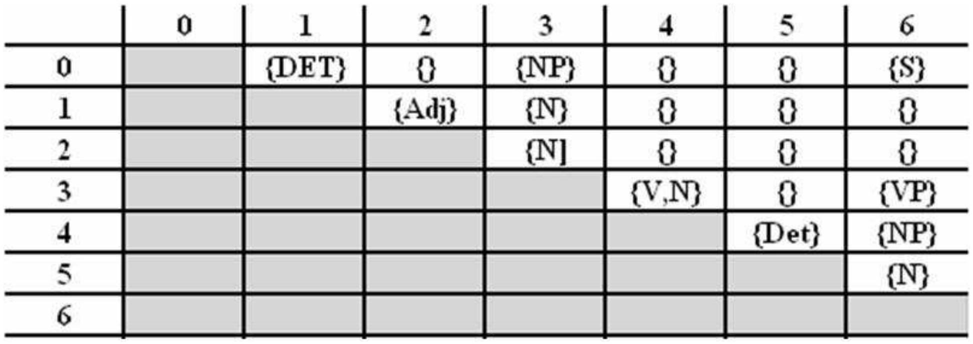
```
CYK (sentence, rules)
{
    n= size of the sentence
    Create a n+1* n+1 table;
    for 0< k<n {
        If (A->k-th word in the sentence) belongs to the grammar
            Add the left side to table[k-1][k]
        for 0=< i < k – 2 {
            for i<j<k
        {
        B= list of RHS in table[i][j];
        C= list of RHS table[j][k];
        For all combination of values in B and C
                If (A->BC) belongs to the grammar
                    Add the LHS to table[i][j]

        }
    }
    If  table[0][n] contains A and TOP->A belongs to the grammar
        true
    else
        false
}
```

The CYK algorithm takes as input a sentence and a grammar in CNF.
The algorithm uses a table structure (only half of it) to store partial parses, and loops through this structure to enumerate possible combination of these partial parses.
To better explain how the CYK algorithm works, without getting too much in the implementation details, let's use a basilar example on a small grammar. We want the CYK algorithm to check if the sentence "the new student bought the book" can be generated by the following grammar:

- S -> NP VP
- VP -> Vt NP
- NP -> Det N
- N -> Adj N
- V -> saw
- N -> saw
- Det -> the
- Det -> a
- N -> book
- N -> student
- Adj -> new

In the initialization step, the algorithm checks for possible LHSs (left hand sides) for the terminal words in the sentence, placing them in the "diagonal 0" (the left-most diagonal) shown in figure 3. Note that the word "saw" appears in the grammar as "V" or "N", so both values appear in column 4.



	0	1	2	3	4	5	6
0		{DET}	{}	{NP}	{}	{}	{S}
1			{Adj}	{N}	{}	{}	{}
2				{N]	{}	{}	{}
3					{V,N}	{}	{VP}
4						{Det}	{NP}
5							{N}
6							

***Figure 3**: The CYK table*

In the second phase, the algorithm looks for the production rules in the grammar that covers the filled LHSs (on diagonal 0) for adjacent words. In this case, the only available production rules are "*N -> Adj N*" and "*NP -> Det N*". This operation is then repeated for the other cells in the table, varying the values of "i" and "j", and iterating over different values of "k",. This concept is graphically represented in figure 4, where the algorithm looks for a LHS A in order to combine the subtrees B and C, which together cover the input sentence from "i" to "j"

