Statistical methods for Categorial Grammars

Jeroen Bransen
Utrecht University

Supervisor: Michael Moortgat

July 2008

Contents

0 Introduction

The research after machines or computers that can understand natural language has been going on in the field of Artificial Intelligence for decades. Many approaches have been taken to accomplish this, resulting in quite a few successes. Although the systems work in all kinds of different ways, there are certain problems that all systems face. One of these problems is the so called ambiguity, where a sentence can have multiple meanings. Human beings are often very good in finding the correct meaning of the sentence, but for computers this is not as easy as it might look.

A solution to this problem would be very useful for Artificial Intelligence, and in this thesis I will outline a possible approach to this problem. To accomplish this I will use the linguistic framework of categorial grammars, a fully logical approach.

The advantage of a logical approach compared to other approaches is that it is fully lexicalized, so there are no rules. While in, for example, Context Free Grammars one needs both a lexicon and a set of rules, the logical approach only needs a lexicon. Another advantage related to this is that there exist algorithms for extracting the lexicon from an annotated treebank. This means that the lexicon can be generated automatically and very little work has to be done to get the lexicon.

The third advantage is that in the logical approach the relation between syntax and semantics is very clear, and that it allows so called deep dependencies. That is, the deep relations between the words in the sentence are represented within the derivations.

Within the CKI Bachelor this thesis can be placed in the field of the Computational Linguistics. It combines ideas from the courses *Parsing as deduction* and *Computational Grammars*.

1 The problem

In linguistics, people have been trying to build, for many years, computer programs that analyse sentences and can separate well-formed sentences from ill-formed sentences. The process of analysing sentences (or parts of sentences) and creating *derivations* (often tree-structures) is called parsing. A well known problem in parsing *natural languages* is that there is a lot of *ambiguity*. In parsing there usually is a *lexicon* with the words, a *grammar* that specifies how to derive sentences, and a parsing algorithm that uses the grammar to generate (or recognize) sentences. When there are multiple ways to derive the same sentence using the lexicon and the grammar, that sentence is called *ambiguous*.

In this thesis I will consider three types of ambiguity: *lexical ambiguity*, *structural ambiguity* and *spurious ambiguity*.

1.1 Lexical ambiguity

Lexical ambiguity is the most common form of ambiguity, and it means that one word has multiple entries in the lexicon. In daily life this kind of ambiguity often appears when a word has multiple meanings. Consider the word "*letter*", which can either mean "a symbol in the alphabet" or "a written message". Now look at the following sentence:

> John reads the letter.

Because the word "letter" has two different meanings, the whole sentence has two different meanings. This sentences therefore is called *lexically ambiguous*.

1.2 Structural ambiguity

Structural ambiguity occurs when a sentence has more then one underlying structure. An example is:

> The Dutch history teacher smiles

This sentence can have two different meanings:

1. There is a Dutch person teaching history and he smiles
2. There is a person teaching Dutch history and he smiles

1.3 Spurious ambiguity

Spurious ambiguity is the kind of ambiguity where there are multiple ways of deriving the sentence with the given lexicon, grammar and parse method, while the derivations are essentially the same. This means that, although the representation of the derivations can be different, the meaning of the derivations is the same. So with this kind of ambiguity the sentence itself does not necceserily have to be ambiguous, but the derivations of the sentence are.

1.4 This thesis

When reading ambiguous sentences, human beings usually have a lot less trouble deciding which reading (or derivation) is preferred than computer programs have. Let us look at an example sentence:

> I see the man with a knife.

In this case, there are 2 readings, due to structural ambiguity:

1. The case where the man has a knife, this is the preferred reading
2. The case where I use the knife to see the man, like in "I see the man with the binoculars"

In this thesis I will try to add some steps to existing parsing methods in order to (partially) solve the problem of ambiguous sentences and derivations. I will do this by preferring certain derivations to others using statistical methods.

I will first explain some of the techniques used in solving the problem. I will give a short introduction to Categorial Grammars and the representation of the derivations. I will continue with explaining the linguistical framework of proofnets, which already solves the spurious ambiguity. After this I will show a few other cases where statistical methods have been used in parsing, and I will explain how I will use these ideas in categorial grammars.

Then I will first focus on generating a type-logical lexicon from a treebank. This allows me to test the algorithms I will find later on. The next step is to ignore the lexical ambiguity and focus on a solution for the structural ambiguity first. When the solution for structural ambiguity has been found, I will address the problem of lexical ambiguity. In the last chapter I will display the results of the tests and give some ideas for future research.

2 Methods

2.1 Categorial Grammars

The linguistic framework that I will use for derivations is the framework of *categorial grammars*. With this method, each word gets a (or usually several different) type(s), and there are a few inference rules that allow you to combine the types of the word to derive a sentence. For displaying the proofs I will use the Gentzen calculus style presentation. The associative rules from *(Lambek, 1958) [5]* are shown in figure 1.

$$\frac{}{A \Rightarrow A}\,[Ax] \qquad \frac{\Delta \Rightarrow A \quad \Gamma, A, \Gamma' \Rightarrow C}{\Gamma, \Delta, \Gamma' \Rightarrow C}\,[Cut]$$

$$\frac{\Delta, B \Rightarrow A}{\Delta \Rightarrow A/B}\,[/R] \qquad \frac{\Delta \Rightarrow B \quad \Gamma, A, \Gamma' \Rightarrow C}{\Gamma, A/B, \Delta, \Gamma' \Rightarrow C}\,[/L]$$

$$\frac{B, \Delta \Rightarrow A}{\Delta \Rightarrow B\backslash A}\,[\backslash R] \qquad \frac{\Delta \Rightarrow B \quad \Gamma, A, \Gamma' \Rightarrow C}{\Gamma, \Delta, B\backslash A, \Gamma' \Rightarrow C}\,[\backslash L]$$

$$\frac{\Gamma, A, B, \Gamma' \Rightarrow C}{\Gamma, A \bullet B, \Gamma' \Rightarrow C}\,[\bullet L] \qquad \frac{\Delta \Rightarrow A \quad \Delta' \Rightarrow B}{\Delta, \Delta' \Rightarrow A \bullet B}\,[\bullet R]$$

