University of Warsaw
Faculty of Mathematics, Informatics and Mechanics

Paweł Marczewski
Student number: 262952

Building a lexicon for a categorial grammar of the Polish language

Master's Thesis
in INFORMATICS

Supervisor
dr Wojciech Jaworski
Institute of Informatics

May 2012

Oświadczenie kierującego pracą

Potwierdzam, że niniejsza praca została przygotowana pod moim kierunkiem i kwalifikuje się do przedstawienia jej w postępowaniu o nadanie tytułu zawodowego.

Data Podpis kierującego pracą

Oświadczenie autora (autorów) pracy

Świadom odpowiedzialności prawnej oświadczam, że niniejsza praca dyplomowa została napisana przeze mnie samodzielnie i nie zawiera treści uzyskanych w sposób niezgodny z obowiązującymi przepisami.

Oświadczam również, że przedstawiona praca nie była wcześniej przedmiotem procedur związanych z uzyskaniem tytułu zawodowego w wyższej uczelni.

Oświadczam ponadto, że niniejsza wersja pracy jest identyczna z załączoną wersją elektroniczną.

Data Podpis autora (autorów) pracy

Streszczenie

Celem pracy jest konwersja *Składnicy*, polskiego banku drzew rozbioru składniowego, do formatu wywodów rozszerzonej wersji CCG (Combinatory Categorial Grammar). CCG jest gramatyką leksykalną, która umożliwia pozyskanie informacji składniowych ze zdania za pomocą małej liczby uniwersalnych reguł oraz leksykonu kategorii dla poszczególnych słów. Praca wprowadza rozszerzenia CCG mające na celu przystosowanie formalizmu do opisu polskiej składni (multizbiory argumentów i kategorie z dozwoloną kompozycją) oraz reguły składniowe dla nich. Następnie przedstawiona jest odpowiednia wersja rachunku lambda oraz język reprezentacji treści, które razem tworzą warstwę semantyczną formalizmu. Na koniec opisany jest proces konwersji *Składnicy*. Powstały w efekcie leksykon jest głównym wynikiem pracy. Przedstawiona jest jego statystyczna oraz opisowa ewaluacja.

Słowa kluczowe

CCG, gramatyka kategorialna, gramatyka języka polskiego, bank drzew

Dziedzina pracy (kody wg programu Socrates-Erasmus)

11.3 Informatyka

Klasyfikacja tematyczna

I. Metodologie obliczeniowe
I.2. Sztuczna inteligencja
I.2.6. Przetwarzanie języka naturalnego

Tytuł pracy w języku polskim

Budowa leksykonu dla gramatyki kategorialnej języka polskiego

Abstract

The goal of this dissertation is conversion of *Składnica*, the Polish bank of syntax trees, to a collection of derivations in an extended version of CCG (Combinatory Categorial Grammar). CCG is a lexicalized grammar that allows for extracting syntactic information from a sentence using a small number of universal rules and a lexicon of word categories. The dissertation introduces extensions of CCG motivated by Polish syntax (argument multisets and categories with permitted composition) and their syntactic rules. We then introduce an appropriate version of lambda calculus and a content representation language – they are used together as a semantic layer for the formalism. Finally, we describe the process of converting *Składnica* to our grammar. The resulting lexicon is the main outcome of this work. We present its quantitative and descriptive evaluation.

Keywords

CCG, combinatory categorial grammar, Polish grammar, treebank

Subject area (codes from Socrates-Erasmus)

11.3 Informatics, Computer Science

Topic classification

I. Computing Methodologies
I.2. Artificial Intelligence
I.2.7. Natural Language Processing

Contents

Chapter 1

Introduction

In this dissertation we attempt to adapt the Combinatory Categorial Grammar (CCG) to the specifics of Polish language, and create a bank of sentence derivations in the resulting formalism.

CCG is a simple but expressive grammar, proposed by Mark Steedman as an extension of Categorial Grammar first developed by Kazimierz Ajdukiewicz in 1935 (see [Steedman11]). It relies on combinatory logic and is able to describe a wide range of syntactic phenomena in the English language. It is a lexicalised formalism – the linguistic information is stored not in the grammar rules, but in the categories of individual words. The categories are then combined using a few general rules to produce the sentence structure.

In 2003, Julia Hockenmaier built a bank of CCG derivations for English sentences, using Penn Treebank (a manually constructed 1-million-word corpus of syntax analyses) as source data [Hockenmaier03]. We try to construct a similar bank of derivations for the Polish language, based on Polish bank of constituency trees called *Składnica* [Sklad].

Because Polish is a free-word-order language, the grammar must be adapted before it can model at least simple sentence structure adequately. As a solution to a similar problem in the Turkish language, a grammar called Multiset-CCG has been proposed by Beryl Hoffman [Hoffman95]. We use his idea as a starting point for developing our own formalism.

1.1. Results of this work

The dissertation provides the following:

- A **grammar** called PL-CCG, based on Combinatory Categorial Grammar and its Multiset-CCG extension by Beryl Hoffman [Hoffman95]. The syntactic part of the formalism is basically Multiset-CCG extended with composable categories and feature vectors. PL-CCG is intended as an adequate target for conversion of syntactic trees from the existing Polish treebank [Sklad].

- A **semantic representation** layer of PL-CCG, which is a "multi-argument" version of lambda calculus, combined with term assignment rules governing its use in the derivations.
- A **content representation** layer built on the semantics, used to represent the parsing results. We use the lambda-terms to build tree-like structures from which the syntactic dependencies can be extracted. The process of extraction is non-trivial because of the presence of adjuncts in derivations.
- A **conversion algorithm** that produces PL-CCG derivations from the constituency trees used by the existing Polish treebank. The general structure of the algorithm is inspired by Julia Hockenmaier's work for English [Hockenmaier03], but the method of conversion is adapted both to the new formalism and to the specifics of source data.
- Finally, a **treebank** produced by the conversion, of which the most important part is the **lexicon** – a dictionary of lexical categories for words encountered. We evaluate the quality of the lexicon both quantitatively and descriptively.

While we attempt to adapt the overall formalism to the specifics of Polish language (especially the syntactic part, as explained in chapter 4), the main motivation for that comes from the source data, not linguistics. We do not aim to cover a wide range of linguistic phenomena.

1.2. Outline of the dissertation

Chapter 2 describes the Combinatory Categorial Grammar. We present the syntactic rules of the formalism, using a sequent system called Lambek calculus as a starting point.

Chapter 3 introduces lambda calculus, and shows how it is used as a semantic layer of CCG.

Chapter 4 provides the linguistic motivation for a more general version of the formalism suitable for the Polish language, which we call PL-CCG.

Chapter 5 develops the semantics of PL-CCG. We adapt the simply-typed lambda calculus defined in chapter 2 to our version of the formalism.

Chapter 6 defines *content trees*, which are used to represent syntactic dependencies in the sentence. We show how to use the PL-CCG semantic layer to build the trees, and how the dependencies are extracted from them.

Chapter 7 describes the PL-CCG treebank, a bank of derivations that is the subject of this work. We begin with a summary of corresponding treebank for the English language (as described in [Hockenmaier03]) and a description of Polish constituency trees bank that is our source data. We then present the conversion algorithm used to make the PL-CCG treebank.

Chapter 8 attempts to evaluate the resulting treebank, or rather, its set of lexical categories. First, a quantitative evaluation is performed: we define an appropriate measure and test the lexicon using traditional machine learning methodology. Afterwards, we examine the individual categories produced by our algorithm to determine where the conversion is adequate and where it still needs improvement.

Chapter 9 briefly sums up the results of this work, and suggests some possibilities of follow-up.

Chapter 2

Combinatory Categorial Grammar

Combinatory Categorial Grammar (CCG, [Steedman11]) is a lexicalized linguistic formalism in which words are characterized by their *categories*, resembling function types. The types are then combined, creating a partial syntactic derivation of sentence fragments and ultimately the whole sentence. One of the useful properties of CCG is the ability to combine sentence fragments not traditionally regarded as meaningful constituents, which allows describing linguistic phenomena such as long-distance dependencies and coordination.

Along with the derivation, a semantic structure is usually constructed. Usually lambda-terms are attached to the constituents, and combined using rules taken from combinator calculus, corresponding directly to the combinatory rules of the CCG grammar.

We introduce the CCG grammar using a version of Lambek calculus [Lambek58]. Lambek's formalism is a sequent calculus from which we can derive the necessary combinatory rules, giving them solid logical grounding.

This chapter covers the basic features of CCG as a starting point to develop our own formalism. For a more thorough description of CCG itself, see [Steedman11].

2.1. Categories

We define a set of *categories* as follows:

- a finite set of *atomic categories*, such as S (sentence), NP (noun phrase), PP (prepositional phrase)
- *functor (slash) categories*: X/Y, $X\backslash Y$ where X, Y are categories.

Functor categories X/Y, $X\backslash Y$ describe a function type with argument Y and result X. The slash denotes the direction: an argument with a slash (/) is expected on the right side of the functor, a backslash (\) means the argument is expected on the left side.

A simple lexicon of categories (example from [Steedman11]) could be:

- *Marcel* : NP
- *proved* : (S\NP)/NP
- *completeness* : NP

In the example, we consider a noun phrase (such as *Marcel*) to be an atomic type. A transitive verb such as *proved* is a function that expects a noun phrase to its right (the object), then another one to its left (the subject), and finally returns a sentence.

2.2. Lambek calculus

Before we present the actual combinatory rules of CCG, we need to introduce the more general Lambek calculus, on which we will base them. The formalism presented here is a slightly simplified version of one described in [Lambek58] (we omit the concatenation operator).

Lambek calculus is a proof system operating on *sequents* of the form

$$X_1, X_2, \ldots, X_n \vdash Y$$

where $X_1 \ldots X_n$ and Y are categories. The intended reading is: a sequence of constituents of types X_1, X_2, ..., X_n can be used as a constituent of type Y. In our example, to parse the sentence *Marcel proved completeness*, we will need to prove

$$\mathsf{NP},\ (\mathsf{S}\backslash\mathsf{NP})/\mathsf{NP},\ \mathsf{NP} \vdash \mathsf{S}$$

Proofs are constructed using the following rules. In each derivation rule, X and Y are categories, while Γ, Δ, Δ' are sequences of categories.

The Axiom rule says that a type can be reduced to itself.

$$(Axiom)\ \frac{}{X \vdash X}$$

The Cut rule allows us to use one sequent in another.

$$(Cut)\ \frac{\Delta, X, \Delta' \vdash Z \qquad \Gamma \vdash X}{\Delta, \Gamma, \Delta' \vdash Z}$$

Finally, we have left- and right-side rules for both connectives.

$$(\vdash /)\ \frac{\Gamma, Y \vdash X}{\Gamma \vdash X/Y} \qquad (/ \vdash)\ \frac{\Delta, X, \Delta \vdash Z \qquad \Gamma \vdash Y}{\Delta, X/Y, \Gamma, \Delta' \vdash Z}$$

$$(\vdash \backslash)\ \frac{Y, \Gamma \vdash X}{\Gamma \vdash X\backslash Y} \qquad (\backslash \vdash)\ \frac{\Delta, X, \Delta \vdash Z \qquad \Gamma \vdash Y}{\Delta, \Gamma, X\backslash Y, \Delta' \vdash Z}$$

2.3. CCG combinatory rules

In the actual CCG grammar, we don't use the Lambek calculus, but a number of combinatory rules derived from it. Lambek calculus serves here as a foundation for a simpler set of rules.

2.3.1. Application

The basic rule for combining the categories is the application rule (referred to as $>$ or $<$, depending on the direction):

$$(>)\ \ X/Y,\ Y \vdash X \qquad (<)\ \ Y,\ X\backslash Y \vdash X$$

The application rule is a trivial result in the Lambek calculus. We present the derivation for $>$; the one for $<$ is analogous:

$$(\vdash /)\ \frac{\dfrac{}{X \vdash X} \quad \dfrac{}{Y \vdash Y}}{X/Y,\ Y \vdash X}$$

The Cut rule justifies using the application anywhere in the category sequence. In a more traditional CCG notation, an example derivation for the sentence *Marcel proved completeness* could be:

$$\dfrac{\dfrac{\textit{Marcel}}{\mathsf{NP}} \quad \dfrac{\dfrac{\textit{proved}}{(\mathsf{S}\backslash\mathsf{NP})/\mathsf{NP}} \quad \dfrac{\textit{completeness}}{\mathsf{NP}}}{\mathsf{S}\backslash\mathsf{NP}}\,>}{\mathsf{S}}\,<$$

What we have proved is basically the sequent:

$$\mathsf{NP},\ (\mathsf{S}\backslash\mathsf{NP})/\mathsf{NP},\ \mathsf{NP} \vdash \mathsf{S}$$

However, the process can also be viewed as *parsing* the sentence.

2.3.2. Composition

The second rule in the CCG formalism is the composition rule ($> B$ or $< B$):

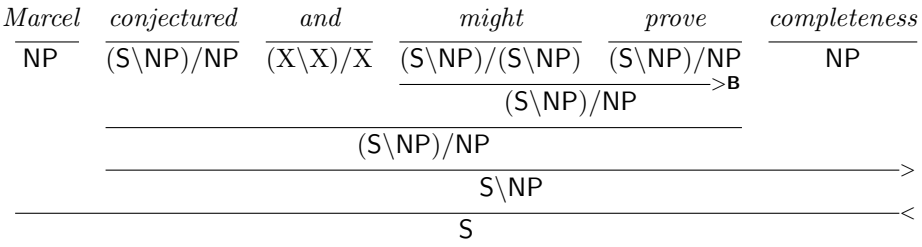
$$(> B)\ \ X/Y,\ Y/Z \vdash X/Z \qquad (< B)\ \ Y\backslash Z,\ X\backslash Y \vdash X\backslash Z$$

The intuition is that of a function composition. This rule is certified by the following proof in Lambek calculus:

$$(/\vdash)\ \frac{(\vdash /)\ \dfrac{(\vdash /)\ \dfrac{\dfrac{}{X \vdash X} \quad \dfrac{}{Y \vdash Y}}{X/Y,\ Y \vdash X} \quad \dfrac{}{Z \vdash Z}}{X/Y,\ Y/Z,\ Z \vdash X}}{X/Y,\ Y/Z \vdash X/Z}$$

A corresponding proof exists for $< B$.

An example (adapted from [Steedman11]) shows how the composition rule can be usede to capture coordination in language. Even though *might* in our grammar is a modifier for verb phrases (S\NP), using the composition rule we have been able to use it before we have a complete verb phrase (i.e. before the verb is applied to its object).



For simplicity, we can assume the type of *and* is $(X\backslash X)/X$, where X is (S\NP)/NP. A more general solution would be to allow polymorphic categories ("$(X\backslash X)/X$ for any X"), or even a special conjunction category with its own derivation rule, as in [Hockenmaier03].