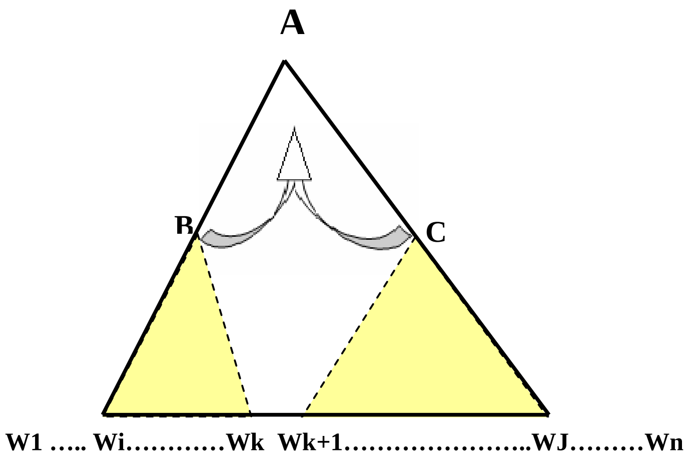


***Figure 4**: Partial parses combination*

When the algorithm terminates, the content of the most right-top cell of the table can contain (on not contain) the root of the input sentence. The presence of a value in this cell confirms that the parse could be built from the original grammar, but it doesn't guarantee that it is a complete sentence. To further assess if the sentence is actually generated by in the grammar, one should check if this parse root appears as a starting symbol, which consist on verifying whether or not the grammar contains the parse root as a RHS (right hand side) of the starting symbol "TOP".
In the example in figure 4, the full parse generated by the CYK algorithm for the given input sentence is shown in figure 5.

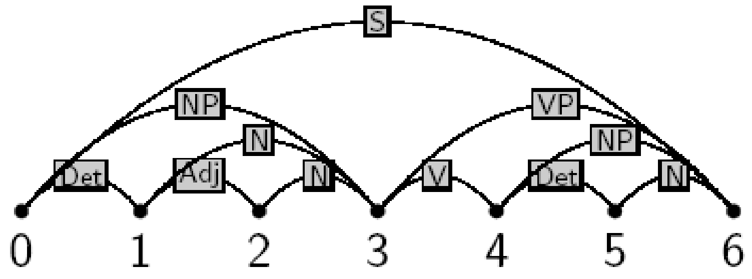


Figure 5*: Full parse for the example in figure 3*

A nice characteristic of the CYK algorithm is its adaptability to deal with probabilities. Until now, we discussed the CYK in its original form. Recalling the noisy channel model, we would like to use the CYK as a parser to generate all possible parse trees, and to use a probability model as disambiguator.
The first required modification to the CYK algorithm consists on the integration of probabilities in the partial parses. We define the probability of an arbitrary parse tree as the Inside-probability [2][3][8] of the parse tree:

$$Inside(\langle i, A, j\rangle) = P(A \rightarrow^{*} w_{i+1}...w_{j} \mid A)$$

The inside probability at the top node can then represent the full sentence probability in the given grammar:

$$P(w_1,...,w_n \mid G) = Inside(\langle 0, TOP, n\rangle)$$

This probability will be the sum of the probability of each sub tree in the parse. For the example shown in figure 3, this can be formalized as:

$$Inside(\langle i, A, j\rangle) := Inside(\langle i, A, j\rangle) + P(A \rightarrow BC \mid A) \times Inside(\langle i, B, k\rangle) \times Inside(\langle k, C, j\rangle)$$

If, every time that the CYK combines partial parses, we use the inside probability to have the probability of the new full parse, in the end of the algorithm, together with the possible parses, we will have the sentence probability.
We now want to change this new version of the CYK algorithm to be able to disambiguate the possible parses by returning the parse with the highest probability. In this case we don't need the sum of the probabilities for each sub parse; instead we are looking for the sub parses with maximum probability. The required modification is called "Viterbi-parse"[2][3][8] and makes use of the recursive definition of the inside probabilities. The difference is that, in the

Viterbi-parse, we select the sub parse with maximum probability instead of summing over all sub parses:

$$Viterbi(\langle i, A, j\rangle) = \arg\max_{der \in DER(w_{i+1}...w_j)} P(der = A \rightarrow^{*} w_{i+1}...w_j \mid A)$$

Beside this little modification, another required addition is a pointer structure to the RHSs involved in each found LHS. This is required when, once we obtained the possible parses through the CYK, we need to follow the “parse path” with the highest probability, starting from the “TOP” symbol.

Using the CYK algorithm on the sentence “the_ agency_ mail_ and_ the_ labor_ codes_” with the CNF PCFG shown in Appendix A. resulted in a sentence probability of 2.8001888465279303E-10. By using the Viterbi-parse modification to the CYK algorithm, the resulting suggested tree (in string representation) is:

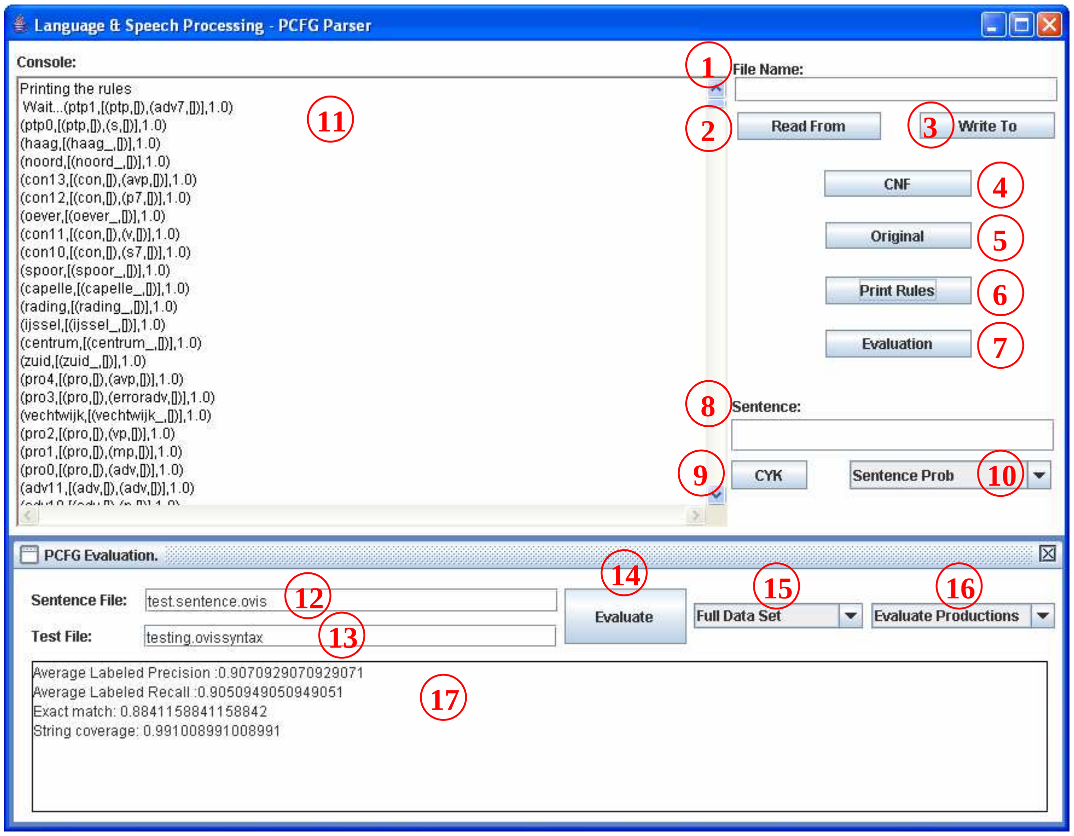
(NP,[(NP,[(DT,[(the_,[])]),(NBAR,[(N,[(agency_,[])]),(N,[(mail_,[])])])]),(CC,[(and_,[])]),(NP,[(DT,[(the_,[])]),(NBAR,[(N,[(labor_,[])]),(N,[(codes_,[])])])])]),

Implementation Details

For this project we used Java as our programming environment. This preference of using Java over other programming languages was mainly for its platform independency.
The cornerstone of our implementation is the Rule class, which contains all the attributes needed for a production rule and is implemented in such a way that is easy to perform the most common operations. For instance, a Rule class can be easily display its content in a string representation, or can be constructed passing the string representation in its constructor. The class with all the implemented function is Data, which loads, parses and converts the grammar as soon it is instantiated. To perform additional operations with the data, it is necessary to type them in the main class file, called PCFG, or to use the implemented user interface (see next session).
During the implementation phase we decided to use a “state” approach to the project.
By “state” approach we refer to an array containing all the rules shared in the environment. Every time one function is executed, usually it passes through this rule array, or an arbitrary rule array used as input, performs some operation in its own structures and finally updates the rule array with the result of the operation, modifying the state of the environment. One of the multiple advantages that came with this choice was a unification of the inputs and outputs format for the function, allowing us to easily create pipelines between them. Of course, once the graphical user interface was implemented, a lot of the power of this approach was lost by forcing those functions in graphical static buttons. The previous approach and its advantages are still accessible bypassing the interface and running the parser from the normal main file.