Figure 1: Gentzen calculus style inference rules for the associative Lambek calculus

The lexicon consists of a list of words and types. If a word has multiple types, it will occur multiple times in the list. An example grammar is:

```
i :: np.
see :: ((np\s)/np).
with :: ((n\n)/np).
with :: (((np\s)\(np\s))/np).
the :: (np/n).
knife :: n.
man :: n.
```

With this grammar we can for example derive the sentence *I see the man with the knife* of type `s` as follows:

$$\dfrac{n \Rightarrow n \quad \dfrac{np \Rightarrow np \quad \dfrac{np \Rightarrow np \quad \dfrac{n \Rightarrow n \quad \dfrac{np \Rightarrow np \quad \dfrac{s \Rightarrow s}{np,\ (np\backslash s) \Rightarrow s}(\backslash L)}{np,\ ((np\backslash s)/np),\ np \Rightarrow s}(/L)}{np,\ ((np\backslash s)/np),\ (np/n),\ n \Rightarrow (np\backslash s)}(/L) \quad \dfrac{np \Rightarrow np \quad s \Rightarrow s}{np,\ (np\backslash s) \Rightarrow s}(\backslash L)}{np,\ ((np\backslash s)/np),\ (np/n),\ n,\ (n\backslash n) \Rightarrow s}(\backslash L)}{np,\ ((np\backslash s)/np),\ (np/n),\ n,\ ((n\backslash n)/np),\ np \Rightarrow s}(/L)}{np,\ ((np\backslash s)/np),\ (np/n),\ n,\ ((n\backslash n)/np),\ (np/n),\ n \Rightarrow s}(/L)$$

Because the word `with` has 2 lexical entries, we could have also chosen the other type for the word `with`. This can for example result in the following derivation:

$$\dfrac{n \Rightarrow n \quad \dfrac{np \Rightarrow np \quad \dfrac{n \Rightarrow n \quad \dfrac{np \Rightarrow np \quad \dfrac{np \Rightarrow np \quad \dfrac{s \Rightarrow s}{np,\ (np\backslash s) \Rightarrow s}(\backslash L)}{(np\backslash s) \Rightarrow (np\backslash s)}(\backslash R) \quad \dfrac{np \Rightarrow np \quad s \Rightarrow s}{np,\ (np\backslash s) \Rightarrow s}(\backslash L)}{np,\ (np\backslash s),\ ((np\backslash s)\backslash(np\backslash s)) \Rightarrow s}(\backslash L)}{np,\ ((np\backslash s)/np),\ np,\ ((np\backslash s)\backslash(np\backslash s)) \Rightarrow s}(/L)}{np,\ ((np\backslash s)/np),\ (np/n),\ n,\ ((np\backslash s)\backslash(np\backslash s)) \Rightarrow s}(/L)}{np,\ ((np\backslash s)/np),\ (np/n),\ n,\ (((np\backslash s)\backslash(np\backslash s))/np),\ np \Rightarrow s}(/L)}{np,\ ((np\backslash s)/np),\ (np/n),\ n,\ (((np\backslash s)\backslash(np\backslash s))/np),\ (np/n),\ n \Rightarrow s}(/L)$$

Because different choices of the lexical item `with` both give valid derivations, the sentence is *lexically ambiguous*. Due to *spurious ambiguity* there are also other derivations with the same types, which are essentially the same proof, for example:

$$\dfrac{\dfrac{n \Rightarrow n \quad np \Rightarrow np}{(np/n),\ n \Rightarrow np}(/L) \quad \dfrac{n \Rightarrow n \quad \dfrac{np \Rightarrow np \quad \dfrac{np \Rightarrow np \quad \dfrac{s \Rightarrow s}{np,\ (np\backslash s) \Rightarrow s}(\backslash L)}{(np\backslash s) \Rightarrow (np\backslash s)}(\backslash R)}{((np\backslash s)/np),\ np \Rightarrow (np\backslash s)}(/L)}{((np\backslash s)/np),\ (np/n),\ n \Rightarrow (np\backslash s)}(/L) \quad \dfrac{np \Rightarrow np \quad s \Rightarrow s}{np,\ (np\backslash s) \Rightarrow s}(\backslash L)}{np,\ ((np\backslash s)/np),\ (np/n),\ n,\ ((np\backslash s)\backslash(np\backslash s)) \Rightarrow s}(\backslash L)}{np,\ ((np\backslash s)/np),\ (np/n),\ n,\ (((np\backslash s)\backslash(np\backslash s))/np),\ (np/n),\ n \Rightarrow s}(/L)$$

Now to illustrate an example with *structural ambiguity*, let us look at the following lexicon:

```
every :: (s/(np\s))/n.
man :: n.
loves :: (np\s)/np.
a :: ((s/np)\s)/n.
woman :: n.
```

Using this lexicon the sentences *Every man loves a woman* can be derived as follows:

$$\dfrac{\dfrac{n \Rightarrow n \qquad \dfrac{\dfrac{\dfrac{\dfrac{np \Rightarrow np \quad s \Rightarrow s}{np,\ np\backslash s \Rightarrow s}\,(\backslash L)}{np\backslash s \Rightarrow np\backslash s}\,(\backslash R) \qquad s \Rightarrow s}{np \Rightarrow np \qquad s/(np\backslash s), np\backslash s \Rightarrow s}\,(/L)}{s/(np\backslash s),\ (np\backslash s)/np, np \Rightarrow s}\,(/R)}{s/(np\backslash s), (np\backslash s)/np \Rightarrow s/np}\,(/R) \qquad s \Rightarrow s}{n \Rightarrow n \qquad s/(np\backslash s), (np\backslash s)/np, (s/np)\backslash s \Rightarrow s}\,(\backslash L)}{(s/(np\backslash s))/n, n, (np\backslash s)/np, (s/np)\backslash s \Rightarrow s}\,(/L)}{(s/(np\backslash s))/n, n, (np\backslash s)/np,\ ((s/np)\backslash s)/n, n \Rightarrow s}\,(/L)$$