2.3.3. Type-raising

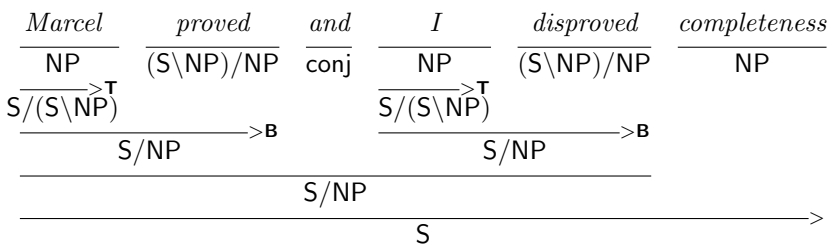
The third of the standard rules is the unary rule of type-raising. Like other rules, it comes in two variations:

$$(> T) \;\; X \vdash Y/(Y\backslash X) \qquad (< T) \;\; X \vdash Y\backslash(Y/X)$$

The Lambek derivation (as usual, shown only for the right variation) is:

$$\begin{array}{l}(\backslash\vdash)\\(\vdash/)\end{array}\dfrac{\dfrac{}{Y \vdash Y} \quad \dfrac{}{X \vdash X}}{\dfrac{X,\; Y\backslash X, \vdash Y}{X \vdash Y/(Y\backslash X)}}$$

The type-raising rules, as explained in [Steedman11], "turn arguments into functions over functions-over-such-arguments." An example of coordination that requires type-raising to be parsed is (again from [Steedman11]):



2.4. Notation: slash associativity

In the subsequent chapters, we assume that slashes are left-associative, i.e. $\mathsf{X/Y\backslash Z}\ldots$ should be understood as $(\mathsf{(X/Y)\backslash Z})\ldots$. The convention mirrors the usual right-associativity of arrows in types and logic formulas ($\alpha \to \beta \to \gamma \equiv \alpha \to (\beta \to \gamma)$) and corresponds to the notion of a multi-argument (curried) function.

Chapter 3

CCG semantics

The derivation is central to parsing sentences in CCG, but the derivation tree is not the main result of parsing. As a result of introducing the B and T rules, a sentence can be parsed in a number of different, but intuively equivalent ways. The phenomenon, called *spurious ambiguity*, is one of the reasons for introducing a *semantic* layer to CCG.

In addition to types, the constituents in a derivation can be supplied with a semantic interpretation. The interpretations are a part of the lexical entries for the single words. In the course of derivation, they are combined according to the derivation rules. We will consider the semantics of the whole sentence to be the final result of parsing, and call derivations *equivalent* if they produce to the same semantics.

The natural choice for such interpretation is lambda calculus. The reason for that is the *Curry-Howard isomorphism* (see [Urzyczyn03]), a relationship between logical formulas and types of lambda-terms. Our formalism can be viewed either as a logical inference system, or equivalently as a *type system* for the underlying calculus.

In this chapter, we very briefly introduce lambda calculus, then extend the Lambek calculus with lambda-terms. Using the modified calculus, we can present the resulting "semantic" version of CCG rules. Finally, we show how the formalism can be used in a derivation.

3.1. Lambda calculus

We define a *lambda-term* as one of the following:

- variable: $x, y, z \ldots$,
- abstraction: $\lambda x.M$, where x is a variable and M is a lambda-term,
- application: $M@N$, where M and N are lambda-terms.

We assume that application is left-associative, that is, $M@N@P \ldots \equiv ((M@N)@P) \ldots$

The lambda-terms are transformed using an operation called β*-reduction*, which allows us to rewrite a pair of adjacent abstraction and application:

$$(\lambda x.M)@N \rightarrow_\beta M[x := N]$$

The notation means that $(\lambda x.M)@N$ can be rewritten as M with all instances of x substituted with N. The operation can be performed on any subterm inside a term, as long as it is of the form $(\lambda x.M)@N$.

Before we can apply β-reduction, we sometimes have to rename the bound variables (that is, variables inside their corresponding abstractions) in N so that the names don't conflict. The process of renaming bound variables is called α-conversion.

A more formal and thorough introduction to lambda calculus can be found in the beginning chapters of [Urzyczyn03].

3.2. The type system

To use the lambda calculus in CCG grammar, we introduce a type system. We attach a type (a CCG category) to each term we use, and to each subterm inside the term. The notation $M : X$ means that the category of term M is X.

The typing rules are as follows:

- all free instances of a variable in a term (that is, all instances of x that are not inside an abstraction $(\lambda x.M)$ must have the same type,
- if $M : X$, and $x : Y$ in M, then $\lambda x.M$ must be either of type X/Y, or $X\backslash Y$,
- conversely, if M is of type X/Y or $X\backslash Y$, then in the term $M@N$ we must have $N : Y$ and $M@N : X$.

An obvious consequence of these rules is that β-reduction preserves types:

Theorem 1. *If $T \to T'$, and $T : X$, then $T' : X$.*

Proof. Because the beta-reduction changes only a specific subterm, it's enough to verify the property for $T = (\lambda x.M)@N$. According to the typing rules, it must be typed as $(\lambda x.(M : X))@(N : Y)$, with all free instances of x in M having type Y. Because we substitute N, which has also type Y, the resulting $T' = M[x := N]$ is still a term of type X. □

In the system of simple types, β-reduction also has the important *strong normalization* property.

Theorem 2 (Strong normalization)**.** *If M is a typed term, then every β-reduction sequence beginning with M terminates at the same unreducible term M'.*

Proof. See the proof in [Urzyczyn03] (chapter 2) for simply typed lambda calculus, which can be easily extended to our category system with directional slashes. □

The unreducible M' is called the *normal form* of a term. The strong normalization property allows us to choose any reduction strategy we want, with the guarantee that we will always arrive at the unique normal form of a given term.

3.3. Lambek calculus with lambda-terms

We extend Lambek calculus with lambda-terms. Each category in the sequent has to be accompanied by a lambda-term.

The Axiom and Cut rules do not change the terms. However, we require that the M we introduce in the Axiom rule is a suitable term for type X.

$$(Axiom)\ \frac{}{M : X \vdash M : X} \qquad (Cut)\ \frac{\Delta, M : X, \Delta' \vdash N : Z \qquad \Gamma \vdash M : X}{\Delta, \Gamma, \Delta' \vdash N : Z}$$

The slash rules allow us to introduce abstraction and application. The typing rules again require of us that the terms will be well-typed, i.e. that $y : Y$ in the term M in rules $\vdash /$ and $\vdash \backslash$.

$$(\vdash /)\ \frac{\Gamma, y : Y \vdash M : X}{\Gamma \vdash \lambda y.M : X/Y} \qquad (/ \vdash)\ \frac{\Delta, M@N : X, \Delta \vdash P : Z \qquad \Gamma \vdash N : Y}{\Delta, M : X/Y, \Gamma, \Delta' \vdash P : Z}$$

$$(\vdash \backslash)\ \frac{y : Y, \Gamma \vdash M : X}{\Gamma \vdash \lambda y.M : X\backslash Y} \qquad (\backslash \vdash)\ \frac{\Delta, M@N : X, \Delta \vdash P : Z \qquad \Gamma \vdash N : Y}{\Delta, \Gamma, M : X\backslash Y, \Delta' \vdash P : Z}$$

We also add the β rule to be able reduce the lambda-terms. By Theorem 1, the type X stays the same.

$$(\beta)\ \frac{}{M : X \vdash M' : X}\ \text{if } M \to_\beta M'$$

3.4. CCG rules with semantics

We can now derive our application rule using lambda-terms.

$$(\vdash /)\ \frac{\dfrac{}{M@N : X \vdash M@N : X} \qquad \dfrac{}{N : Y \vdash N : Y}}{M : X/Y,\ N : Y \vdash M@N : X}$$

A similar derivation can be built for composition:

$$(/ \vdash)\ \frac{(\vdash /)\ \dfrac{\dfrac{}{f@(g@x) : X \vdash f@(g@x) : X} \qquad \dfrac{}{g@x : Y \vdash g@x : Y}}{f : X/Y,\ g@x : Y \vdash f@(g@x) : X} \qquad \dfrac{}{x : Z \vdash x : Z}}{f : X/Y,\ g : Y/Z,\ x : Z \vdash f@(g@x) : X}$$
$$f : X/Y,\ g : Y/Z \vdash \lambda x.f@(g@x) : X/Z$$

and for type-raising:

$$(\backslash \vdash)\ \frac{\dfrac{}{x@a : Y \vdash x@a : Y} \qquad \dfrac{}{a : X \vdash a : X}}{(\vdash /)\ \dfrac{a : X,\ x : Y\backslash X, \vdash x@a : Y}{a : X \vdash \lambda x.x@a : Y/(Y\backslash X)}}$$

Finally, we add the β-reduction rule to our system.

$$(\beta)\ \ t : X \vdash t' : X\ \ \text{if } t \to_\beta t'$$

3.4.1. Example derivations with semantics

Assuming we have some constants in our lambda-calculus to describe the entities and relations, our first example (*Marcel proved completeness*) can now be extended with simple semantic information:

$$\dfrac{\dfrac{\textit{Marcel}}{\mathbf{m} : \mathsf{NP}} \quad \dfrac{\dfrac{\textit{proved}}{\lambda x.\lambda y.\mathbf{p}@x@y : (\mathsf{S}\backslash\mathsf{NP})/\mathsf{NP}} \quad \dfrac{\textit{completeness}}{\mathbf{c} : \mathsf{NP}}}{\lambda y.\mathbf{p}@\mathbf{c}@y : \mathsf{S}\backslash\mathsf{NP}}>}{\mathbf{p}@\mathbf{c}@\mathbf{m} : \mathsf{S}}<$$

Assigning the semantics can be more complicated than choosing the syntactic category. We can show that on the example of *and*, for which we earlier proposed a class of categories $(X\backslash X)/X$ (for any category X).

The obvious choice for a simple semantic representation of *and* is the term

$$\lambda x.\lambda y.\mathbf{a}@x@y$$

(equivalently, just $\mathbf{a}$). This solution seems to work in the simple case of noun phrases:

$$\dfrac{\dfrac{\textit{dogs}}{\mathbf{d} : \mathsf{NP}} \quad \dfrac{\dfrac{\textit{and}}{\lambda x.\lambda y.\mathbf{a}@x@y : (\mathsf{NP}\backslash\mathsf{NP})/\mathsf{NP}} \quad \dfrac{\textit{cats}}{\mathbf{c} : \mathsf{NP}}}{\lambda y.\mathbf{a}@\mathbf{c}@y : \mathsf{NP}\backslash\mathsf{NP}}>}{\mathbf{a}@\mathbf{c}@\mathbf{d} : \mathsf{NP}}<$$

However, when we want to coordinate more complex categories, like verbal phrases ($\mathsf{S}\backslash\mathsf{NP}$) in a sentence like *Marcel runs and jumps*, we often aim to represent it as a conjunction of two separate sentence representations ($\mathbf{a}@(\mathbf{j}@\mathbf{m})@(\mathbf{r}@\mathbf{m})$). In that case, we need to choose another semantic representation for the conjunct:

$$\dfrac{\dfrac{\textit{Marcel}}{\mathbf{m} : \mathsf{NP}} \quad \dfrac{\dfrac{\textit{runs}}{\lambda x.\mathbf{r}@x : \mathsf{S}\backslash\mathsf{NP}} \quad \dfrac{\dfrac{\textit{and}}{\lambda x.\lambda y.\lambda z.\mathbf{a}@(x@z)@(y@z) : (\mathsf{S}\backslash\mathsf{NP})\backslash(\mathsf{S}\backslash\mathsf{NP})/(\mathsf{S}\backslash\mathsf{NP})} \quad \dfrac{\textit{jumps}}{\lambda x.\mathbf{j}@x : \mathsf{S}\backslash\mathsf{NP}}>}{\lambda y.\lambda z.\mathbf{a}@(\mathbf{j}@z)@(y@z) : (\mathsf{S}\backslash\mathsf{NP})\backslash(\mathsf{S}\backslash\mathsf{NP})}<}{\lambda z.\mathbf{a}@(\mathbf{j}@z)@(\mathbf{r}@z) : \mathsf{S}\backslash\mathsf{NP}}<}{\mathbf{a}@(\mathbf{j}@\mathbf{m})@(\mathbf{r}@\mathbf{m}) : \mathsf{S}}<$$

We assign the term $\lambda x.\lambda y.\lambda z.\mathbf{a}@(x@z)@(y@z)$ to *and*, abstracting away the NP argument and replicating it inside the term. Using even more complex categories as conjuncts (such as $\mathsf{S}\backslash\mathsf{NP}/\mathsf{NP}$) would require abstracting away additional arguments – as many as the conjunct has – before returning an atomic result.

3.4.2. A remark on terminology

In this document, we use the word *semantics* to mean any kind of information attached to the CCG categories, not necessarily a semantic one in the linguistic sense. While a popular use case indeed is to build a semantic representation of the sentence, there are other possible applications of CCG semantics, such as extraction of the dependency structure, described later in this work.

Chapter 4

PL-CCG, a formalism for the Polish language

In this chapter, we describe problems that make the original CCG formalism inadequate for the Polish language – rich morphology, free word order, sentence modifiers, and certain phrases appearing at both adjunct and complement positions. We introduce several enhancements to the formalism aiming to solve these problems.

The chapter ends with a summary of the new formalism, which we call PL-CCG. In the next chapter, we extend PL-CCG with a variant of lambda calculus as semantics.

As we already said once in the introduction, while PL-CCG attempts to solve a number of problems, it remains more data-oriented than linguistically motivated. The main motivation is an adequate conversion of Polish treebank. Another formalism: Logical Categorial Grammar, a generalization of CCG in the framework of intuitionistic linear logic; is currently under development by Wojciech Jaworski [Jaworski11]. PL-CCG can be considered a simplification, or a special case, of LCG.

4.1. Syntactic features

Because Polish is a highly inflected language, various constituents need to be annotated with syntactic features in order to capture phenomena such as agreement. For instance, in order to accurately parse sentences with noun phrases, we need to consider their case, gender, number and person.

PL-CCG stores attributes in atomic categories, so that e.g. noun phrases have an atomic category of the form $\mathsf{NO}_{case,gender,number,person}$, for instance $\mathsf{NO}_{\mathsf{mian,zen,poj,3}}$ (nominative, feminine, singular, third person[1]). The feature values are extracted from the source data – the conversion process is explained in chapter 7.

Unfortunately, the resulting categories are often too specific. In the current system, the arguments of a functor include all syntactic features. This is obviously a simplification, since e.g. the adjective doesn't agree with the noun on person, the present tense of

[1]The full list of attributes can be found in Appendix A.

a verb doesn't agree with its subject on gender, and a verb doesn't place any restrictions on its noun phrase complements apart from the case.

To use the verb and subject example, the verb form *biegnie* (runs) can have either one of 5 categories

- $\mathsf{S|NO}_{\mathsf{mian,mos,3}}$
- $\mathsf{S|NO}_{\mathsf{mian,mzw,3}}$
- $\mathsf{S|NO}_{\mathsf{mian,mnz,3}}$
- $\mathsf{S|NO}_{\mathsf{mian,zen,3}}$
- $\mathsf{S|NO}_{\mathsf{mian,nij,3}}$,

one for each of the five genders we recognize (masculine personal, animal and inanimate; feminine; and neuter).