The Graphical User Interface

Figure 6 is a screenshot of the interface we implemented to easily use and debug some of the implemented parser features without recompiling the code every time. The interface can be shattered in three main components, the first one allows to perform data operations, the second one allows to use the parser and the last one is dedicated to evaluation operations.
The program automatically loads the default training Treebank, which is embedded in the application. If the user wishes to, the interface allows loading another dataset.



```
Printing the rules
 Wait...(ptp1,[(ptp,[]),(adv7,[])],1.0)
(ptp0,[(ptp,[]),(s,[])],1.0)
(haag,[(haag_,[])],1.0)
(noord,[(noord_,[])],1.0)
(con13,[(con,[]),(avp,[])],1.0)
(con12,[(con,[]),(p7,[])],1.0)
(oever,[(oever_,[])],1.0)
(con11,[(con,[]),(v,[])],1.0)
(con10,[(con,[]),(s7,[])],1.0)
(spoor,[(spoor_,[])],1.0)
(capelle,[(capelle_,[])],1.0)
(rading,[(rading_,[])],1.0)
(ijssel,[(ijssel_,[])],1.0)
(centrum,[(centrum_,[])],1.0)
(zuid,[(zuid_,[])],1.0)
(pro4,[(pro,[]),(avp,[])],1.0)
(pro3,[(pro,[]),(erroradv,[])],1.0)
(vechtwijk,[(vechtwijk_,[])],1.0)
(pro2,[(pro,[]),(vp,[])],1.0)
(pro1,[(pro,[]),(mp,[])],1.0)
(pro0,[(pro,[]),(adv,[])],1.0)
(adv11,[(adv,[]),(adv,[])],1.0)
```

Sentence File: test.sentence.ovis
Test File: testing.ovissyntax

```
Average Labeled Precision :0.9070929070929071
Average Labeled Recall :0.9050949050949051
Exact match: 0.8841158841158842
String coverage: 0.991008991008991
```

***Figure 6**: The Graphical User Interface*

To perform this operation, the file name of the personal tree bank has to be typed in (1) and the button (2) pressed. After a grammar is loaded from a file, it is possible to execute some operations with it. To convert it to a CNF grammar, the user just needs to press on (4), while to switch back to the original the user should press on (5). At any time, it is possible to print the current rules in the console (11) by pressing on (6).
After the grammar is in CNF, the possibility of using the parser becomes available. A sentence, with words separated by spaces, has to be typed in (8). The combo box (10) allows the user to select if he just needs to check if the sentence is in the grammar and its probability or if he wants the most probable parse for the input sentence. By pressing on (9) the parser's results with the defined setting will appear in (11).
In order to easily evaluate the performances PCFG Parser, the user should press on the evaluation button (7). By doing so, a new sub window will appear, which offers four different ways of evaluation. By default, the suggested test set is "test.sentence.ovis" is already typed in (12), which should in any case contain the filename of the unparsed test sentences. The same for the field (13), which requires the filename of the gold parses for the sentences read from (12). The combo box (15) and (16) allows the user to select the evaluation setting that will be extensively illustrated in the next session. After all the evaluation settings are selected, the user can press on (14) and the results will appear in the specific console (17).
As a special interface stability measure, every time that some error occurs or the user does some mistakes a small window will appear to inform the error or to show a warning message.

Evaluation

In order to test the accuracy of the parser, we implemented a function (accessible through interface) which takes as input a file with sentences and a file with the correct parse tree associated with the respective sentence. The function reads each sentence and calls the parser in order to propose a possible parse tree.The obtained parse tree is then compared to the tree read from the other file (the gold standard). This comparison is done through the following measures:

- exact-match
- string-coverage
- labeled precision
- labeled recall

The exact-match is the most straight-forward measure that can be implemented for the given task, since is obtained as the averaged result of the comparison of the string representation of the proposed parse and the gold parse, over all the test set.

$$\text{Exact Match } = \frac{\text{\# equal test and gold parses}}{\text{\# gold parses}}$$

The string coverage represents an indication of the number of the test sentences which are successfully parsed by our parser. By successfully parsed, we indicate a successful outcome of the CYK algorithm in recognizing the sentence as being generated from the grammar that we used for training.

$$\text{String Coverage } = \frac{\text{\#successfully parsed sentences}}{\text{\# sentences in the test set}}$$

Since we are dealing with trees, special comparison measures are needed in order to correctly evaluate the performances of the parser. To better understand why we need to introduce special measures, it is easier to start from an example. Let's suppose that, feeding our parser with the sentence "correctie_ leiden_ venlo_", the suggested parse is:

(a,[(np,[(n,[(correctie_,[])]),(np,[(leiden_,[])])]),(np,[(venlo_,[])])])

while the correct (gold) parse is indicated in the test set as:

(a,[(n,[(correctie_,[])]),(np,[(np,[(leiden_,[])]),(np,[(venlo_,[])])])])

It is clear that the two parses are somewhat correlated, but it is difficult to see how much just from this string. In a graphical tree representation, the two parses appear as in figure 7.