The other derivation, which represents another meaning of the sentence, is:

$$\dfrac{\dfrac{n \Rightarrow n \qquad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{np \Rightarrow np \quad s \Rightarrow s}{np,\ np\backslash s \Rightarrow s}\,(\backslash L)}{np \Rightarrow np \qquad np,\ np\backslash s \Rightarrow s}\,(/L)}{np,\ (np\backslash s)/np, np \Rightarrow s}\,(/R)}{np, (np\backslash s)/np \Rightarrow s/np}\,(/R) \qquad s \Rightarrow s}{np, (np\backslash s)/np,\ (s/np)\backslash s \Rightarrow s}\,(\backslash L)}{(np\backslash s)/np, (s/np)\backslash s \Rightarrow np\backslash s}\,(\backslash R) \qquad s \Rightarrow s}{s/(np\backslash s), (np\backslash s)/np, (s/np)\backslash s \Rightarrow s}\,(/L)}{n \Rightarrow n \qquad (s/(np\backslash s))/n, n, (np\backslash s)/np, (s/np)\backslash s \Rightarrow s}\,(/L)}{(s/(np\backslash s))/n, n, (np\backslash s)/np,\ ((s/np)\backslash s)/n, n \Rightarrow s}\,(/L)$$

2.2 Proof nets

Proof nets are graph structures that correspond to type-logical proofs like the proofs in categorial grammars. While in sequent calculus there are more derivations for essentially the same proof, proof nets are a unique representation of a proof. To form a proof net, first all logical types have to be unfolded to small tree structures. The different unfolding functions by *(Roorda, 1992) [13]* for the Lambek calculus are shown in figure 2, each one has its conclusions at the bottom and its premisses at the top.

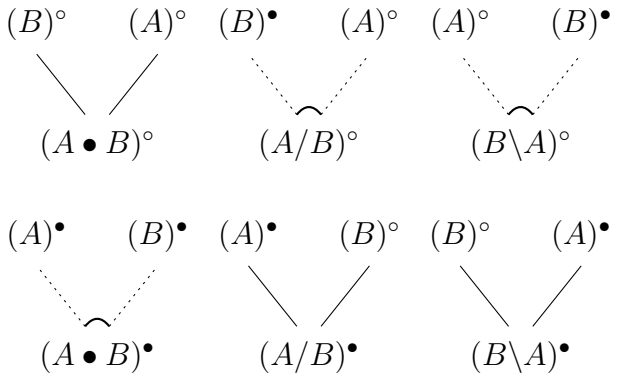
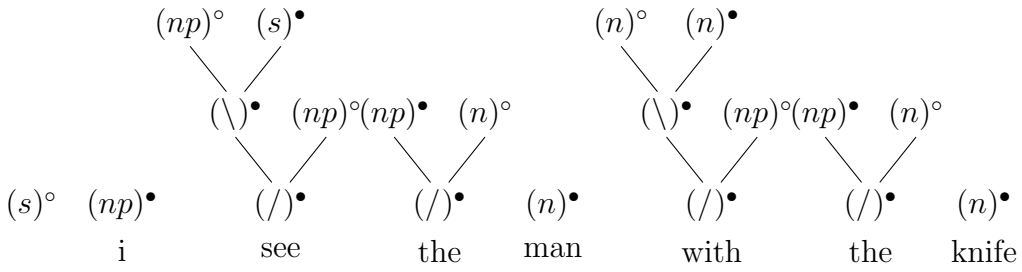


Figure 2: Functions for unfolding formulas to proof structures

The formulas are input ($X^\bullet$) or output (X°), and the connections can be par (dotted) or tensor (solid). To unfold all premisses of a proof, take the main connective of the logical type, make this an input connective, and unfold the rest of the type according to the rules. All unfolded formulas should be placed next to each other in the order of the input. Finally, one takes the conclusion of the proof and unfolds it making the main connective an output formula. It does not matter if this is placed to the left or to the right of the other formulas.

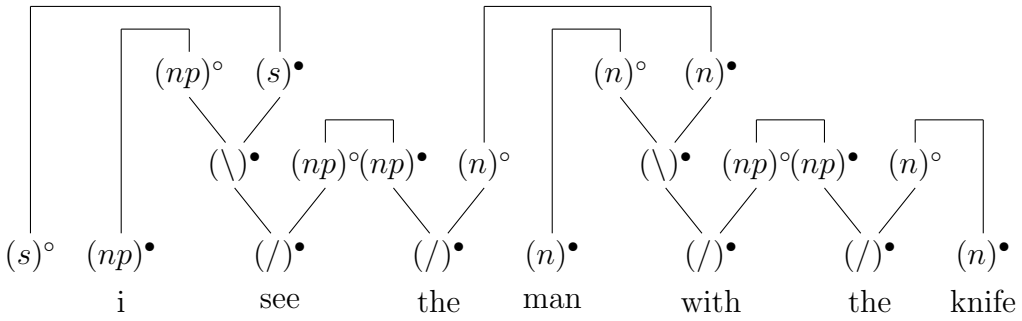
For example, the unfolding of the formulas of *I see the man with the knife* looks like this:



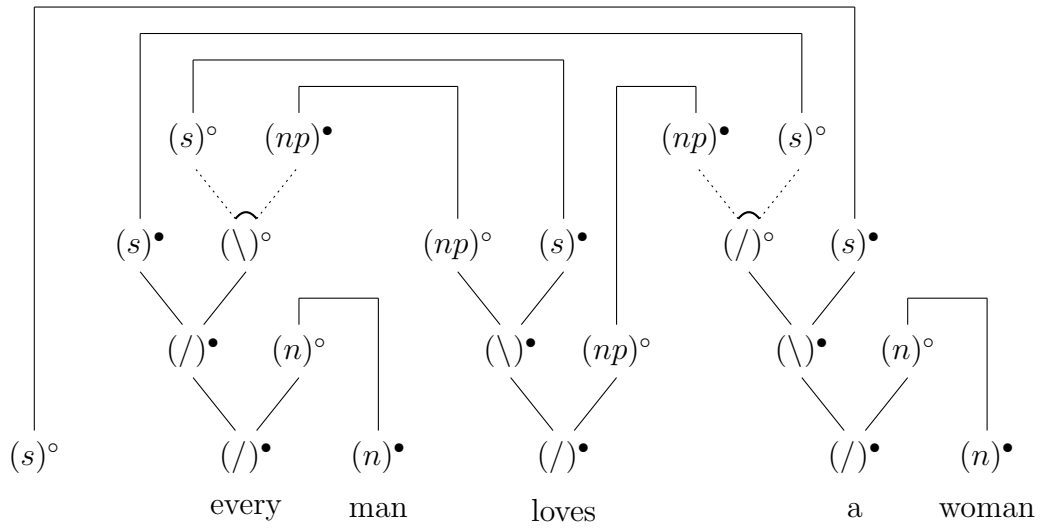
To form the proof structures into a proofnet, every input leaf has to be connected to an output leaf of the same type. The final proof net should have the following properties:

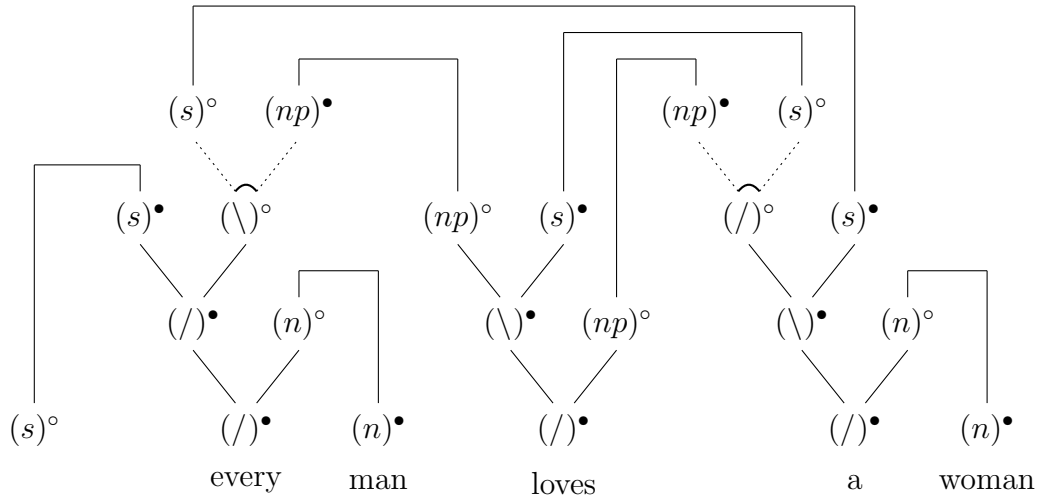
- For each par-link, choose one of the 2 possible connections. For each possible combination of choices for the par-links, the following properties should hold:
  - The resulting graph should be *a-cyclic*.
  - The resulting graph should be *connected*.
- In the case of the Lambek calculus, the resulting proof structure should be *planar*, although it is known that for languages like Dutch this is not always possible. To be able to work with languages like Dutch, *(Moot and Puite, 2002) [7]* have created a more flexible system.

The resulting proof net of the example *I see the man with the knife* looks like this:



While in sequent calculus there are more derivations for this sentence with these words, this is not the case for proofnets. This is the only valid proofnet for these types, so there is no spurious ambiguity anymore. Now let us look again at the example sentence *Every man loves a woman*. This sentence is structural ambiguous and therefore there are 2 different proofnets that correspond to the 2 different meanings of the sentence:





The main reason for using categorial grammars with proof nets is because proof nets solve the problem of spurious ambiguity. With proof nets, each unique derivation has exactly 1 representation as a proofnet. Using this I now only have to solve 2 more types of ambiguity: lexical ambiguity and structural ambiguity.

2.3 Statistical methods

Statistical methods have been used many times before in parsing. The idea that these methods have in common, is that one uses a big dataset of existing sentences and their preferred derivations, called a *treebank*, to gather

statistical information about the different structures that appear in different contexts. When this statistical information is available, it is used to find derivations of other sentences, or for example to pick the most probable derivation among other derivations.

2.3.1 Probabilistic Context Free Grammars

An example of this are the so called *Probabilistic Context Free Grammars* (PCFG's), a technique first used by *(Booth and Thompson, 1973) [2]. (Manning and Shütze, 1999) [6]* define a PCFG as follows:

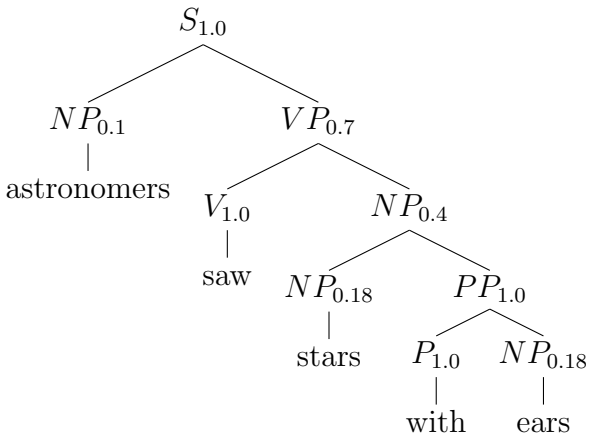
- A set of terminals, $\{w^k\}, k = 1, ..., V$
- A set of nonterminals, $\{N^i\}, i = 1, ..., n$
- A designated start symbol, N^1
- A set of rules, $\{N^i \rightarrow \zeta^j\}$, where ζ^j is a sequence of terminals and nonterminals
- A corresponding set of probabilities on rules such that:
  $\forall i \Sigma_j P(N^i \rightarrow \zeta^j) = 1$

Let us look at an example grammar:

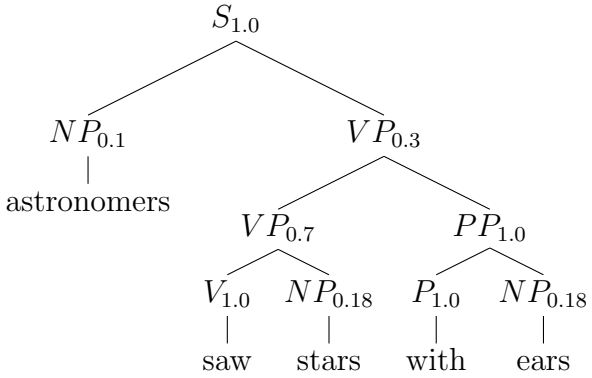
S	→	NP VP	1.0	NP	→	NP PP	0.4
PP	→	P NP	1.0	NP	→	astronomers	0.1
VP	→	V NP	0.7	NP	→	ears	0.18
VP	→	VP PP	0.3	NP	→	saw	0.04
P	→	with	1.0	NP	→	stars	0.18
V	→	saw	1.0	NP	→	telescopes	0.1