To capture agreement accurately, it's possible to include "wildcard" features in the grammar, so that $\mathsf{S|NO}_{\mathsf{mian,\_,3}}$ is a proper category – we would then have to employ pattern-matching in the derivation rules. Other, more fine-grained solutions include introducing a notation for polymorphic features (see [Jaworski11]) or even devising an inheritance hierarchy for the syntactic features, as proposed by Przepiórkowski for genders in [Przepiórkowski03].

In the current version of PL-CCG we don't include any of these. The reason is our data-orientedness mentioned before: wildcard features make sense linguistically, but recovering them from the existing Polish treebank is not straightforward, and beyond the scope of this work.

As a result, the syntactic features introduce no additional complexity to the current derivation rules and most of the time we can safely skip them in our examples. In fact, before chapter 7, introducing the actual treebank, the examples don't use the categories from the treebank at all. Chapter 7 and the subsequent chapter on evaluation mention the real categories, but abbreviate them whenever possible.

4.2. Free word order

The Polish language has a relatively free word order. While the predominant sentence structure is subject–verb–object (SVO), other orderings are also permitted. Because the nouns are inflected by case, a sentence such as *Marysia lubi Janka* (Marysia(nom) likes Janek(acc)) can be also written as:

- *lubi Marysia Janka*
- *Janka lubi Marysia*
- *Janka Marysia lubi*
- etc.

While the other permutations are less common, in all the sentences the predicate is an instance of the same lexeme LUBIĆ. In the CCG formalism as described before, we would

have to assign to it a number of different categories: $\mathsf{S}\backslash\mathsf{NP}_{\mathsf{acc}}/\mathsf{NP}_{\mathsf{nom}}$, $\mathsf{S}/\mathsf{NP}_{\mathsf{nom}}/\mathsf{NP}_{\mathsf{acc}}$, $\mathsf{S}/\mathsf{NP}_{\mathsf{acc}}/\mathsf{NP}_{\mathsf{nom}}$ etc. The problem gets even worse with verbs expecting two complements, such as *dawać* (to give).

The naive solution would be to relax the direction requirement: we could introduce a third type of slash-category, $\mathsf{X}|\mathsf{Y}$, with rules for both forward and backward application (and similar rules for composition and type-raising):

$$(>)\ \ X|Y,\ Y \vdash X \qquad (<)\ \ Y,\ X|Y \vdash X$$

This, however, still doesn't solve the problem completely – the formalism is unable to "forget" the argument order. For the word *lubi* we can use a type such as $\mathsf{S}|\mathsf{NP}_{\mathsf{nom}}|\mathsf{NP}_{\mathsf{acc}}$ or $\mathsf{S}|\mathsf{NP}_{\mathsf{acc}}|\mathsf{NP}_{\mathsf{nom}}$, but there is no single type that would enable parsing both *Marysia Janka lubi* and *Janka Marysia lubi* – one argument always has to be realized before the other.

4.3. Multiset-CCG

A way to relax the argument order has been proposed by Beryl Hoffman as a solution to a similar problem in the Turkish language [Hoffman95]. His formalism, called Multiset-CCG, replaces single arguments with *argument multisets*: collections of argument categories that can be realized in any order. We adapt his solution for our project.

Let an *argument* be a term composed of a slash and category, where slash is one of $\backslash, /, |$. A functor category then consists of a result category and a multiset (unordered sequence) of arguments:

- *Marysia* : NP, *Janka* : NP
- *lubi* : $\mathsf{S}\{|\mathsf{NP}, |\mathsf{NP}\}$
- *biegnie* : $\mathsf{S}\{|\mathsf{NP}\}$

When there is only one argument, we will revert to the old notation: $\mathsf{S}\{|\mathsf{NP}\} \equiv \mathsf{S}|\mathsf{NP}$.

An important feature of the argument multisets is that they don't join, e.g., $\mathsf{S}\{...\}\{...\}$ doesn't collapse into $\mathsf{S}\{...\}$. Instead, the first multiset must be exhausted before we can use the second. A simple example for why this is necessary can be a negative form of a verb, such as *biegnie*: $\mathsf{S}\{|\mathsf{NP}\}\{/\text{'nie'}\}$. With this category, we want to be able to parse a sentence like *Marysia nie biegnie*, but not *nie Marysia biegnie*. If we considered the category to be equivalent to $\mathsf{S}\{|\mathsf{NP}, /\text{'nie'}\}$, the latter sentence would be permitted as well.

The application rules in the new formalism are as follows:

$$(>)\ \ X\{|Y\} \cup \alpha,\ Y \vdash X\alpha \qquad (>)\ \ X\{/Y\} \cup \alpha,\ Y \vdash X\alpha$$

$$(<)\ \ Y,\ X\{|Y\} \cup \alpha \vdash X\alpha \qquad (>)\ \ Y,\ X\{\backslash Y\} \cup \alpha \vdash X\alpha$$

where α is an argument multiset. We consider a "zero-argument" functor category to be the same as a simple category ($X\{\} \equiv X$), so the application rules as stated above are a generalization of CCG application.

The multiset formalism is used by the main version of our lexicon, but we also evaluate an alternate version with a 'no-sets' flag (see section 8.1).

4.4. Sentence modifiers

Another problem involves sentence modifiers, or adjuncts – adverbs, prepositional phrases, interjections and other optional constituents in a sentence. Because of the free word order, most of them can appear in virtually any position in the sentence, that is, between any arguments of the verb. The same modifier can also be used with verbs of different valence. For instance *szybko* (quickly) is the modifier, and arguably the same lexeme, in all of the following:

- *Janek szybko je obiad* (Janek eats dinner quickly)
- *Janek je szybko obiad*
- *Janek je obiad szybko*
- *Janek szybko biegnie* (Janek runs quickly)
- *Janek biegnie szybko*

Clearly *szybko* can't be applied to the predicate (*je*, *biegnie*) alone, because in the third sentence they are not next to each other – and even if it could, we would assign it different categories depending on the verb: $\mathsf{S}\{|\mathsf{NP_{nom}}, |\mathsf{NP_{acc}}\}|\mathsf{S}\{|\mathsf{NP_{nom}}, |\mathsf{NP_{acc}}\}$ or $(\mathsf{S}|\mathsf{NP_{nom}})|(\mathsf{S}|\mathsf{NP_{nom}})$.

A much more uniform solution is to modify the *result* or the whole sentence (S), by assigning to the modifier the type $\mathsf{S}|\mathsf{S}$. Of course, if the rest of the sentence is not next to the modifier, but around it, we need to use the composition rule.

4.5. Multiset composition

For multiset CCG, the following composition rule could be introduced:

$$(>B)\ \ X\{|Z\}\cup\alpha,\ Z\beta\vdash X\alpha\cup\beta \qquad (>B)\ \ X\{/Z\}\cup\alpha,\ Z\beta\vdash X\alpha\cup\beta$$

$$(<B)\ \ Z\beta,\ X\{|Z\}\cup\alpha\vdash X\alpha\cup\beta \qquad (<B)\ \ Z\beta,\ X\{\backslash Z\}\cup\alpha\vdash X\alpha\cup\beta$$

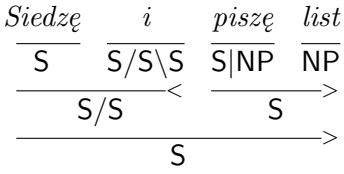
where α, β are argument multisets.

An example derivation utilizing composition is:

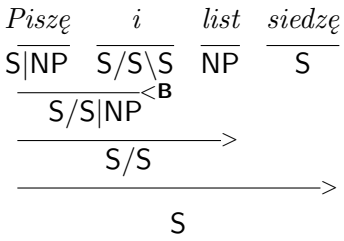
$$\dfrac{\dfrac{\dfrac{\textit{Janek}}{\mathsf{NP_{nom}}} \quad \dfrac{\dfrac{\textit{je}}{\mathsf{S}\{|\mathsf{NP_{nom}}, |\mathsf{NP_{acc}}\}} \quad \dfrac{\textit{szybko}}{\mathsf{S}|\mathsf{S}}}{\mathsf{S}\{|\mathsf{NP_{nom}}, |\mathsf{NP_{acc}}\}}\,{<}\mathbf{B}}{\mathsf{S}|\mathsf{NP_{nom}}}\,{>} \quad \dfrac{\textit{obiad}}{\mathsf{NP_{acc}}}}{\mathsf{S}}\,{<}$$

However, introducing unrestricted composition might have unintended consequences. One is a greater number of derivations produced by the parser, increasing the computational cost. Another, more important one, is that composition can actually lead to incorrect parses. Allowing every constituent to compose relaxes the ordering requirements for the whole sentence, allowing every constituent to behave like a sentence modifier described above.

We demonstrate the problem on a sentence with two coordinated phrases: *Siedzę i piszę list* (I'm sitting and writing a letter). A CCG derivation for that sentence could be:



With the same categories, and unrestricted composition, an incorrect *Piszę i list siedzę* is also accepted:



For that reason, we restrict the composition feature to sentence modifiers only, by introducing special *composable* categories with "double slashes": $X/\!\!/Y$, $X\backslash\!\backslash Y$, $X\|Y$. The categories must have a single immediate argument (i.e. $X\{\|Y, /Z\}$ is not a valid category, but $X\|Y/Z$ is). The composition rule applies only to the composable categories. A similar idea has been implemented by Baldridge [Baldridge03] in the form of multi-modal CCG.

4.6. Modifiers as arguments

Another problem with modifiers (not only the ones acting on a sentence level, but also e.g. adjectives in noun phrases) is that their "modifier" status is not absolute. There are some verbs that require a prepositional phrase with a specific preposition (*idę do domu* – I'm going home) or an adverb (*wyglądam dobrze* – I look good), and others for which the same prepositional phrases and adverbs are just modifiers (*dobrze piszę* – I write well). Even though *dobrze* is in both cases intuitively the same form, it can function either as an adjunct or a complement, and our grammar would assign different categories to it depending on the context.

There are several ways the dilemma can be resolved:

- **Always use the modifier category** for certain phrases (*do domu* : $\mathsf{S}\|\mathsf{S}$) and require it as needed (*idę* : $\mathsf{S}|(\mathsf{S}\|\mathsf{S})$). The problem with this approach is that we lose significant information: there is no way to distinguish a prepositional phrase from an adverb, not to mention different kinds of a prepositional phrase, if all have the same type.
- Add **optional arguments** to the grammar, i.e. allow certain arguments to stay unfulfilled (*wyglądam* : $\mathsf{S}\{|\mathsf{AdvP}\}$, *biegnę* : $\mathsf{S}\{|\mathsf{AdvP}^?\}$). To incorporate as many modifiers as we want in a sentence, we also would possibly need "inexhaustible" arguments ($\mathsf{S}\{|\mathsf{AdvP}^*\}$). This idea requires a more complex grammar formalism and underlying semantics, and is currently being explored by Wojciech Jaworski in [Jaworski11].
- Translate the constituents as atomic (AdvP, $\mathsf{PP}_{\mathsf{do}}$), but include **special derivation rules** in the grammar that convert these categories to modifiers if necessary ($\mathsf{PP} \rightarrow \mathsf{S}\|\mathsf{S}$). Special rules have been added by Hockenmaier [Hockenmaier03] to the English derivation bank as means of converting passive verbal phrases to modifiers (as in the phrase *workers exposed to it*).
- **Don't use special categories** for modifiers – consider all modifiers to be required arguments of a verb in the grammar, and use atomic categories for them (*do domu*: $\mathsf{PP}_{\mathsf{do}}$, *dobrze*: AdvP). This allows us to retain the information about constituent type, but also requires us to include all that information in the verb type, which creates many very specific categories for what should be a single lexical entry of a verb.

  As a refinement, this rule could also apply only to modifiers which legitimately can be confused with verb complements, e.g. prepositional phrases but not interjections.
- **Keep the status quo** – let *dobrze* have a category of AdvP in some cases, and $\mathsf{S}\|\mathsf{S}$ in others, etc.

In our work, we are taking the last approach, but test the one before it as well – in the main version of the corpus, we use distinct categories depending on whether a word is a modifier, and as a baseline, we also evaluate a 'no-mods' flag (see section 8.1). While at the moment we do not implement the special rules approach, we consider it a promising alternative.

4.7. PL-CCG rules

Figure 4.1 summarizes the rules of our formalism. It differs from standard CCG in the following ways:

- The ordinary application rules have been superseded by multiset application (which, of course, still includes the single argument case).
- The vertical slash (|) allows us to relax the directional requirement.

$$(>)\ \ X\{|Y,...\},\ Y \vdash X\{...\} \qquad (>)\ \ X\{/Y,...\},\ Y \vdash X\{...\}$$

$$(<)\ \ Y,\ X\{|Y,...\} \vdash X\{...\} \qquad (<)\ \ Y,\ X\{\backslash Y,...\} \vdash X\{...\}$$

$$(>B)\ \ X\|Z,\ Z\{...\} \vdash X\{...\} \qquad (>B)\ \ X/\!\!/Z,\ Z\{...\} \vdash X\{...\}$$

$$(<B)\ \ Z\{...\},\ X\|Z \vdash X\{...\} \qquad (<B)\ \ Z\{...\},\ X\backslash\!\backslash Z \vdash X\{...\}$$

$$(\|)\ \ X\|Y \vdash X|Y \qquad (\|)\ \ X/\!\!/Y \vdash X/Y \qquad (\|)\ \ X\backslash\!\backslash Y \vdash X\backslash Y$$

Figure 4.1: PL-CCG rules.

- The composition rule is allowed for a special class of composable categories ($\mathsf{X}/\!\!/\mathsf{Y}$). We consider them a sub-class of regular categories (and allow them to be used in application), so we include derivation rules for demoting them.
- We do not use type-raising.

Chapter 5

PL-CCG semantics

In this chapter, we present the semantic part of the previously defined grammar. As in ordinary CCG, the formalism is based on simply-typed lambda calculus. The semantic layer will be later used as a starting point for a content representation layer, but we hope that the system can provide a good foundation which could be extended to accomodate other needs.

We will use a type system with *ordered* argument lists, because it simplifies both describing and implementing the lambda calculus. However, we will also present a mechanism that makes the full system behave in an order-oblivious manner.

A category in this chapter is thus either

- *atomic*, such as S, NP, or
- a *functor*, consisting of a result and an ordered argument list, in which arguments are accompanied with one of $|, /, \backslash$, for instance $\mathsf{S}\langle/\mathsf{NP}, \backslash\mathsf{NP}\rangle$, or
- a *composable functor*, consisting of a result and a single argument: $X/\!\!/Y$, $X\backslash\!\!\backslash Y$, $X\|Y$, where X and Y are categories.

We will omit the directional slashes whenever possible. As before, we assume that a functor with an empty argument list is an atomic category, that is, $Y\langle\rangle \equiv Y$.