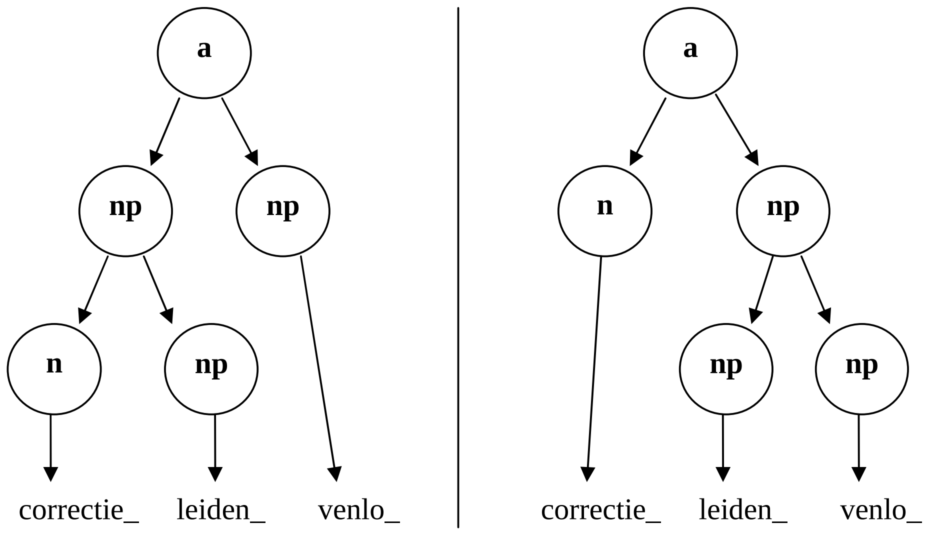


Figure 7*: Proposed Vs Gold parse for the sentence "correctie_ leiden_ venlo_"*

It is easy to note that, while the root and the leaves are correct in both cases, the underlying knowledge of the tree is slightly wrong. Let us define a constituent of a parse: from wikipedia [6]*"the term constituent is used in syntactic analysis to refer to a single word or a group of words that function together as a unit and are embedded into a hierarchical structure"*.
We can play a bit with this definition of constituent, defining a "coverage" constituent and a "production" constituent. While the production constituent is characterized by a root and its production (or its direct children), the coverage constituent is characterized by a root and the part of the sentence that it covers.
The coverage for the parses in figure 7 is shown in table 1, while their production is represented in table 2.

Proposed coverage			**Correct coverage**		
A	1	3	A	1	3
NP	1	2	NP	2	3
N	1	1	N	1	1
NP	2	2	NP	2	2
NP	3	3	NP	3	3

Table 1: *Coverage analysis*

Proposed productions			**Correct productions**		
A	NP	NP	A	N	NP
NP	N	NP	NP	NP	NP
N	correctie_		N	correctie_	
NP	leiden_		NP	leiden_	
NP	venlo_		NP	venlo_	

Table 2: *Production analysis*

The red values in table 1 and 2 indicate that the correspondent constituent couldn't be found in the gold parse.

We can use this information in measures known as ParsEval, which are very similar to the normal precision and recall measures, but adapted in order to evaluate trees structures.
ParsEval comprehend labeled precision and labeled recall, defined as follows:
Labeled Precision is the ratio between the number of correct constituents in the proposed parse and the number of constituents in the proposed parse, while the labeled recall is given by the ration between the number of correct constituents in the proposed parse and the number of constituents in the gold parse.
In the case of the given example, the labeled precision and the labeled recall for coverage evaluation will be computed as:

$$\text{Labeled precision } = \frac{\text{\# correct constituents}}{\text{\# constituents in the tree}} = \frac{4}{5} = 0.8$$

$$\text{Labeled recall } = \frac{\text{\# correct constituents}}{\text{\# constituents in gold tree}} = \frac{4}{5} = 0.8$$

One might note that in the given example that the precision equals the recall both cases (coverage and production evaluation). This is because the number of constituents in the proposed tree and the number of constituents in the gold parse are the same.