The start symbol of this grammar is S, and this grammar satisfies all conditions. Now parsing a sentence with this grammar starts with the start symbol S, followed by a number of rewriting operations to rewrite it to the given input sentence. A rewrite operation is defined as replacing the left hand side of a rule (a nonterminal) by it's right hand side (both terminals and nonterminals). The derivation of a sentence can also be displayed in a tree format, where the left hand side of the rule is the top, and the right hand side are it's children. The root of the tree will be contain start symbol and each leaf of the tree will contain a nonterminal.

The probability of a derivation is the product of the probabilities of all rules that were used. For example, the following tree corresponds to the sentence *astronomers saw stars with ears*:



The probability of this sentence is 1.0 * 0.1 * 0.7 * 1.0 * 0.4 * 0.18 * 1.0 * 1.0 * 0.18 = *0.0009072*. Because of structural ambiguity, there is another possible derivation:



Because this derivation has a probability of *0.0006804*, it is less probable than the first derivation. Therefore the first derivation will be preferred.

2.3.2 Statistical Parsing with CCG's

Another approach using statistical methods in parsing is that of *(Julia Hockenmaier, 2003) [4]*. In her thesis she proposes a number of ways to use probabalistic models in *Combinatory Categorial Grammars* (CCG's). Using the *Penn Treebank* she has generated a probabilistic CCG lexicon, called *CCGBank [3]*, and showed that the accuracy of the derivations using the probabilistic rules is very high.

In CCG's, like in CG's, each word gets one or more categories. Categories are defined as follows:

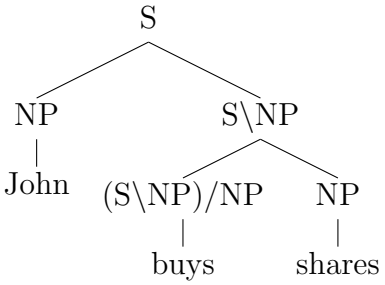
- **Atomic categories:** The grammar for each language defines a finite set of categories, like `S, NP, N, Det, ...`

- **Complex categories:** if `X` and `Y` are categories, `X/Y` and `X\Y` are categories with argument `Y` and result `X`

Note that in CG's argument `Y` on the left with result `X` is written as `Y\X`, while in CCG's this is written as `X\Y`. Assume we have the following lexicon:

```
John :- NP
buys :- (S\NP)/NP
shares :- NP
```

Although derivations are usually written in natural deduction style, it is also possible to represent them as a tree. This is used for the probability rules that are added. The derivation of *John buys shares* represented as a tree looks like:



Now there are 4 different types of information in the tree:

- Leaf nodes, or the lexical items (words), called *leaf*
- Unary trees, called *unary*, for lifted types like:

$$\begin{array}{c} S\backslash(S/NP) \\ | \\ NP \end{array}$$

- Binary trees with head left, called *left*
- Binary trees with head right, called *right*

Using these different tree structures, the probability of the whole derivation is defined as the product of the relevant probabilities of the following 4:

- **Expansion probability:** $P(exp|C)$, the probability of exp given category C, where $exp \in \{leaf, unary, left, right\}$
- **Head probability:** $P(H|C, exp)$, the probability of category H of the head daughter, given category C and exp, where $exp \neq leaf$

- **Non-head probability:** $P(D|C, exp, H)$, the probability of category D of sister daughter, given category C, exp and head category H, where $exp \in \{left, right\}$
- **Lexical probability:** $P(w|C, exp = leaf)$, the probability for word w given category C

Although these probabilities can be useful in my thesis, I will not be able to use them all. Because I have a lexicon too, the lexical probability can be used as it is used in CCG's. This can be the start in solving lexical ambiguity. Because in CG's I do not have grammar rules but only lexical items, I will not be able to use the other 3 probabilities directly. Instead, I will need to find probabilities that can be used in proofnets.

(Moot, 2004) [9] uses graph algorithms not only to filter out links that will never result in a valid proofnet, but also to find the minimum-weighted solution for a proofnet. Here he uses the distance between 2 atomic formulas as weight function, but this can of course be adapted to use a more advanced weight function. Using the ideas from the probabilities that were used for the CCG project, it might be for example possible to assign probabilities to the lexical types indicating what the probability is for having a link to its left or its right respectivally.

3 Extracting a lexicon from a treebank

To approach this problem, I will need a *treebank*, and a lexicon generated from that treebank. *(Moortgat and Moot, 2001) [11]* used the *Corpus Gesproken Nederlands (CGN)* to generate a type-logical lexicon. Although this seems to be pretty useful, I decided not to use the CGN in my thesis. Because the sentences in the corpus are spoken language, there are a lot of weird constructions and sentences that are grammatically incorrect. I will however use the scripts that *(Moortgat and Moot, 2001) [11]* wrote to generate the type-logical lexicon.

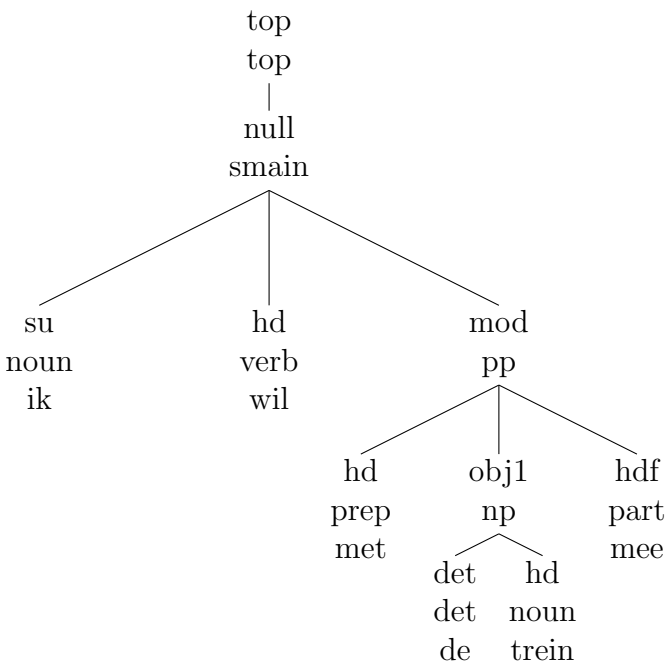
Because Dutch is my native language and an interesting language for linguistics, I decided to use another Dutch corpus: *The Alpino Corpus [1].* In the Alpino corpus, there are a few different smaller treebanks that can be used, all fully annotated. Because the base format is XML, this treebank can be easily translated to the notation that can be used with the scripts that generate the type-logical lexicon.