5.1. Multiset lambda calculus

The lambda-calculus part of the formalism has to be modified to include multiple arguments. A term can be:

- a variable: x,
- an abstraction: $\lambda\langle x_1, \ldots, x_n\rangle.M$, where M is a term,
- an application: $M@_i N$, where M, N are terms, and i is a positive integer; meaning that M is applied to N as its i-th argument.

The β-reduction rule for the formalism is:

$$(\lambda\langle x_1, \ldots, x_{i-1}, x_i, x_{i+1}, \ldots x_n\rangle.M)\ @_i\ N \quad \rightarrow_\beta \quad \lambda\langle x_1, \ldots, x_{i-1}, x_{i+1}, \ldots x_n\rangle.M[x_i := N]$$

with the usual requirement that the variables don't conflict in the resulting term. In the one-argument case, we drop the empty argument list from the result: $\lambda\langle\rangle.M \equiv M$.

5.1.1. The type system

As in CCG, assigning a category to a term is governed by the typing rules:

- all free instances of a variable in a given term must be of the same type,
- if $x_1 : Y_1$, $x_2 : Y_2$, ..., $x_n : Y_n$ in M, and $M : X$, then the term $\lambda\langle x_1, \ldots, x_n\rangle.M$ must have the type $X\langle Y_1, \ldots, Y_n\rangle$,
- in an application $M@_iN$, M must have at least i arguments, and if $M : X\langle Y_1, \ldots, Y_n\rangle$, then $N : Y_i$, and the whole term has type $X\langle Y_1, \ldots, Y_{i-1}, Y_{i+1}, \ldots, Y_n\rangle$.

The results for the previous version of lambda calculus still apply:

Theorem 3. *If $T \to T'$, and $T : X$, then $T' : X$.*

Proof. A trivial modification of the proof for Theorem 1. □

Theorem 4 (Strong normalization)**.** *If M is a typed term, then every β-reduction sequence beginning with M terminates at the same unreducible term M'.*

Proof. See again the second chapter of [Urzyczyn03]: the proof makes use of the fact that two β-redexes in a term never partially overlap, which is still the case in our formalism. □

5.1.2. Reordering the arguments

Because the calculus defined above uses ordered argument lists, one more step is necessary before we define the semantics for PL-CCG. To make the derivation rules "ordering-oblivious", we need to be able to reorder the arguments.

Let $reorder(M : X)$ be the set of all results of argument reordering in term M of category X. The process has to recursively consider all arguments and results. Nonetheless, it is a straightforward use of abstraction and application, so instead of including a verbose definition, we will give an example:

$$reorder(term : X\langle/Y, |Z\rangle) = \{term : X\langle/Y, |Z\rangle, \\ \lambda\langle x, y\rangle.term@_1y@_1x : X\langle|Z, /Y\rangle\}$$

5.2. PL-CCG rules with semantics

We can now present the semantic version of the PL-CCG derivation rules. We begin with the application and composition rules (the slashes in arguments are omitted for brevity).

$$(>) \quad f : X\langle Y_1, \ldots, |Y_i, \ldots, Y_n\rangle, a : Y_i \vdash f@_ia : X\langle Y_1, \ldots, Y_{i-1}, Y_{i+1}, \ldots, Y_n\rangle$$

$(> B)\ \ f : X \| Z,\ g : Z\langle Y_1, \ldots, Y_n\rangle \vdash \lambda\langle x_1, \ldots, x_n\rangle . f @_1 (g @_1 x_1 @_1 \ldots @_1 x_n) : X\langle Y_1, \ldots, Y_n\rangle$

The rules are analogous for the other variations of connective and direction.

The demotion rule (also in three variations) converts a composable functor to a regular one, without changing the underlying term.h

$$(\|)\ \ t : X \| Y \vdash t : X | Y$$

We also need a rule for β-reduction:

$$(\beta)\ \ t : X \vdash t' : X \ \text{ if } t \to_\beta t'$$

Theorem 3 allows us to keep the same type for t'.

The final rule allows us to reorder the arguments in terms, so that equivalent types can be used in place of each other.

$$(R)\ \ t : X \vdash t' : X' \ \text{ if } t' : X' \in \mathit{reorder}(t : X)$$

Because of the reordering rule, the above system should be syntactically equivalent to the one presented at the end of the previous chapter.

Chapter 6

Content trees

We have now fully defined an extension of the syntactic and semantic rules of CCG to the multiset case. To use the system for actual parsing, we need one more element: a *content tree* that will be the output of our parser. Because our aim is to extract syntactic dependencies, the resulting structure will encode them, albeit in not entirely straightforward way.

The representation will be of course built during the CCG derivation, using the lambda calculus defined before. The partial results will consist of lambda-terms "scaffolding" supporting the tree nodes. However, we will show that the final result of parsing will always be a pure content tree, with all the scaffolding gone. Because of that, we can present the content representation layer in separation, and only later show how it is added to the semantics.

We begin with a few examples explaining the rather complex shape of the system. Afterwards, we formally define the content trees, and present the algorithm used to extract dependencies from them. Finally, we reintroduce the content trees as part of the lambda calculus defined in the previous chapter.

6.1. Motivation

The most obvious semantic representation is an application of verb to its arguments, as in chapter 3, or equivalently, a tree. For *Marcel proved completeness*, it's easy to see how a term like $\mathbf{p}(\mathbf{c}, \mathbf{m})$ could be transformed into dependencies (*proved*, *completeness*), (*proved*, *marcel*). Labeled dependencies could also be easily encoded in such a format by annotating the tree: $\mathbf{p}(subj = \mathbf{m}, obj = \mathbf{c})$.

The modifiers are a complication, because there is no obvious way to attach them to the tree using lambda-terms. For a sentence like *A quick brown fox jumps over the lazy dog*, the structure we want is

$$jumps(fox(quick, brown), over(dog(lazy)))$$

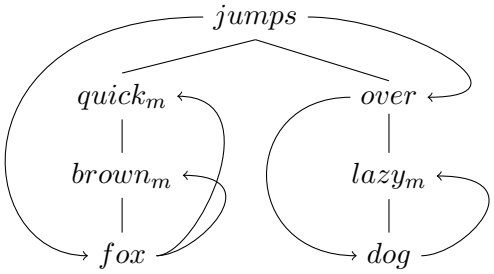
or some variant of the above utilizing simple application (@).

This, however, requires assigning lambda-terms like $\lambda\langle n\rangle.n@quick$ to the adjectives. A system like that is hard to extend with second-order modifiers – modifiers acting on other modifiers, such as adjectival adverbs. Because our adjective is not a simple node, but a lambda-abstraction, there is no easy way to "attach" anything to it.

What we do instead is use the modifier as the *parent* of the node being modified. For our sentence, the representation is:

$$jumps(quick(brown(fox)), over(lazy(dog)))$$

Naively extracting dependencies from such a tree leads to an obviously wrong result, because e.g. *quick* is a special kind of node, and not the real head of its noun phrase. We need to skip modifiers such as *quick* and *brown* when we look for the dependent of *jumps*. The simplest way to achieve this is to mark the modifiers as special:



The rules are simple – when looking for a dependent, we go down from the head skipping the modifier nodes. We alos reverse the dependencies originating from modifiers, so that *quick* is the dependent of fox, not the other way around.

This design, however, still fails to account for the second-order modifiers. Consider the phrase *very big dog*. The natural assignment of categories is

- *dog* : NP
- *big* : NP/NP
- *very* : (NP/NP)/(NP/NP)

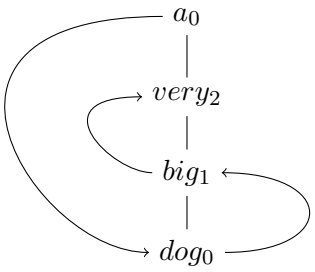
and causes the term for *very* to be applied to the term for *big*, which corresponds to the expected syntactic dependencies. However, translating that to the term representations is hard to achieve. The functor category NP/NP might be a type of some lambda-abstraction, and a term like $very(\lambda x.\dots)$ does not tell us much about the resulting dependencies.[1]

We therefore use the modifier as a parent of *the whole phrase*, which in the example gives us $very_m(big_m(dog))$. Of course, knowing that *very* and *big* are modifiers is not

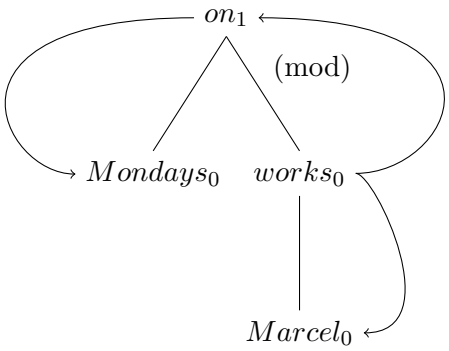
[1] One possible solution would be to introduce a special kind of application to our calculus, effectively including a special case for modifiers in the reduction rules. The downside is that such a reduction system is both harder to reason about and more domain-specific. We decided to keep to the well-known core of lambda calculus, and build our representation as an additional layer on top of it.

enough to extract the dependencies, because *very* has to modify *big*, not *dog*. We make the system work by refining of the annotation idea. Each node is given a *modifier order*. Order-1 modifiers act on ordinary (order-0) nodes, order-2 modifiers act on order-2 modifiers, etc.

The following tree shows the result for the phrase *a very big dog*. The article is used as a head of the entire phrase, to show how we skip over both modifiers to find its dependent.



Finally, a modifier node can itself have dependents, as is the case with prepositional phrases. In the sentence *Marcel works on Mondays*, the preposition *on* accepts a noun phrase as its complement, then acts on a sentence ($\mathsf{S}\backslash\mathsf{S}/\mathsf{NP}$). Because of that, our modifier nodes must have ordinary arguments and a special "modified" argument.



Our trees also have to include the dependency labels, which we will add in the final version presented in the next section.

6.2. Definition of a content tree

Having briefly explained the reasons for our design, we can now define it formally. A *content tree*, or a content node, is either

- $id_0[arg_1, \ldots, arg_n]$ – a *regular* content node, or
- $id_i\{arg_0\}[arg_1, \ldots, arg_n]$ – a *modifier* content node, where $i > 0$ is the *modifier order*.

id is a textual node identifier (e.g. orthographic form of a given word[2]), and each arg is of the form $d = node$, where d is the dependency label, and $node$ is again a content node. Arguments $arg_1 \ldots arg_n$ are called *ordinary arguments*, and arg_0 in the modifier node is a *modifier argument*.

In the above definitions, n can be 0, in which case the argument list $[arg_1, \ldots, arg_n]$ is omitted.

6.3. Dependency extraction algorithm

The method of extracting the dependencies has been specified very informally in the section on motivation, but now we can give it in detail. First, we recursively define the operation $find(i, node)$ that looks for an order-i node, skipping over modifiers of higher order:

$$find(i, node) = \begin{cases} id & \text{if } node = id_0[\ldots] \quad \text{and } i = 0 \\ node' & \text{if } node = id_k\{d = node'\}[\ldots] \quad \text{and } i = k \\ find(i, node') & \text{if } node = id_k\{d = node'\}[\ldots] \quad \text{and } i < k \end{cases}$$

We use $find$ to define $deps(node)$, which extracts all dependency relations originating in a given node:

$$deps(node) = \begin{cases} \{(id, d^1, find(0, node^1)), \\ \ldots, \\ (id, d^n, find(0, node^n))\} & \text{if } node = id_0[d^1 = node^1, \ldots, d^n = node^n] \\ \\ \text{the above and} \\ (find(k-1, node^0), d^0, id) & \text{if } node = id_k\{d^0 = node^0\}[\ldots] \end{cases}$$

We invoke *deps* for each node in the content tree. The result is a set of triples $(id^1, deprel, id^2)$ describing the dependency relations $(id^1 \rightarrow_{deprel} id^2)$. We also use $find(0, tree)$ on the full tree to recover the root of a phrase.

6.4. Content trees in lambda calculus

In the actual derivation, the content trees are a part of the semantics. We simply use the content node as another kind of term, along with abstraction, application and variables. A content node must be of an atomic type.

We allow other lambda terms as arguments of the content nodes, however, we also require their types to be atomic, so that the whole structure can later be β-reduced to a "pure" content tree. In fact, thanks to the typing rules the reduction is guaranteed:

[2] In the implementation, apart from the orthographic form we store the word's position in a sentence, to disambiguate between repeated words in a sentence. For clarity, we omit that detail in this chapter.

Theorem 5. *If a term M has an atomic category (e.g. S), contains no free variables, and is in normal form, then M is a well-formed content tree (i.e. contains no terms other than content nodes).*

Lemma. *If a term Q has a functor category, contains no free variables, and is in normal form, then it is an abstraction.*

Proof. Obviously, Q cannot be a variable or a content tree. Let's suppose it's an application: $Q = Q'@_i W$. We know that Q is in normal form, has a functor type, no free variables, and is not an abstraction (otherwise Q would not be in normal form). By the same argument, Q must be also an application: $Q' = Q''@_j W'$, etc. We can repeat the process *ad infinitum*, but Q is a term and as such contains a finite number of applications. We conclude that our inital assumption was wrong and Q cannot be an application. □

Proof of the main theorem. Let T be the topmost subterm of M that is not a content tree. Because it is topmost, it must be either the whole M, or one of subtree arguments. Therefore, the type of T is atomic. If it's not a content node, it must be either

- a variable: impossible, because it would be a free variable,
- an abstraction: impossible, because no abstraction has an atomic type,
- an application of an one-argument functor: $T = Q@_1 R$, $Q : X/Y$ (or $Q : X|Y$, $Q : X\backslash Y$). Impossible, because by the Lemma Q is an abstraction, so M is not in normal form.

Therefore, T cannot exist and M is a well-formed content tree. □

6.5. Examples

We now show how all the elements described in this chapter (content trees, dependency extraction algorithm, integration with lambda calculus) are used together to describe natural language phrases. "Standard" semantic representations for a few parts of speech are shown. While the actual representations we use in PL-CCG treebank are determined from the data, this section should give a good idea of how they look for some simple cases.

For the example sentence, *Marcel works on Mondays*, we annotate the words as follows:

$$
\begin{aligned}
Marcel &:= && Marcel_0 : \mathsf{NP} \\
works &:= && \lambda\langle x\rangle.works_0[subj = x] : \mathsf{S\backslash NP} \\
on &:= && \lambda\langle x\rangle.\lambda\langle y\rangle.on_1\{adjunct = y\}[comp = x] : \mathsf{S\backslash S/NP} \\
Mondays &:= && Mondays_0 : \mathsf{NP}
\end{aligned}
$$

Parsing the sentence and β-reducing the result returns the folowing:

$$on@_1 Mondays@_1(works@_1 Marcel) \rightarrow^*_\beta$$
$$on_1\{adjunct = works[subj = Marcel]\}[comp = Mondays] : \ \mathsf{S}$$

The content tree described by the above term is shown in figure 6.1. The extracted dependencies are (*works*, subj, *Marcel*), (*works*, adjunct, *on*), (*on*, comp, *Mondays*), (*ROOT*, root, *works*).