Experiments

While testing our parser on the suggested test set, which consists of 1001 sentences and respective parses, we realized that a lot replicated sample were used. One very obvious example can be seen by searching the words “nee_” and “ja_” and counting how many times the words appear as test sentences.
Removing the repeated sentences and parses, the size of the suggested test set drops to 618 instances. From now on, we will refer to this version of the test set as “clean”.
In order to extensively analyze the performances of the implemented parser, we decided to evaluate it on both “normal” and “clean” test set (a discussion about this decision can be found on the next session). Table 3 shows the differences between the exact match and the String coverage obtained in the two experiments.

	Exact Match	**String Coverage**
Clean Test Set	81.23%	98.54%
Full Test Set	88.41%	99.10%

Table 3: *Exact match and String coverage for the two test set versions*

The other two analyzed measures, average labeled precision and average labeled recall, are represented in table 4 for the clean test set, and in table 5 for the full test set. In these tables, both the rule coverage and productions rules are compared.

	Rule Coverage	**Productions**
Average Labeled Precision	81.72%	85.00%
Average Labeled Recall	81.72%	84.63%

Table 4: *Performances on the clean test set*

	Rule Coverage	**Productions**
Average Labeled Precision	88.71%	**90.71%**
Average Labeled Recall	88,71%	90.51%

Table 5: *Performance on the full test set*

As an additional way to have some insight about how the parser is dealing with data, we analyzed the average labeled precision and recall per sentence length on the full test set. The results are shown graphically in figure 8 for production analysis and for coverage analysis.