3.1 The corpus

The Alpino Corpus XML format contains all information needed to create tree representations of the derivations. For each word or leaf it has the following information:

- The word at this leaf
- The base of this word, i.e. for verbs the infinitive
- The position in the original sentence
- The POS tag of this leaf
- The dependency label of this leaf

For nodes in the tree this is almost the same, only the word and base word are missing and instead of the position we know the spanning (i.e. word 2 to 4) of this node. Using a simple script, I have generated tree stuctures from the XML format. A tree can for example look like this:



3.2 The script

The script is given the tree structure, which represents the structure of the input sentence. Using that structure, it will generate logical types for all words in the sentence. The pseudocode of the script looks like this:

```
# returns: main type of this tree, if input_type
# is set this will be input_type
generate_types(input_tree, optional input_type)
   if input_tree.isLeaf() then
      if input_type != null then
         this_type <- input_type
      else
         this_type <- input_tree.type()
      print input_tree.word() + " :: " + this_type + "."
      return this_type
   else
      children <- input_tree.getChildren()
      context <- input_tree.getContext()
      (head, left, right) <- select_head(children, context)
      if input_type != null then
         main_type <- input_type
      else
```

```
            main_type <- input_tree.type()
        head_type <- main_type
        foreach left_part in left
            left_type <- generate_types(left_part)
            head_type <- "(" + left_type + "\" + head_type + ")"
        foreach right_part in right
            right_type <- generate_types(right_part)
            head_type <- "(" + head_type + "/" + right_type + ")"
        generate_types(head, head_type)
        return main_type

# returns:
# head: The type that was selected as head
# left: The trees to the left of the head
# right: The trees to the right of the head
select_head(children, context)
    left <- []
    for i = 0 to children.length
        if children[i] is head in context then
            return (children[i],
                    children.get(0 to i - 1).reverse(),
                    children.get(i + 1 to children.length))
    error("No head found.")
```

3.3 Parameters

When generating the lexicon, a lot of parameters can be set to influence the generated lexicon. The first one is the number of atomic types. In this thesis I have chosen to use 40 different atomic types. This is a lot and will result in a bigger lexicon , but will also result in more specific types for the given sentences. In other words, having more atomic types will increase lexical ambiguity, but will decrease structural ambiguity.

Another set of parameters is used for deciding which words will be used as the head of a certain subtree and which words will be the arguments. While this does not influence the number of generated types, it will influence the resulting types itself and therefore the resulting derivations. I have chosen to use the tagging information from the treebank and use the types tagged as `hd` as the head of a structure. The only exception is an np structure with a determiner, here the determiner will be chosen as the head, to genererate types like `np/noun`, which is usually used in categorial grammars.

(Moot, 2003) [8] has done research after improving the lexicon generated from a treebank. He for example replaces the atomic category `ab` by `n\n`, which reduces the number of atomic types and improves the accuracy of the lexicon. Using ideas from this article I would have probably been able to get a smaller lexicon (where the average number of types per word, so the lexical ambiguity, will be smaller) which could have a higher accuracy. Unfortunately my limited timeframe did not allow me to try this.

When running `generate_types` on the example tree, it will generate the following lexicon:

```
ik :: noun.
wil :: ((noun\top)/pp).
met :: ((pp/part)/np).
de :: (np/noun).
trein :: noun.
mee :: part.
```

3.4 Properties of the generated lexicon

Using the script on the *g_suite* treebank in the Alpino corpus, the generated lexicon has the following properties:

Number of sentences:	998
Total number of words:	7068
Number of unique words:	1254
Number of unique types:	310
Average number of types per word:	1.98804

An example of a word with high lexical ambiguity is the determiner "de", which gets 40 different types in this lexicon. Due to tagging errors or wrong parameter choices, a lot of these types are not what one would usually choose.

```
de :: (det\(np/noun)).
de :: (detp\(np/noun)).
de :: ((np/noun)/detp).
de :: ((np/mwu)/noun).
de :: ((np/np)/noun).
de :: ((np/name)/noun).
de :: ((np/noun)/noun).
```

```
de :: (((conj/np)/vg)/noun).
de :: (((np/rel)/noun)/noun).
de :: ((np/rel)/noun).
de :: ((np/conj)/noun).
de :: ((np/pp)/noun).
de :: ((np/rel)/adj).
de :: ((top/cp)/noun).
de :: ((top/rel)/noun).
de :: ((np/cp)/noun).
de :: ((np/whsub)/noun).
de :: ((np/ti)/noun).
de :: (np/mwu).
de :: (np/name).
de :: (np/noun).
de :: (np/adj).
de :: ((((conj/np)/vg)/adj)/adj).
de :: (((np/pp)/noun)/adj).
de :: (((np/pp)/noun)/num).
de :: (((top/rel)/noun)/adj).
de :: ((np/adj)/adj).
de :: ((np/adj)/adv).
de :: ((np/noun)/adj).
de :: ((np/noun)/ap).
de :: ((np/noun)/num).
de :: ((top/adj)/np).
de :: ((top/noun)/adj).
de :: ((top/noun)/ap).
de :: ((top/noun)/ti).
de :: (((((conj/np)/vg)/adj)/adj)/adj).
de :: (((np/adj)/adj)/adj).
de :: ((mwu/noun)/noun).
de :: adj.
de :: det.
de :: fixed.
```

4 Solving structural ambiguity

Now the lexicon has been generated, the next step would be to use this lexicon to generate derivations for sentences, and see if the derivation trees are the same as the original annotated trees. Unfortunately, there is a problem with this approach: in such a big lexicon, there is huge lexical ambiguity. Although the aim of this thesis is to find heuristics (using statistical data from the treebank) to solve this problem, generating derivations takes too long to use that method in real-life situations for now, and without testing I will not be able to evaluate my heuristics.