Another example, for the phrase *a very big dog*, shows why modifier order annotations are necessary. The terms are[3]:

$$\begin{aligned} a :=& \quad a_0 : \ \mathsf{NP/NP} \\ very :=& \quad \lambda\langle a\rangle.\lambda\langle n\rangle.very_2\{adv = a@_1 n\} : \ \mathsf{(NP/NP)/(NP/NP)} \\ big :=& \quad \lambda\langle x\rangle.big_1\{adj = x\} : \ \mathsf{NP/NP} \\ dog :=& \quad a_0 : \ \mathsf{NP} \end{aligned}$$

The parser then returns:

$$a@_1((very@_1 big)@_1 dog) \rightarrow^*_\beta$$
$$a_0[very_2\{adv = big_1\{adj = dog_0\}\}] : \ \mathsf{NP}$$

Figure 6.2 shows the resulting content tree, from which we extract the dependencies: (*a*, art, *dog*), (*big*, adv, *very*), (*dog*, adj, *big*), (*ROOT*, root, *a*).

[3] For simplicity, we assume that NP is at the same time the type of bare nouns and of noun phrases. A more accurate lexicon, such as the one in [Hockenmaier03], could assign N to *dog* and NP/N to *a*. The lambda-terms used would be the same in either case.

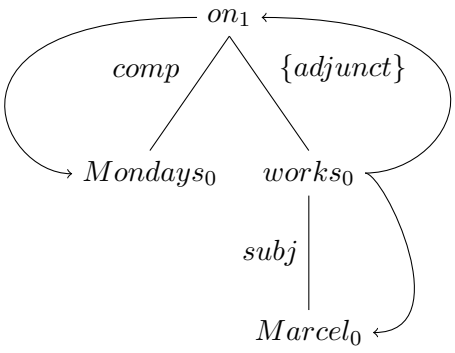


Figure 6.1: Content tree for the sentence *Marcel works on Mondays.*

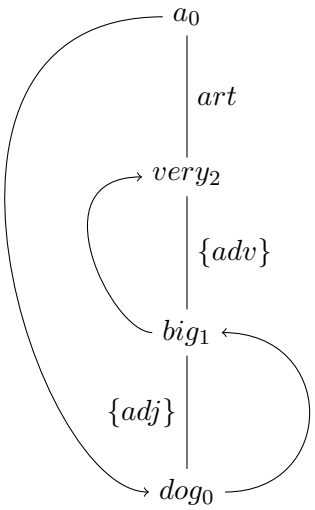


Figure 6.2: Content tree for the noun phrase *a very big dog.*

Chapter 7

PL-CCG treebank

This chapter describes the process of building the PL-CCG treebank. First, an analogous work for the English language and its treebank (Penn Treebank) is described. The subsequent section describes the available data – a Polish constituency trees bank called *Składnica* [Sklad]. We then detail the algorithms used to convert the constituency trees to PL-CCG derivations.

7.1. CCGbank

The work presented here is heavily inspired by Julia Hockenmaier's CCGbank project [Hockenmaier03]. CCGbank is a bank of CCG derivations converted from Penn Treebank, a manually annotated corpus of English syntax trees. This is an example sentence in Penn Treebank:

```
(S
  (NP-SBJ (NNP Mr.) (NNP Vinken))
  (VP
    (VBZ is)
    (NP-PRD
      (NP (NN chairman))
      (PP
        (IN of)
        (NP
          (NP (NNP Elsevier) (NNP N.V.))
          (, ,)
          (NP (DT the) (NNP Dutch) (VBG publishing) (NN group))))))
  (. .))
```

The parse tree is shown in the original bracket notation. The structure includes the following node labels:

- *part-of-speech tags* for single words, such as proper noun (`NNP`) or present tense verb (`VBZ`)

- *chunk tags* describing other constituents: noun phrase (NP), verb phrase (VP), sentence (S)
- *relation tags* that clarify the role of a constituent, such as subject (-SBJ) or predicate (-PRD).

For a more detailed description of Penn Treebank, see chapter 3 of [Hockenmaier03].

Hockenmaier's algorithm produces the following output for the tree:

```
{S[dcl]
  {S[dcl]
    {NP {N {N/N  Mr.} {N  Vinken}}}
    {S[dcl]\NP
      {(S[dcl]\NP)/NP  is}
      {NP
        {NP {N chairman}}
        {NP\NP
          {(NP\NP)/NP of}
          {NP
            {NP
              {N {N/N  Elsevier} {N  N.V.}}}
              {NP[conj]
                {, ,}
                {NP
                  {NP[nb]/N  the}
                  {N
                    {N/N  Dutch}
                    {N
                      {N/N  publishing}
                      {N  group}}}}}}}}}}
  {.  .}}
```

The output describes a CCG derivation. CCGbank uses special derivation rules for a number of cases, including getting rid of sentence-ending punctuation ($\mathsf{S}\ . \rightarrow \mathsf{S}$), special conjunction rules (, $\mathsf{NP} \rightarrow \mathsf{NP_{conj}}$, $\mathsf{NP}\ \mathsf{NP_{conj}} \rightarrow \mathsf{NP}$) and auto-promotion of N ($\mathsf{N} \rightarrow \mathsf{NP}$). The rest of the derivation shown here follows the application rule.

The CCGbank utilizes only a small number of atomic categories: noun (N), noun phrase (NP), prepositional phrase (PP, not shown here), sentence (S) and categories for punctuation. Some of the categories include a special feature further describing its type, such as declarative sentence ($\mathsf{S_{dcl}}$) or noun-bare noun phrases ($\mathsf{NP_{nb}}$). The features are matched during the derivation with no feature acting as wildcard, so that $\mathsf{S_{dcl}}$ and S match, but $\mathsf{S_{dcl}}$ and $\mathsf{S_{wh}}$ (a sentence with who-question) don't.

Figure 7.1 shows the same derivation in a graphical form, simplified by collapsing some of the noun phrases and omitting the ending full stop.

The conversion process is fully automatic. While there are some special cases, the general algorithm is simple:

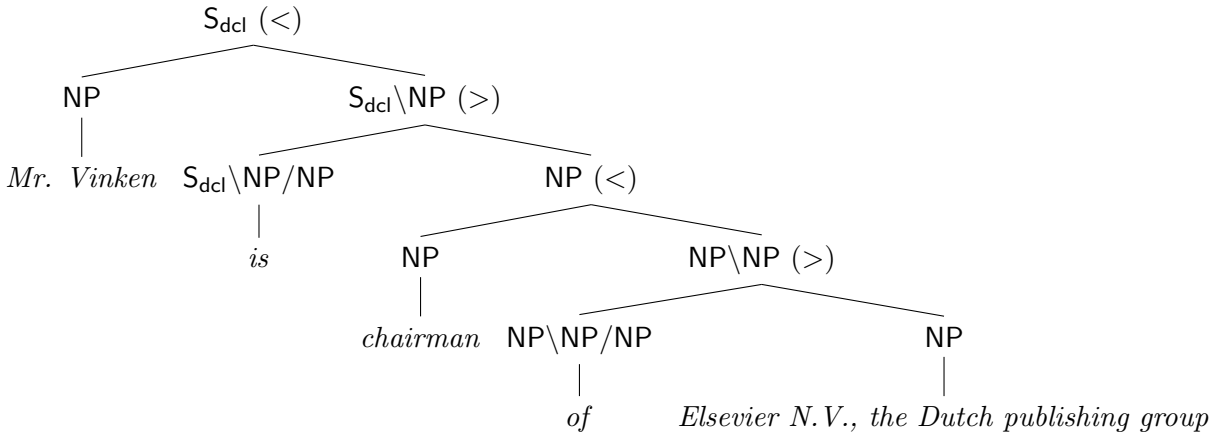


Figure 7.1: Simplified derivation for the example sentence in CCGbank.

- for each subtree, find its syntactic head,
- classify the remaining children nodes as complements or adjuncts,
- binarize the tree, starting from the nodes on the right side of the head, then nodes on the left,
- convert the resulting binary tree into a derivation using application (for complements) and composition (for adjuncts).

It's worth noting that Penn Treebank contains neither the phrase head, nor complement vs. adjunct information. Both have to be determined heuristically.

7.2. Polish treebank (*Składnica*)

Składnica, the Polish treebank [Sklad], is a corpus of grammatical derivations for (as of May 2012) 8227 Polish sentences. The sentences were first parsed by *Świgra*, a parser built by Marcin Woliński [Wolinski04] based on a definite clause grammar (DCG) described by Marek Świdziński [Swidzinski92]. The correct parse tree for each sentence was then chosen by human annotators.

The parse trees are more detailed than in Penn Treebank: they include information about head of each phrase, and distinguish between adjuncts and complements of a predicate.

Figure 7.2 shows an example tree for the sentence *Omawiana książka jeszcze raz to potwierdza.* (The discussed book confirms that again).

To summarize the labels in the tree:

- The top level is an utterance (`wypowiedzenie`) and contains a sentence (`zdanie`) and the ending full stop (`znakkonca`).

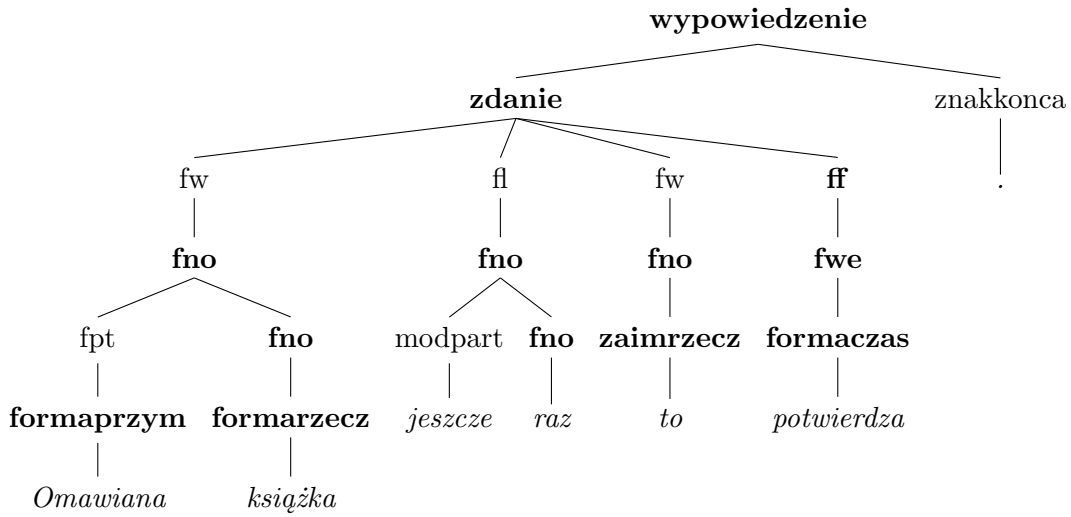


Figure 7.2: Simplified parse tree for the sentence *Omawiana książka jeszcze raz to potwierdza* in *Składnica*.

- A sentence consists of a finite phrase (`ff`) containing the verb; accompanied by up to three required phrases (`fw`) acting as the subject and complements, and several free phrases (`fl`) – typically sentence adjuncts, but also e.g. interjections.
- After specifying their function in a sentence, constituents are in turn categorized as verbal phrase (`fwe`), nominal phrase (`fno`) and adjectival phrase (`fpt` under the nominal phrase).
- On a lower level, the constituents are typically single-word, and roughly correspond to parts of speech: adjectival and noun forms (`formaprzym`, `formarzecz`), particle modifier (`jeszcze`), nominal pronoun (`zaimrzecz`), verb form (`formaczas`).
- The lowest level consists of terminal symbols, containing the actual orthographic forms.

The tree includes information about the head of every node (here in bold face).

Apart from the labels, the nodes also contain several syntactic features. The full tree, with all features visible, is shown in figure 7.3.

The features relevant to our conversion algorithm described later include:

- morphological features: case (`przypadek`), number (`liczba`), gender (`rodzaj`), person (`osoba`),
- the preposition in prepositional phrases (`przyim`, not shown here),
- type of required phrase (`tfw`), similar to the relation tag in Penn Treebank: subject (`subj`), accusative noun phrase complement (`np(bier)`), various types of prepositional phrases (`prepnp`, `prepadjp` – not shown here), etc.

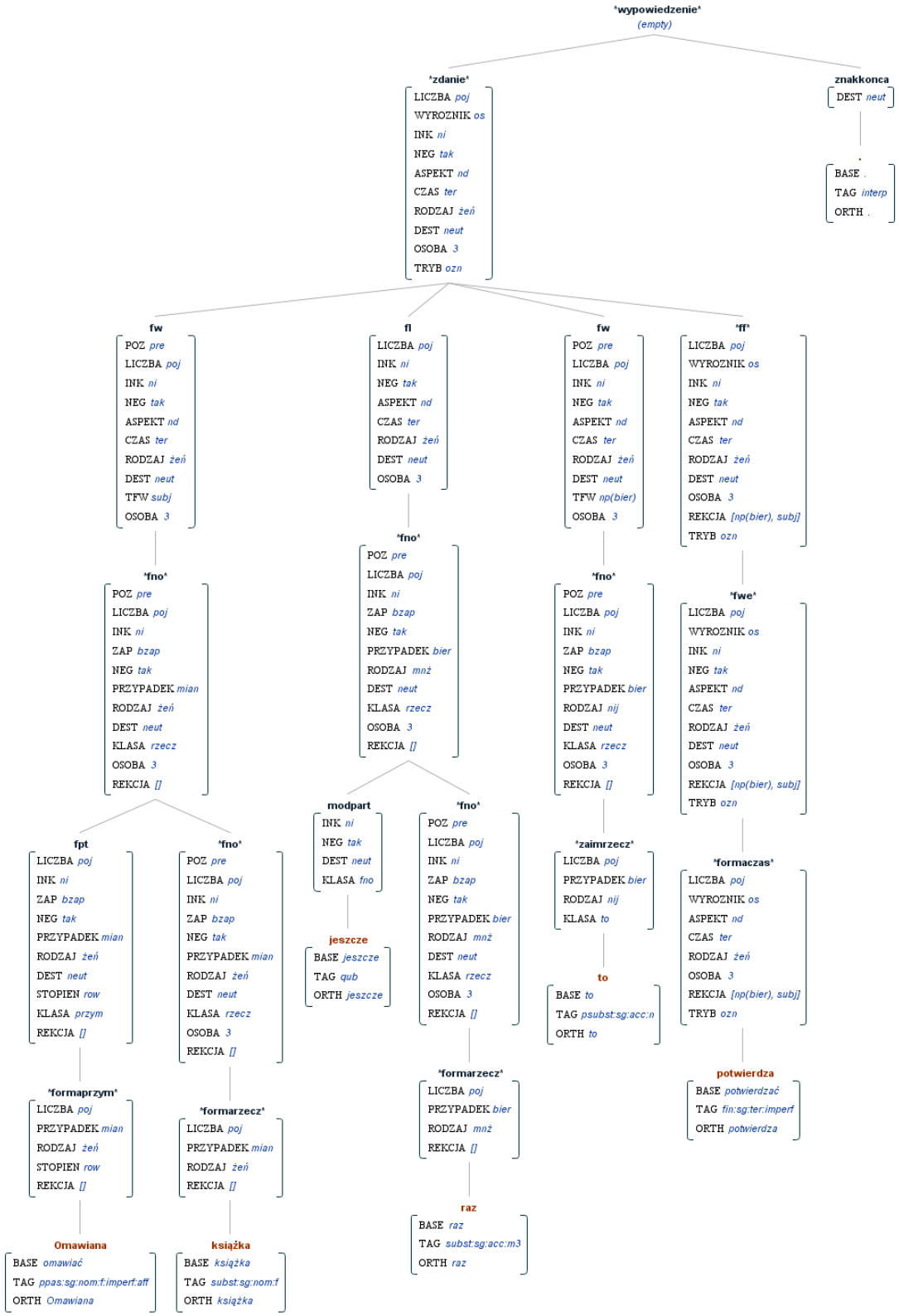


Figure 7.3: Full parse tree for the sentence *Omawiana książka jeszcze raz to potwierdza* in *Składnica*.