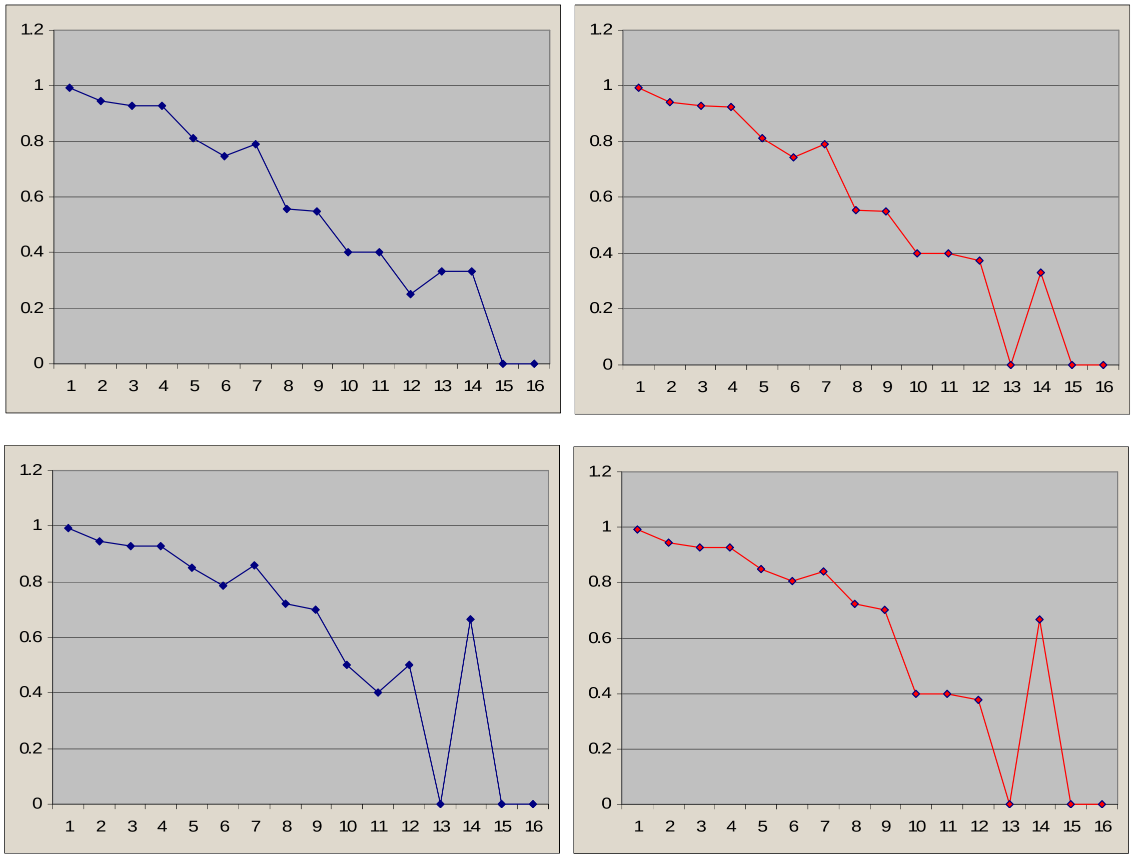


Figure 8*: Labeled precision (left) and labeled recall (right) for sentence length.*
Top: production evaluation, bottom: coverage evaluation

Discussion

The choice of analyzing the performances on both clean and full test set came from the observation that the test set is probably an application outcome, which reflects the probability to encounter the represented sentences in the designed task, during a day or even its lifetime. By choosing to test both versions of the test set, we actually chose to measure the performances of the parser per se and the performance of the parser in the context of the task it was trained for (a public transports information system). Analyzing the results of the experiments in tables 3, 4 and 5, it is immediate to notice the amount of differences between the coverage and the production analysis and between the full and the clean test set. An improvement between 5% and 7% is achieved using full test set. This is probably because the repetitions we removed on the clean test set were mainly small and very frequent sentences, and this gave more weight to errors performed on longer sentences if compared with the weights for these sentences in the full dataset. This observation is also reflected in the graphs in figure 8, which show a drop in both precision and recall for 8 or 9 words long sentences, and instability after 12 words long sentences. We think that the reason behind this instability

can be found in the difference between the number of long sentences and the average size of the training set. In fact, it is difficult for a grammar learned from very small sentences, to correctly cover a long sentence. This is why the biggest part of the sentences that were not found in the grammar are the longest ones, generating these big jumps in precision and recall in the graphs.
Speaking about the coverage and production analysis comparison, the former analysis always achieves 2% lower labeled precision and recall than the production analysis. Of course, even if the difference is just 2%, the 90.7% obtained by the production analysis sounds much better than the 88.7% of the coverage analysis, but we believe that the latter reflect the nature of the parses and the real performances of the parser in a much better and strict way.

Beside on the obtained results, another important discussion should be made about some implementation details that we didn't want to add directly in the report: In order to deal with unseen words, a straight forward smoothing technique was implemented, which assigns to this word a list of all know tags and let the algorithm pick up one that will actually push the sentence to be in the grammar. This solution helped us improve mainly the string coverage, which went from 96% to 99% on the full test set. The other measures did not change significantly, since the correct partial parses remained more or less the same, and the incomplete parses were already included in the evaluation. Of course, using this kind of smoothing can introduce significant errors: for instance, a sentence which is completely out of the grammar will have a parse (even if completely wrong), while some other sentence with some word that are actually in grammar can result as not belonging to the grammar. Choosing whether or not to use this kind of smoothing is completely task dependent.
Thinking about the task reflected in the Treebank, we thought about improving the results by adding a little human knowledge in the simple smoothing technique. In fact, looking at the training set, it seems that the OVIS is often using numbers which represent time values. Since, probably, not all the possible numbers were in the training set, we decided to implement a little string parser which recognized all the numbers from zero to sixty (all possible numbers in a clock and a calendar). When a number was recognized, the smoothing added a single "NUM" or a "NUM" and a "DET" tag instead of the full list of known tags. This human knowledge addition improved all measures (besides string coverage) from 0.1% to about 1%. Given this, it would be interesting to embed in the implemented parser the POS tagger implemented in [7] in order to "translate" the sentences of another task and analyze them with this parser. Of course, since the grammar rules will be different, we cannot hope to create a general parser with this "hack" to the terminals, but this will surely help to slightly increase its performances for different task rather than the task that the parser was originally trained for.

Conclusions

In this project we successfully implemented a full PCFG parser. The parser performed pretty well for the OVIS task on which it was trained, but we believe it won't perform as well if used for other tasks (i.e. parsing normal Dutch sentences) without a proper, new and task-specific training. Together with a simple smoothing technique and a non expansive way to reduce the grammar when is transformed to CNF, we additionally implemented a usable interface for the program which allows users to interact end experiment with the implemented parser without changing and recompiling the code. Finally, we tried to boost the performances of the parser integrating some human knowledge in the simple smoothing for unseen words, obtaining a little improvement on both labeled precision and recall.