In order to solve this problem, I will need to be able to do some testing. Instead of using the whole lexicon for testing, I will only use the types for the words as they are generated from the sentence that is being derived. This way I avoid the lexical ambiguity and focus on the structural ambiguity. Although this is not the optimal way of testing, it can be useful in finding heuristics for efficient parsing to solve the structural ambiguity. So in the first step I will set lexical ambiguity aside and I will only focus on solving structural ambiguity.

When I have succeeded in finding the heuristics for the structural ambiguity, I will be able to move on and solve the lexical ambiguity. This is possible because, when I choose the types for the words from the lexicon, I can use the heuristics to evaluate the chosen types very fast. So the second step is to find heuristics for solving the lexical ambiguity and thereby solving the overall ambiguity problem.

4.1 Generating proofnets

Because it is known that a proofnet corresponds to a proof, picking the links in a proofnet is essentially the same as generating the proof. I will try to find heuristics for picking the links, and compare the resulting proofnet to the proofnet generated from the treebank.

The main algorithm for picking the links is as follows:

```
find_linking(atomlist)
   links <- {}
   inp_list <- input_atoms(atomlist)
   out_list <- output_atoms(atomlist)
   if inp_list != out_list then
```

```
   return -1 # no linking possible
sort inp_list by number
# start with atoms with least number of possible links
for each atom in inp_list
   for i = 0 to atom.count
      possible <- find_possible_linkings(atom)
      for j = 0 to possible.length
         possible[j].weight <- weight_function(possible[j], links)
      sort possible by weight
      links <- links.add(possible[0]) # pick lowest weighted link
return links
```

The functions `input_atoms` and `output_atoms` are functions that count the number of input/output atoms of a certain type and return a list of atom types and their counts. This list is something like `{(num, 1), (np, 3), (n, 1)}`. The check if both lists are the same is not needed in this experiment (as we already know the atom counts match), but it's an easy check to exclude a lot of impossible proofstructures.

The algorithm starts with the atoms with the least number of possible links to make sure that the easy links are made first, and that there is more information available about the proofnet that is being constructed. If the sorted list is `{(num, 1), (n, 1), (np, 3)}`, then both `num` and `n` links can be connected because there is no point of choice. Next, the weight function will be used on the possible options for the np links. Because the `num` and `n` links are already connected, statistics about for example the number of crossing links are more useful than when there are no links yet.

The `find_possible_linkings` function is a function that finds all possible linkings for the given atom type. This function uses code from the *Grail 3* theorem prover, which uses ideas from *(Moot, 2007) [10]* to filter out impossible axiom linkings.

The `weight_function` is obviously the most important part of this code and will be explained in the next section.

4.2 Weight function

The weight function combines 3 different kinds of information to find the weight of the links. These are:

- *Crossing links*: When this link is added to the proofnet, it will possibly

cross with the links chosen so far. The *crossing links* statistic is the number of extra crossings that adding the actual link will introduce.

- *Distance*: The distance between the link is calculated as the difference in indexes in the atom list, so 2 atoms next to each other will have distance 1.

- *Neighbour links*: In the dataset there exist a lot of pairwise linkings, that means that 2 input atoms next to each other are linked to 2 output atoms in the opposite order. The *Neighbour links* statistic is 1 if there are no other links that can be a pairwise match, and 0 if there exists such a link in the actual network.

Using the treebank I have searched for the optimal combination of these 3 different kinds of information. The final weight of a link is calculated as:

*Crossing links * 5 + Distance + Neighbour links*

4.3 Performance

The weight function has been chosen using the *g_suite* treebank from the Alpino corpus. When correct types for a sentence are known, so when we ignore the lexical ambiguity, the `find_linking` function finds the correct linking in **98.48 %** of the cases. When the weight function is constant, so links are randomly chosen with equal probability, this is only **51.93 %**.

Because the weight function has been optimalised for the *g_suite* treebank, it will work very well on that treebank. When tested on a different treebank from the Alpino corpus, the *h_suite* treebank, the `find_linking` function finds the correct linkings for **97.03 %** of the cases. Knowing that this treebank is completely new and has never been used in finding the weight function, this is a very good result.

5 Solving lexical ambiguity

Now that the weight function for solving the structural ambiguity has been found, the next step is to find a solution to the lexical ambiguity. Let us assume we have a sentence s with words $w_1, w_2, w_3, ..., w_n$. Now for each word we have different types, so for word i we have $t_{i,1}, t_{i,2}, t_{i,3}, ..., t_{i,m_i}$. We also have the probability that word w_i has type $t_{i,j}$, let us call this $P(w_i, t_{i,j})$.

5.1 Finding the correct types

The first approach could be to generate all combinations of types, and generate all the proofnets using the weight function. Then we could define the overall weight of a proofnet as the sum of the weights of all links, times the product of the probabilities of the different types. So if the types for the words are chosen as $w_1 : t_{1,x_1}, w_2 : t_{2,x_2}, w_3 : t_{3,x_3}, ..., w_n : t_{n,x_n}$, the overall weight of the proofnet would be:

$(link\_1\_weight + link\_2\_weight + link\_3\_weight + ... + link\_y\_weight) * P(w_1, t_{1,x_1}) * P(w_2, t_{2,x_2}) * P(w_3, t_{3,x_3}) * ... * P(w_n, t_{n,x_n})$

Now for the sentence s, generate all possible proofnets, and choose the one with the highest overal probability as the most preferred one. This could theoretically be a good approach, but unfortunately this will not work. The number of different combinations is $m_1 * m_2 * m_3 * ... * m_n$, or worst case $O(n^m)$, where n is the number of words in a sentence and m is the number of different types per word. So the algorithm is *exponential* and could take years to run on longer sentences.