A list of all symbols in *Składnica* (including categories, features, and feature values) is included as Appendix A.

7.3. The conversion algorithm

In this section, we describe the process of converting parse trees from *Składnica* to CCG derivations.

We start by describing how the nodes are classified as head, arguments and modifiers. We then describe the process of assigning categories to the nodes and ultimately to the single words. Finally, we briefly describe the dependency labeling and assignment of the semantic information.

7.3.1. Classifying the node elements

The first step of the algorithm is to classify every element (i.e. child) of a given node as a head, argument, or modifier.

- A *head* is the element that will be applied to all the arguments in the derivation. Exactly one child of every node is marked as head.
- A *modifier* is an "optional" element that will be either applied to the partial result, or composed with it. Its existence does not influence any other categories – an essentially the same derivation can be constructed without it.
- All other elements are considered *arguments*.

The head element

In most cases, the information about head element is taken directly from the corpus. Currently, there are two exceptions to that rule:

- Some constituents, such as sentences of the form *Gdyby ..., to ...* don't contain information about the head element. We use the first child node as head.
- Verbs with conditional auxiliary suffixes (`condaglt`), such as *zrobiłby*, are stored in the corpus as root and auxiliary parts (*zrobił+by*), with the auxiliary marked as head. To maintain consistency with other constructions with the same auxiliaries, (*on by to zrobił*), we instead use the verb root as head, and the auxiliaries as arguments.

Modifiers

We classify the following nodes as modifiers:

- subtrees explicitly specified as modifiers in the corpus: `modpart` (particle modifier), `modjaki` (*jaki*-type modifier), `fl` (free phrase),

Phrase	Node type	Category	Features
nominal	fno	NO	case, gender, number, person
adjectival	fpt	PT	case, gender, number
prepositional-nominal	fpm	PM	preposition, case
prepositional-adjectival	fpmpt	PMPT	preposition, case
adverbial	fps	PS	
verbal	fwe	WE	
sentential	fzd	SS	type
sentence	zdanie, wypowiedzenie	S	
free	fl	L	

Table 7.1: Atomic categories used in the PL-CCG treebank. The L category is used for the multi-word free phrases in the version without modifiers (see section 8.1).

- adjectives (`fpt`), adverbs (`fps`) and prepositional phrases (`fpm`, `fpmpt`), except in a required phrase (`fw`) in a sentence, or as a part of another node of the same kind (which happens in coordinate phrases),
- punctuation at the beginning (`pauza` – initial dash in dialogue lines) and at the end (`znakkonca`, e.g. '.', '?', '!') of the sentence, allowing us to effectively ignore it and simplify the other categories. A similar conversion has been done in [Hockenmaier03] by introducing special derivation rules (S . $\rightarrow$ S).

7.3.2. Building the derivation

We then select the categories for elements, at the same time building the derivation. The algorithm recursively considers every node in the graph, starting from the top. Apart from the head–modifier–argument classification described above, we are given a *required result* for the given part of a derivation, that is, the category of the node we are currently processing. The top-level required result is S.

We first describe how the arguments are processed, and how the algorithm constructs a derivation tree out of head and argument nodes. We then explain how the modifiers fit in the process. Finally, we describe the exceptions made to allow for free word order.

The argument nodes

Processing the argument nodes is relatively simple: they receive the atomic categories. The categories are determined given the information in the node type.

Compared to CCGBank, which in essence uses only three categories (S, NP, PP), the category set in our work is much more complex. We also extract syntactic features and include them in the categories. Table 7.1 lists all the atomic categories.

After we assign an atomic category to a given node, we can recursively process the nodes inside it (with that category as the required result).

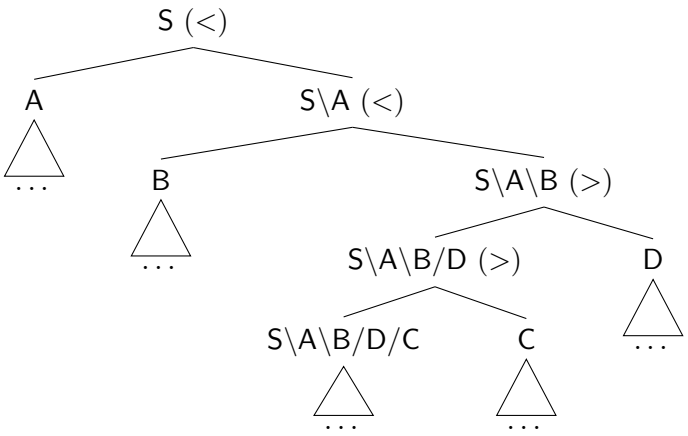
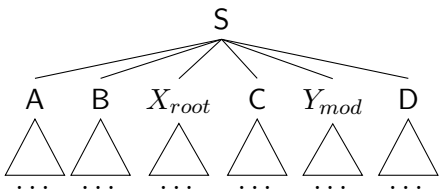


Figure 7.4: Example derivation without modifiers.

The basic derivation

When all the arguments have been processed, we can proceed to build a derivation. We will ignore the modifier nodes for now.

We will trace the process for an example tree with nodes A, B, X, C, Y, D, where $\mathsf{A}, \mathsf{B}, \mathsf{C}, \mathsf{D}$ are the categories of arguments, Y is determined to be a modifier, and X is the head node (X and Y have as of yet undetermined categories). The category for the whole node is S.



We assume that the head is applied first to all arguments to the right, then to all arguments to the left. The resulting category mirrors that derivation.

In our example, the order of applications is C, D, B, A, and the category of the head node is S\A\B/D/C. Figure 7.4 shows the derivation.

After we determine the category of the head node, we can also recursively process it with that category as the given result.

Adding the modifiers

The modifiers can now be inserted into the derivation, with a category that allows them to modify the partial application result. In our example, the modifier Y receives category (S\A\B/D)\(S\A\B/D), because it is applied after we have used the argument C (see figure 7.5).

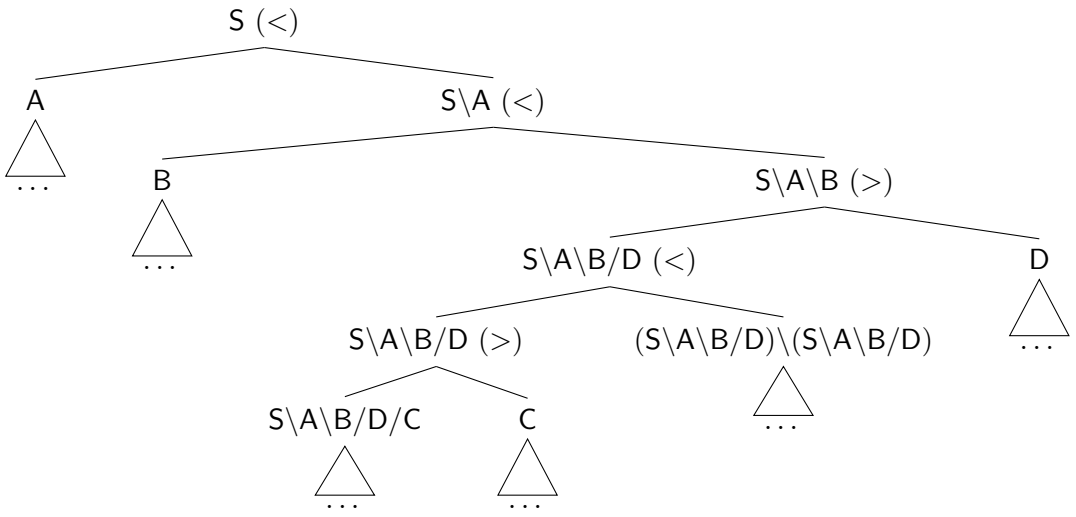


Figure 7.5: Example derivation with the modifier included.

As before, we can recursively process the modifiers after we have determined their category.

Word order relaxation

Some constituents are classified by our process and *unordered* and transformed using a slightly different process:

- the head node's category is an argument multiset with undirected arguments (in our case, $\mathsf{S}\{|\mathsf{A}, |\mathsf{B}, |\mathsf{D}, |\mathsf{C}\}$),
- the sentence modifiers modify *the result* and not the partial result. Sentence modifiers receive composable, undirected categories (in our case, $\mathsf{S}\|\mathsf{S}$) and the resulting derivation utilizes composition if necessary.

Figure 7.6 shows a version of our example derivation for unordered nodes.

The word order relaxation applies to sentences (`zdanie`), sentential phrases (`fzd`), and verbal phrases (`fwe`); with the exception of phrases that don't follow the usual "finitive phrase + required phrase + free phrases" structure (that is, phrases containing none of `ff`, `fw`, `fl`). These include verbal phrases of the form "*nie*+verb" and coordinate clauses – in effect, in a coordinate clause the constituent clauses are order-relaxed, but the top-level `zdanie`+'i'+`zdanie` is not.

We also consider some non-sentence modifiers *undirected*, converting their slashes to undirected ones (|). These are adjectives, adverbs and prepositional phrases (`fpt`, `fps`, `fpm`, `fpmpt`) classified as modifiers. We only lose the directionality requirement, not the order requirement (e.g. prepositions in noun phrases are $\mathsf{NO}|\mathsf{NO}/\mathsf{NO}$, not $\mathsf{NO}\{|\mathsf{NO}, /\mathsf{NO}\}$).

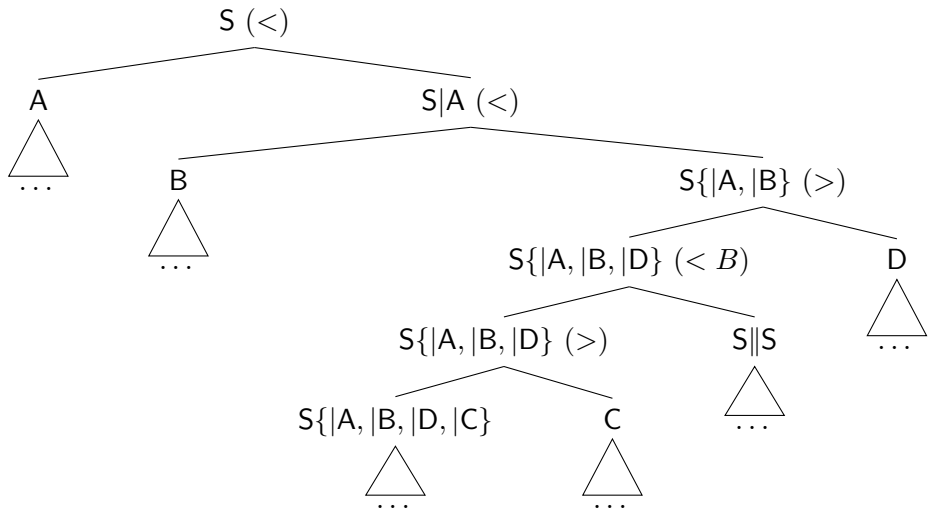


Figure 7.6: Example unordered derivation.

7.3.3. Dependency extraction

We can extract syntantic dependencies from the data gathered already. Basically, for every head-argument application, and every modifier-head application (or composition) we connect the two constituents with an unlabeled dependency relation. Connecting the constituents in turns means connecting their root terminals, i.e. the terminal elements determined by repeatedly selecting the head of a subtree.

After that, we need to label the relations. Unfortunately the source data in most cases does not include dependency annotation, which makes the labelling hard.[1] We use only a limited set of dependency relations:

- *subj*, for subjects of a predicate (marked in the treebank by the required phrase type (`tfw`) attribute),
- *punct*, for the punctuation beginning and ending the sentence (`pauza`, `znakkonca`),
- *aux*, for all auxiliary verbs and particles (*nie*, *się*, verb suffixes such as *-by*) – all nodes of types `condaglt`, `posiłk` and `partykuła`, and all arguments in a verb form (`formaczas`),
- *adjunct*, for all other modifiers,
- *comp*, for all other non-modifiers,
- *root*, for the sentence root (that is, the main predicate X always receives a dependency relation ($ROOT$, root, X) from the parser).

[1] Such labelling has been done before – [Wroblewska11] describes the Polish Dependency Bank, which is a conversion of trees from *Składnica* to labeled dependencies. We decided to use the constituent,

The set of labels serves mostly the process of evaluation, and should be regarded more as a proof of concept than any linguistically meaningful construction.

7.3.4. Constructing the semantic representation

We now have all the components necessary to assign the lambda-terms for the words – word categories, the derivation, and dependency labels.

The simplest case is the atomic category. For these, we just assign a content node with no arguments. For instance, the proper name *Marysia* : NO receives the term $marysia_0$.

If the word has a functor category, we can identify in the derivation a chain of applications that ends in one of two ways:

- the word is finally applied to some other constituent – it's a modifier,
- the word is finally used an argument, or becomes the final S of a sentence – it's an ordinary node.

We can now retrace the chain of applications and build the appropriate lambda term that uses the dependency labels. For the ordinary nodes, the resulting term will be

$$\lambda\langle\ldots\rangle\ldots\lambda\langle\ldots\rangle.id_0[d^1 = x^1, \ldots, d^n = x^n]$$

where id is the word identifier, $d^1 \ldots d^n$ are dependency relations, and the abstractions use the arguments $x^1 \ldots x^n$. For instance, the verb form *lubi* : S{|NO, |NO} (likes) could receive a term

$$\lambda\langle x, y\rangle.lubi_0[subj = x, comp = y]$$

For modifiers, the last argument is the modified one, i.e. the one on the special 0-th position in the content node. A simple case of a modifier is an adjective like *wielki* : NO|NO, which receives a term

$$\lambda\langle n\rangle.wielki_1\{adj = n\}$$

A modifier can be applied to several ordinary arguments before the last, modified one. This is the case which prepositions, which first require a nominal phrase, and only then modify the sentence. For instance, the preposition *do* : S‖S/NO receives a term

$$\lambda\langle n\rangle.\lambda\langle s\rangle.do_1\{adjunct = s\}[comp = n]$$

The last special case concerns *higher-order modifiers*. As we said in section 6.1, if our last argument is a modifier (i.e. we're modifying a modifier), we want the *entire phrase*

not the dependency version of the treebank, because the constituent format retains more structure and allows for easier classification of subtrees as ordered or unordered.