References and Bibliography

[1] Abney, Steven, *Statistical Methods and Linguistics*, Judith Klavans and Philip Resnik, eds., The Balancing Act. MIT Press, Cambridge, MA. (1996)

[2] Manning, Christopher D and Schütze, Hinrich, *Foundations of Statistical Natural Language Processing*, MIT Press, Cambridge, MA (1999)

[3] Daniel Jurafsky and James H. Martin, *Speech and Language Processing*, Prentice-Hall, (2000)

[4] Collins, Michael, *A New Statistical Parser Based on Bigram Lexical Dependencies*, proceedings of ACL (1996)

[5] Detlef Prescher, Remko Scha, Khalil Sima'an and Andreas Zollmann.*Treebank Grammars and. Other Infinite Parameter Models* Institute for Logic, University of Amsterdam (2004)

[6]Wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Main_Page

[7] Roberto Valenti, Romulo Gonçalves, *Hidden Markov Model POS Tagger*, Language and Speech processing midterm project, University of Amsterdam, (2005)

[8] Khalil Sima'an, *Language and Speech processing course slides*, University of Amsterdam, (2005)

Appendix A - Toy CFG transformed to CNF

```
(AP,[(A,[]),(RBA,[])],1.0)
(TOP,[(V,[])],1.1010790574763268E-4)
(TOP,[(S,[])],1.1010790574763268E-4)
(TOP,[(RB,[])],1.1010790574763268E-4)
(TOP,[(AP,[])],1.1010790574763268E-4)
(TOP,[(P,[])],2.2021581149526536E-4)
(TOP,[(VBG,[])],2.2021581149526536E-4)
(TOP,[(DT,[])],2.2021581149526536E-4)
(TOP,[(NBAR,[])],5.505395287381634E-4)
(TOP,[(CC,[])],1.1010790574763268E-4)
(TOP,[(NP,[])],3.3032371724289804E-4)
(TOP,[(PP,[])],1.1010790574763268E-4)
(TOP,[(N,[])],8.808632459810614E-4)
(TOP,[(VPG,[])],1.1010790574763268E-4)
(TOP,[(A,[])],2.2021581149526536E-4)
(TOP,[(VP,[])],3.3032371724289804E-4)
(VBG,[(handling_,[])],0.5)
(VBG,[(controlling_,[])],0.5)
(PP0,[(PP,[]),(PP,[])],1.0)
(NP1,[(NP,[]),(PP0,[])],1.0)
(NP0,[(NP,[]),(PP,[])],1.0)
(RB,[(rapidly_,[])],1.0)
(DT,[(the_,[])],0.5)
(DT,[(a_,[])],0.5)
(V,[(sees_,[])],1.0)
(S,[(NP,[]),(VP,[])],1.0)
(P,[(as_,[])],0.5)
(P,[(of_,[])],0.5)
(N,[(use_,[])],0.125)
(N,[(volume_,[])],0.125)
(N,[(costs_,[])],0.125)
(N,[(agency_,[])],0.125)
(N,[(mail_,[])],0.125)
(N,[(codes_,[])],0.125)
(N,[(labor_,[])],0.125)
(N,[(way_,[])],0.125)
(PP,[(P,[]),(NP,[])],1.0)
(VP,[(VBZ,[]),(NP1,[])],0.3333333333333333)
(VP,[(VBZ,[]),(NP0,[])],0.3333333333333333)
(VP,[(VBZ,[]),(NP,[])],0.3333333333333333)
(A,[(widespread_,[])],0.5)
(A,[(growing_,[])],0.5)
(CC0,[(CC,[]),(NP,[])],1.0)
(CC,[(and_,[])],1.0)
(NP,[(DT,[]),(NBAR,[])],0.5)
(NP,[(NP,[]),(CC0,[])],0.5)
(VPG,[(VBG,[]),(NP,[])],1.0)
(NBAR,[(NBAR,[]),(PP,[])],0.3333333333333333)
(NBAR,[(AP,[]),(NBAR,[])],0.3333333333333333)
(NBAR,[(N,[]),(N,[])],0.3333333333333333)
```