Instead of generating all different combinations of types, we will have to take only the most probable one. To do this, we will sort the types for word w_i by probability, in such a way that the following equation holds:

$$P(w_i, t_{i,x_1}) >= (w_i, t_{i,x_2}) >= (w_i, t_{i,x_3}) >= ... >= (w_i, t_{i,x_{m_i}})$$

When generating the proofnet, the algorithm will first try to use the types that have the highest lexical probability. It will run as long as no valid proofnets have been found, so the types with lower probability will only be used after backtracking. The first valid proofnet that has been found using this method will be the resulting proofnet.

5.2 Testing on the treebank

With this method, I am able to run tests on the treebank. For each sentence I know the correct proofnet (extracted from the treebank), and I compare the proofnet found with the described algorithm to the correct proofnet. Although this seem to work pretty well on most sentences, there is still a problem with this approach: In worst case, the only valid proofnet is the proofnet with all types with the lowest probability. Because that combination will be found last, the worst case runtime of the new algorithm still is $O(n^m)$.

To avoid problems with a long runtime and to be able to test the whole lexicon, I decided to set a maximum execution time of 20 seconds per sentence on the algorithm. This means that, for each sentence, if after 20 seconds the algorithm did not find a valid proofnet, it is counted as being wrong. This is a trade-off because the algorithm could for example find the correct solution after 21 seconds, but as I have not got any other good solutions to the long runtime, I decided to use this method. If I use longer runtimes or find a more efficient algorithm, the accuracy of the algorithm can only improve.

5.3 Performance

When run on the `g_suite` treebank, the algorithm finds the correct proofnet in **83.57 %** of the cases. It exceeds the maximum runtime of 20 seconds in 15.92 % of the cases, and in only 0.51 % of the cases it returns an incorrect proofnet. As said, this score could possibly be improved by using a longer runtime or finding more efficient algorithms, but unfortunately I am not able to do this within the current timeframe.

6 Conclusions

6.1 My work

In this thesis I have looked into possible solutions to different types of ambiguity in parsing. The *spurious ambiguity* has been solved by using proofnets as the linguistic framework. To solve the other types of ambiguity I have first looked into the generation of a type-logical lexicon from a treebank. Using this lexicon I searched for heuristics for linking the atoms in proofnets, and created a weight function for the linkings thereby solving the *structural ambiguity*. When tested on an automatically generated lexicon, and without taking lexical ambiguity in account, the heuristics found the correct linkings in 97.03 % of the cases. Finally, I have looked into a way to solve the *lexical ambiguity* by taking only the most probable types for the words in a sentence. Using this and the weight function, the correct proofnet was found in **83.57 %** of the cases.

6.2 Future work

Although the results of my work look promising so far, there still is a lot of research that can be done. Unfortunately my timeframe was too limited to do more work, so I will outline a few ideas that I had below.

- *Using a better lexicon*
  Although the automated generation of a lexicon was good for testing purposes, the lexicon was not 100% correct. Unfortunately the parameter configuration for modifiers would introduce a lot of crossing links, and it did not always lead to a valid proofnet. Also I did not use the ideas from *(Moot, 2003) [8]* to create a smaller lexicon with less lexical ambiguity.

- *Link direction and allowing crossing links*
  Instead of just linking an input and output atom to each other, one could also define the atoms to have a certain direction. For example for the Dutch word "*er*" it is known that it will always be linked to atoms on its right side, possibly crossing other links.

- *Using POS tag information*
  While the weight function as it is now, uses information about the geometrical structure of the proofnet, it does not actually use information about the POS tags that it is processing. For the Dutch language, we know for example that `det` (determiners) should always be linked with a word on the right side of it.

References

[1] Beek, Leonoor van der, Gosse Bouma, Robert Malouf and Gertjan van Noord (2002). The Alpino Dependency Treebank. *In Computational Linguistics in the Netherlands (CLIN) 2001, Twente University, 2002.*

[2] Booth, Taylor L. and Richard A. Thompson (1973). Applying probability measures to abstract languages. *IEEE Transactions on Computers, C-22(5):442-450.*

[3] Hockenmaier, Julia and Mark Steedman (2002). Acquiring Compact Lexicalized Grammars from a Cleaner Treebank. *Proceedings of Third International Conference on Language Resources and Evaluation, Las Palmas*

[4] Hockenmaier, Julia (2003). Data and models for statistical parsing with Combinatory Categorial Grammar. *PhD Thesis, School of Informatics, University of Edinburgh.*

[5] Lambek, Joachim (1958). The Mathematics of Sentence Structure. *The American Mathematical Monthly, Vol. 65, No. 3 (Mar., 1958), pp. 154-170*

[6] Manning, Chris and Hinrich Schütze (1999). Foundations of Statistical Natural Language Processing. *MIT Press. Cambridge, MA: May 1999*

[7] Moot, Richard and Quintijn Puite (2002). Proof Nets for the Multimodal Lambek Calculus. *Studia Logica 71(3):415-442*

[8] Moot, Richard (2003). Parsing corpus-induced type-logical grammars. *in R. Bernardi & M. Moortgat, eds, 'Proceedings of the CoLogNet/ElsNet Workshop on Linguistic Corpora and Logic Based Grammar Formalisms', pp. 70-85*

[9] Moot, Richard (2004). Graph algorithms for improving type-logical proof searc. *Proceedings of Categorial Grammars 2004, pp. 13-28, Submitted to Elsevier Science.*

[10] Moot, Richard (2007). Filtering axiom links for proof nets. *in Laura Kallmeyer, Paola Monachesi, Gerald Penn and Giorgio Satta, eds, 'Proceedings of Formal Grammar 2007', to appear with CSLI.*

[11] Moortgat, Michael and Richard Moot (2001). CGN to Grail: Extracting a type-logical lexicon from the CGN annotation. *in W. Daelemans, ed.,'Proceedings of CLIN 2000', pp. 126*

[12] Moortgat, Michael (2002). Categorial grammar and formal semantics. *Article #231, Encyclopedia of Cognitive Science, Nature Publishing Group, Macmillan Publishers Ltd.*

[13] Roorda, Dirk (1992). Proof Nets for Lambek Calculus. *Journal of Logic and Computation, 1992 - Oxford Univ Press*