Unfortunately, at the time of writing this the dependency extraction algorithm mentioned in [Wroblewska11] has not yet been released. In the future, it might be possible to incorporate Wróblewska's dependency classification method in the PL-CCG building process.

(including that modifier's argument) to be inside our term, so we need to abstract away additional arguments and use them again inside the term.

An example higher order modifier could be *bardzo* : (NO|NO)|(NO|NO) (very), an adjective-modifying adverb. The term our algorithm assigns to *bardzo* is

$$\lambda\langle a\rangle.\lambda\langle n\rangle.bardzo_2\{a@_1 n\}$$

The pattern of abstracting away additional arguments, then using them inside the term, is the same one we used for semantics of conjunction back in the beginning (section 3.4.1).

7.3.5. An example output of the algorithm

Figure 7.7 shows the derivation our algorithm produces for the sentence *Omawiana książka jeszcze raz to potwierdza*, quoted earlier in section 7.2.

Along with categories, the derivation builds a content representation. As shown in figure 7.8, lambda-terms attached to the words are combined, ultimately resulting in a complete content tree in the top-level node – the parsing result. To extract the dependency structure, we use the extraction algorithm (section 6.3) on that content tree.

The extracted dependencies are:

(*ROOT*, root, *potwierdza*(5)),
(*potwierdza*(5), subj, *książka*(1)),
(*potwierdza*(5), adjunct, *raz*(3)),
(*potwierdza*(5), comp, *to*(4)),
(*potwierdza*(5), punct, .(6)),
(*książka*(1), adjunct, *Omawiana*(0)),
(*raz*(3), adjunct, *jeszcze*(2)).

The numbers in the dependency relations ((5), etc.) are the word positions in the sentence.

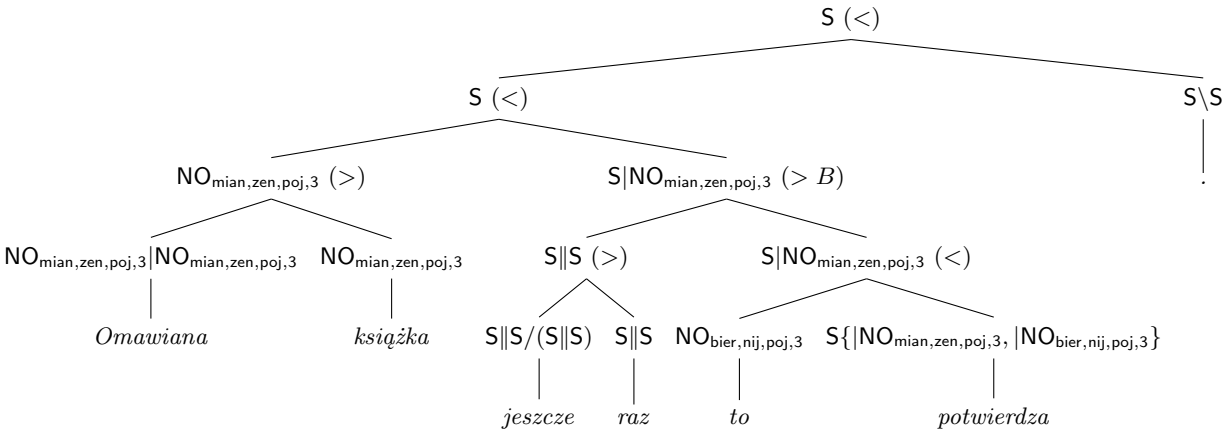


Figure 7.7: PL-CCG derivation tree for the sentence *Omawiana książka jeszcze raz to potwierdza.*

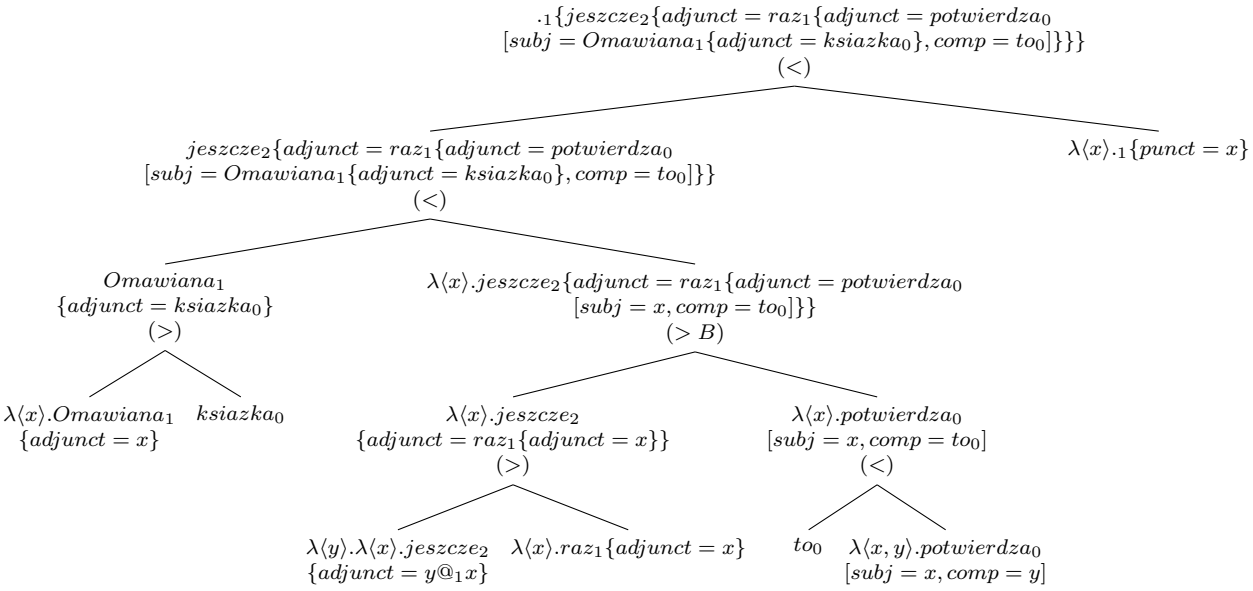


Figure 7.8: Lambda-terms in the example derivation.

Chapter 8

Evaluation

In this chapter, we attempt to evaluate the main result of the previously described algorithm, that is, the lexicon of word categories.

We frame the evaluation as a machine learning problem of training a parser. The derivation trees is divided into training and testing sets, and the lexicon extracted from the training set is used to parse the testing set. We use the dependencies produced by the parser to measure its accuracy. Unfortunately, the data sparseness causes the parser to succeed on only a small percentage of sentences (7% in the best case), but the obtained results still allow us to compare different approaches to conversion.

The chapter ends with a discussion of the lexicon. Examining the categories produced by the algorithm allows us to identify the areas where the algorithm was adequate to the task, and ones where a better conversion method is needed.

8.1. Versions of the lexicon

We decided to evaluate two conversion flags, creating four versions of the lexicon. The flags are

- "no-sets": no argument multisets, arguments in every category are ordered,
- "no-mods": no modifiers, every constituent is an argument.

The versions of the "normal" (multisets and modifiers present), "no-sets", "no-mods", and "no-mods+no-sets".

8.2. Quantitative evaluation

We now present the quantitative evaluation of the lexicon. We begin with a description of the parsing algorithm and methodology used, then show the numerical results and attempt to interpret them.

```
function parse(lexicon, w_1, ..., w_n) returns set of terms

  chart ← ∅
  for each w_i do
    for (w_i, t) ∈ lexicon do
      add (i, i + 1, t) to chart

  for k ∈ [2..n] do
    for i ∈ [1..n + 1 − k] do // try to build edges (i, i + k)
      for j ∈ [i + 1..i + k − 1] do
        for (i, j, t_1) ∈ chart do
          for (j, k, t_2) ∈ chart do // check all edge pairs (i, j), (j, k)

            // try combining t_1, t_2 using application or composition
            for t ∈ combine(t_1, t_2) do
              if (i, i + k, t) ∉ chart then
                add (i, i + k, t) to chart

  return {t: (1, n + 1, t) ∈ chart}
```

Figure 8.1: The PL-CCG parsing algorithm.

8.2.1. Parser

Our PL-CCG parser is using a variant of the non-probabilistic Cocke–Younger–Kasami (CYK) algorithm [Jurafsky09, chap. 13]. In the process of building a derivation, we maintain a collection of edges, called a chart. An edge is a pair (i, j, t), representing a (typed) term t derived form the words i, $i + 1$, ..., $j - 1$. The pseudocode for the algorithm is shown in figure 8.1.

The parser does not produce all possible derivations, however, it returns all terms that can be derived for the whole sentence. In practice, we are concerned only with the terms of type S from that final set.

The original CYK algorithm has a time complexity of $O(n^3 \cdot |G|)$, where $|G|$ is the size of grammar. In our case, the size factor is the number of possible terms for a sentence fragment. We can approximate it by the number of possible dependent-to-head mappings (n^2), giving us a worst case complexity of $O(n^5)$.

In practice, due to data sparseness the number of edges for a given word span remains in single figures most of the time. However, for any kind of wide coverage PL-CCG parsing, not to mention disambiguation between possible parses, a more efficient method (such as a statistical parser with beam seach, used in [Hockenmaier03]) is necessary.

8.2.2. Methodology

To maximize the coverage, we employ the *leave-one-out cross-validation* method. We use the whole corpus of PL-CCG derivation trees except for one as a training set, from

which we collect a lexicon (a mapping from words to typed terms). We then attempt to parse the remaining sentence. The procedure is repeated for every sentence in the corpus.

The end result of parsing the sentence is a set of dependency structures (one for each successful derivation). We combine them all into a set of retrieved dependencies R, which we compare to the set of dependencies O taken from an original dependency structure of the sentence[1]. To do that, we use the *precision* and *recall* measures:

$$precision = \frac{|R \cap O|}{|R|} \qquad recall = \frac{|R \cap O|}{|O|}$$

Precision and recall are calculated both for *labeled* and *unlabeled* dependencies, i.e. taking the dependency labels into account, and wiping them before comparision. The resulting measures are called LP, LR, UP and UR.

8.2.3. Results

The results are shown below. The precision and recall are averaged over all successfully parsed sentences.

Version	Coverage	LP	LR	UP	UR
nm+ns	215/8227 (2.61%)	97.08%	98.09%	98.45%	98.82%
no-mods	220/8227 (2.67%)	96.46%	97.87%	98.12%	98.48%
no-sets	539/8227 (6.55%)	86.78%	96.56%	91.36%	98.46%
normal	588/8227 (7.15%)	85.13%	96.32%	90.11%	98.36%

The very low coverage – 7% of sentences in the best case – is a result of significant data sparseness. Because the formalism is lexicalized, and a rich set of features increases the number of categories, often the parser does not find any category for an encountered word, or finds an unusable one.

Unfortunately, the low coverage makes the result hard to compare the numbers to the model for English CCG described in [Hockenmaier03], which achieves almost 100% coverage[2] (2395 out of 2401 sentences). It is however still possible to compare the performance of different versions of the conversion algorithm.

Surprisingly, the inclusion of argument multisets does relatively little to help parsing. The "not-sets" version of the lexicon still includes non-directional slashes (|) and that seems to be enough to parse most of the sentences the normal version captures.

[1] The procedure may seem suspect, because the original dependencies are also produced by our conversion procedure. However, the dependency extraction method is independent from the rest of the conversion, with the depndencies determined only by the source nodes, their types, and head element annotations. They are always the same regardless of conversion flags.

[2] The high coverage achieved by Hockenmaier can be attributed to a simple maneuver – all rare words (≤ 5 appearances in the corpus) have been replaced with their part-of-speech tag.

This solution is impractical in our case because the resulting grammar is too big to be efficiently used by our parser. Developing a probabilistic model for PL-CCG, and an efficient probabilistic parser, could make this method of increasing coverage feasible.

Using the modifiers in the lexicon has a much larger impact – the resulting categories are more general, which decreases the precision and recall scores, but we are able to parse over twice the number of sentences.

8.3. Discussion of the lexicon

We now examine the specific categories included in the lexicon. The Polish treebank is more complex than the English one, and its conversion is by no means complete. This, apart from the data sparseness, might be a reason for the low parsing coverage.

We first look at the normal version of the corpus, then briefly compare it with the other variants.

8.3.1. The normal version

The corpus contains 83571 tokens, for which the algorithm found 8055 distinct categories, and 1207 generalized categories (distinct categories after ignoring the feature vectors).

Table 8.1 We list the top 20 generalized categories, sorted by their occurences in the corpus, along with example occurences from the corpus. The words of a given type are underlined, their arguments enclosed in brackets. The explanation of the category symbols can be found in the previous in Table 7.1 in the previous chapter.

The function of these categories in the grammar is clear, however, there are some apparent problems:

- In compound noun phrases (*etap konkursu*) the head has a functional category (*etap* : NO/NO), but the dependent is arguably optional (at least from the syntactic point of view). In cases like this, we probably should classify the dependents as adjuncts and assign them modifier categories.
- Prepositions have at least three different categories depending on the use of the overall prepositional phrase (S‖S/NO, PM/NO, NO|NO/NO). The problem with prepositional phrases and other modifiers is mentioned in the section 4.6, along with possible solutions.

A closer look at infrequent categories reveals further issues:

- Argument verbal phrases with infinitives (*zaczyna z wolna zanikać*) are marked as WE, not S. However, the same modifiers apply to both, creating yet another modifier category (*z wolna* : WE‖WE).
- The same problem exists with some kinds of dependent clauses. In some cases the SS (sentential phrase) category just "wraps" a S (as is the case with the *że* connective shown above), but in others it SS functions as a sentence on its own – or worse, the result category includes the accompanying comma, so that the root verb of the dependent clause has type e.g. SS\','{|NO, |NO}. The problem could be solved with a more fine-grained treatment of these cases. Alternatively, the system

No.	# occ.	% occ.	Category	Explanation and example
1.	21071	25.21%	NO	noun phrases *etap konkursu*
2.	8062	9.65%	S\S	sentence-ending punctuation
3.	7171	8.58%	NO|NO	noun modifiers *[służbę] wojskową*
4.	5131	6.14%	NO/NO	nominal phrase head *etap [konkursu]*
5.	3171	3.79%	','	comma
6.	2905	3.48%	S‖S	sentence modifiers *Omawiana książka jeszcze raz to potwierdza.*
7.	2898	3.47%	PM/NO	prepositions (in non-modifier phrases) *Z [domu] wyniosłam zamiłowanie do pracy.*
8.	2590	3.10%	S‖S/NO	prepositions (in sentence modifiers) *Nie znali się za [życia].*
9.	1925	2.30%	NO|NO/NO	prepositions (in noun modifiers) *zmiany w [rządzie]*
10.	1867	2.23%	S|NO	verbs with no complement *odbywa [służbę wojskową]*
11.	1710	2.05%	'się'	reflexive pronoun *zakończył się*
12.	1630	1.95%	S{|NO, |NO}	verbs with one complement *[Omawiana książka] jeszcze raz [to] potwierdza.*
13.	1198	1.43%	S/S	sentence-beginning punctuation (–) *– [Co to jest?]*
14.	1155	1.38%	'nie'	negation particle *Nie znali się za życia.*
15.	1096	1.31%	PT	non-modifier adjective *Pokora uczyniła go cichym*
16.	895	1.07%	S\S/S	connectives in coordinate clauses *[Powrócił do wiary] ,* *[postanowił wrócić do Kościoła katolickiego].*
17.	811	0.97%	S{|NO, |PM}	verbs expecting a prepositional phrase *[ja] [w jego obronie] zawsze staję*
18.	645	0.77%	SS\','/S	*że* connective for dependent clauses *Poseł Włodzimierz Wasiński sądzi [,] że [. . .]*
19.	598	0.72%	NO\NO/NO	conjunctions for nominal coordinate phrases *[Szwecji] i [RPA]*
20.	525	0.63%	S	standalone verbs *– Dziękuję.*

Table 8.1: 20 generalized (no features) categories most frequently occuring in the lexicon.

could include a hierarchy of different S categories for different types of sentential phrases, encompassing WE and SS.

- Dependent clauses with the *który* relative pronoun (*rywala, który grał mecz* – the rival who played a match) are represented by the source trees with *który* as a pronoun subject; and our algorithm translates them as such. The requirement that *który* has to be the first element of the phrase is lost in the conversion.
  However, the source features identify the relative pronoun, so a more sophisticated translation algorithm is possible – we could, for instance, make *który* the main, type-raised functor of the phrase, taking the comma and the sentence as arguments: *który* : (NO|NO)\',’/(S|NO).
- The negated forms of verbs simply take the negation particle as its argument (in *[nie] znałem kościoła* – I didn't know the church – the verb *znałem* has category S|NO\'nie'). This is an efficient way of dealing with the fact that the negated forms expect a complement in a different case (non-negated *znałem* : $\mathsf{S|NO_{acc}}$; negated *nie znałem* : $\mathsf{S|NO_{gen}}$). A more general treatment would make *nie* a modifier; however, a stronger formalism would then be necessary to account for the case change.
- Some of the low-level (verb form, noun form, etc.) nodes still consist of multiple terminals. This doesn't pose a problem with verb suffixes – arguably suffixes like *-em* in *znałem* can have their own separate categories – but also results in a very inelegant treatment of e.g. proper names (in *Joliot-Curie*, the word *Joliot* has category NO/'curie'/'-'.
  In cases like this, the algorithm should probably convert the terminal symbols directly using the part-of-speech information (`tag`) included with them.
- Translation of long enumerations (*heblami, wiertłami, młotkami, nożami, dłutami*) does not retain the regularity of source data. The example phrase is represented in the source data as a "flat" subtree with the the last comma as head. The last comma is therefore applied to all the other nouns and commas, and receives the category (NO\NO)\NO\',’\NO\',’\NO\',’\NO/NO (the type of the phrase is NO\NO). An accurate conversion could binarize the tree first.

While the conversion seems to be adequate for the simple cases, for more complicated constructions the algorithm often gives a clearly suboptimal answer. To improve the conversion, the specifics of the source tree format need to be taken into account to greater extent.

The number of special cases suggests that looking at the source data produced by the *Świgra* grammar might not be the best approach. Perhaps it would be better to move a level higher and use the *grammar* itself, instead of its output, as a source of linguistic insight. The conversion would then be based not only on the end result, but also on rules applied by *Świgra* while parsing the sentence.

8.3.2. Other versions

We now try to determine the impact of two aspects of the conversion – modifiers and argument multisets – on the shape of the lexicon lexicon. Using different versions of the conversion algorithm resulted in the following sizes of category sets:

Version	**Categories**	**Categories (generalized)**
nm+ns	13231	3821
no-mods	12662	3202
no-sets	9735	1546
normal	8055	1207

The "no-mods" flag

As could be expected from the previously shown performance evaluation, using modifiers in the lexicon greatly reduces the number of used categories. The "no-mods" lexicon has the advantage of uniform treatment for e.g. prepositional phrases (all get the category PM) and adverbs (PS), but at the same time, the constituents need to be included as arguments of every head element that uses them. The result is a proliferation of categories like S/'.'{|NO, |PM, |PM}, S/'.'{|NO, |PM, |PS}, etc. Because "no-mods" treats interjections as arguments (giving them the category L), they need to be included in the verb categories as well.

The "no-sets" flag

The "no-sets" flag turns off the order relaxation described in the previous chapter. A noticeable result of that is the loss of generality of sentence modifiers – they no longer all receive the S‖S category, but instead have types like S|S, S|NO|NO|(S|NO|NO), S|NO|SS|(S|NO|SS) etc.

However, the numbers suggest that the effect is not as drastic as it may seem – using the order relaxation does not reduce the amount of categories as much, and as we have seen in the previous section, the impact on performance is also smaller than in the case of modifiers.

While the notion of multisets is attractive from the linguistic standpoint, perhaps including them in the formalism is not necessary for practical applications. The aforementioned problem with modifier categories could be solved with *higher-order composition*, allowing modifiers to act results hidden under two or more arguments. An example of second-order composition rule is:

$$(B^2) \quad X/Y,\ Y/Z/W \vdash X/Z/W$$

Higher-order composition could allow sentence modifiers like S|S to compose with categories such as S|NO|NO. The negative effects of introducing composition (especially higher-order) to the derivation rules could be again mitigated by restricting it to a special kind of categories.

Chapter 9

Conclusion

We have developed a categorial grammar based on CCG, capable of capturing the basic properties of Polish language. The formalism includes syntactic derivation system, semantics based on modified lambda calculus, and a content representation language that allows for extracting syntactic dependencies from the derivation.

The developed formalism allowed us to create a translation method for constituency trees and apply it to the Polish treebank *Składnica*. Unfortunately, we haven't been able to evaluate the resulting lexicon in a way that makes it possible to compare the numbers with other similar projects. However, we have shown that the proposed extensions of CCG are useful, both by reducing the numbers of categories, and by increasing the coverage of the parser. We have also been able to pinpoint some instances where the conversion still needs improvement.

The work presented here can be extended in a number of ways. We briefly recapitulate the possibilities of follow-up we mentioned at various points in the dissertation:

- A generative, or discriminative, probabilistic model of CCG derivations would make it possible to create an effective wide-coverage PL-CCG parser and compare it to other state-of-the-art parsers.
- The conversion method, and the category system, could be improved to cover a number of corner cases more adequately.
- A different treatment of modifier categories (see section 4.6) could also improve the systems's linguistic adequacy and performance.
- Because the performance increases attributed to argument multisets are relatively small, possible advantages of a "no-sets" PL-CCG could be explored.
- Finally, it's possible that a different approach to building a CCG treebank, based on closer integration with *Świgra* grammar, could yield better results.

Appendix A

List of features used by the Polish treebank

The following is a list of all feature values in the Polish treebank and their English translations, copied from the XML data file from [Sklad].

cat	category
wypowiedzenie	utterance
zdanie	clause
ff	finite phrase
fw	required phrase
fl	free phrase
flicz	numeral phrase
fno	nominal phrase
fpm	prepositional-nominal phrase
fpmpt	prepositional-adjectival phrase
fps	adverbial phrase
fpt	adjectival phrase
fwe	verbal phrase
fzd	sentential phrase
formaczas	verb form
formalicz	numeral form
formaprzym	adjectival form
formaprzys	adverbial form
formarzecz	noun form
partykuła	particle
przyimek	preposition
spójnik	conjunction/complementizer
zaimos	personal pronoun
zaimprzym	adjectival pronoun
zaimrzecz	nominal pronoun
posiłk	auxiliary

przyzlo	future auxiliary
condaglt	conditional auxiliary
korelat	correlate
modpart	particle modifier
modjaki	jaki-type modifier
znakkonca	final punctuation
przec	comma
nawias	bracket
cudz	quotation mark
pauza	dash
liczba	number
poj	singular
mno	plural
przypadek	case
mian	nominative
dop	genitive
cel	dative
bier	accusative
narz	instrumental
miej	locative
wol	vocative
pop	post-prepositional
rodzaj	gender
mos	masculine personal
mzw	masculine animal
mnż	masculine inanimate
m	masculine
żeń	feminine
nij	neuter
nmo	not masculine personal
stopien	degree
row	positive
wyz	comparative
naj	superlative
aspekt	aspect
nd	imperfective
dk	perfective
czas	tense
przy	future
ter	present
prze	past
osoba	person
1	first person
2	second person
3	third person
akom	accomodation
uzg	agreement numeral

nuzg	non-agreement numeral
neg	negation
tak	affirmative
nie(_)	negated
ani	«ani» negated
tryb	mood
ozn	indicative
roz	imperative
war	conditional
rekcja	case government
tfw	type of required phrase
subj	subj
adjp(mian)	adjp(mian)
adjp(narz)	adjp(narz)
advp	advp
infp(dk)	infp(dk)
infp(nd)	infp(nd)
np(bier)	np(bier)
np(cel)	np(cel)
np(dop)	np(dop)
np(mian)	np(mian)
np(narz)	np(narz)
prepadjp(do, dop)	prepadjp(do, dop)
prepadjp(na, bier)	prepadjp(na, bier)
prepadjp(za, bier)	prepadjp(za, bier)
prepnp(bez, dop)	prepnp(bez, dop)
prepnp(co do, dop)	prepnp(co do, dop)
prepnp(dla, dop)	prepnp(dla, dop)
prepnp(do, dop)	prepnp(do, dop)
prepnp(na, bier)	prepnp(na, bier)
prepnp(na, miej)	prepnp(na, miej)
prepnp(nad, narz)	prepnp(nad, narz)
prepnp(o, bier)	prepnp(o, bier)
prepnp(o, miej)	prepnp(o, miej)
prepnp(od, dop)	prepnp(od, dop)
prepnp(po, bier)	prepnp(po, bier)
prepnp(po, miej)	prepnp(po, miej)
prepnp(pod, bier)	prepnp(pod, bier)
prepnp(pod, narz)	prepnp(pod, narz)
prepnp(przeciwko, cel)	prepnp(przeciwko, cel)
prepnp(przed, narz)	prepnp(przed, narz)
prepnp(przez, bier)	prepnp(przez, bier)
prepnp(przy, miej)	prepnp(przy, miej)
prepnp(spośród, dop)	prepnp(spośród, dop)

prepnp(u, dop)	prepnp(u, dop)
prepnp(w, bier)	prepnp(w, bier)
prepnp(w, miej)	prepnp(w, miej)
prepnp(z, dop)	prepnp(z, dop)
prepnp(z, narz)	prepnp(z, narz)
prepnp(za, bier)	prepnp(za, bier)
prepnp(za, narz)	prepnp(za, narz)
sentp(że)	sentp(że)
sentp(żeby)	sentp(żeby)
sentp(do, dop, żeby)	sentp(do, dop, żeby)
sentp(jakby)	sentp(jakby)
sentp(o, bier, żeby)	sentp(o, bier, żeby)
sentp(o, miej, że)	sentp(o, miej, że)
sentp(o, miej, pz)	sentp(o, miej, pz)
sentp(pz)	sentp(pz)
sie	sie
dest	
tfz	type of sentential phrase
wyroznik	
os	finite personal
bos	impersonal
bok	inifinitive
psw	present participle
psu	past participle
poz	position
pre	pre
post	post
ink	incorporation
ni	no embedded conjunction
i(p, więc)	«więc» embedded
i(p, bowiem)	«bowiem» embedded
i(r, natomiast)	«natomiast» embedded
i(r, zaś)	«zaś» embedded
zap	
bzap	bzap
tk	tk
ozn	
typn	
z	z
o	o
typc	
z	z
o	o
kor	
nk	no correlate
do/dop	do/dop
o/bier	o/bier

o/miej	o/miej
typ	
rc	rc
rl	rl
rp	rp
sz	sz
szk	szk
po	po
pc	pc
pp	pp
i(p, bowiem)	i(p, bowiem)
i(p, więc)	i(p, więc)
i(r, natomiast)	i(r, natomiast)
i(r, zaś)	i(r, zaś)
do/dop	do/dop
o/bier	o/bier
o/miej	o/miej
przyim	
klasa	
kszt	
o	o
orth	
base	
tag	

Bibliography

[Baldridge03] Jason Baldridge and Geert–Jan Kruijff, *Multi-Modal Combinatory Categorial Grammar*, in: Proceedings of EACL, 2003.

[Jaworski11] Wojciech Jaworski, *Logical Categorial Grammar for Polish Language*, to appear, 2012.

[Hockenmaier03] Julia Hockenmaier, *Data and Models for Statistical Parsing with Combinatory Categorial Grammar*, PhD thesis, University of Edinburgh, 2003.

[Hoffman95] Beryl Hoffman, *The Computational Analysis of the Syntax and Interpretation of "Free" Word Order in Turkish*, PhD thesis, University of Pennsylvania, 1995.

[Jurafsky09] Daniel Jurafsky, James H. Martin, *Speech and Language Processing: An Introduction to Natural Language Processing, Speech Recognition, and Computational Linguistics*, 2nd edition, Prentice-Hall, 2009.

[Lambek58] Joachim Lambek, *The mathematics of sentence structure*, in: *The American Mathematocal Monthly*, Vol. 65, No. 3. (March 1958), pp. 154—170.

[Przepiórkowski03] Adam Przepiórkowski, *A Hierarchy of Polish Genders*, in: Bański and Przepiórkowski, eds., *Generative Linguistics in Poland: Morphosyntactic Investigations*, 109–122.

[Sklad] Linguistic Engineering Group, Department of Artificial Intelligence at the Institute of Computer Science, Polish Academy of Sciences, *Składnica – bank drzew składniowych* (website), `http://zil.ipipan.waw.pl/Sk%C5%82adnica/`.

[Steedman11] Mark Steedman and Jason Baldridge, *Combinatory Categorial Grammar*, draft (2011), to appear in: R. Borsley and K. Borjars (eds.), *Non-Transformational Syntax*, Blackwell, 181–224.

[Swidzinski92] Marek Świdziński, *Gramatyka formalna języka polskiego*, Rozprawy Uniwersytetu Warszawskiego, Wydawnictwa Uniwersytetu Warszawskiego, 1992.

[Urzyczyn03] Morten Heine Sørensen, Paweł Urzyczyn, *Lectures on the Curry-Howard isomorphism*, Elsevier, 2006.

[Wolinski04] Marcin Woliński, *Komputerowa weryfikacja gramatyki Świdzińskiego*, PhD thesis, IPI PAN, 2004.

[Wroblewska11] Alina Wróblewska, *Polish dependency bank*, in: *Linguistic Issues in Language Technology*, 7(1), 2012.