UTRECHT UNIVERSITY

BACHELOR THESIS

Bottom-up parsing for the extended typelogical grammars

Author:
Jaap JUMELET
4129962

Supervisor:
Prof. dr. Michael MOORTGAT

A 15 ECTS thesis submitted in fulfillment of the requirements for the degree of Bachelor of Science

in the

School of Philosophy and Artificial Intelligence
Faculty of Humanities

July 11, 2017

Contents

Introduction

Discovering the universal and underlying structures of natural languages has concerned linguists for ages. The earliest written record of these endeavours dates back as far as the 4th century BCE, when the linguist Pāṇini formulated a formal description of the Sanskrit language. Since then, the field of linguistics has developed into a broad scientific field. This thesis will focus mainly on the field of **formal language theory**, that studies the syntactic aspects of language (the *form*) and the relation of syntax to the semantics of language (the *meaning*).

Computational linguistics studies language from a computational perspective, and is an important sub-field of artificial intelligence. Processing language can be done from a *statistical* or *rule-based* point of view, or a combination of these two approaches. Formal language theory is mostly rule-based. The mathematician Richard Montague formulated the connection between natural languages and formal, mathematically defined languages as follows:

> "*There is in my opinion no important theoretical difference between natural languages and the artificial languages of logicians; indeed I consider it possible to comprehend the syntax and semantics of both kinds of languages with a single natural and mathematically precise theory.*"

In order to interpret a language it must be parsed. **Parsing** is the process of analysing a string, given a formal set of rules (the *grammar*). By parsing a sentence, a parse tree is obtained that provides a hierarchical structure of that sentence. It is desirable that this process does not take too long, relative to the size of the input string. A feasible parsing method for a natural language must therefore be able to parse its input in polynomial time (as opposed to exponential time).

This thesis focuses on the **typelogical grammars**, that provide a logical approach to the composition of natural language. The typelogical framework defines a grammar as a logical system. Words are now assigned a formula, and inferences are made on the basis of these formulas. Parsing a sentence then becomes analogous to a logical deduction in our system. A semantic interpretation is derived following the same inference steps that are made on a syntactic level, which allows us to interpret a sentence in a compositional manner.

The structure of this thesis is as follows. In the first chapter a theoretical background is presented, that places the typelogical grammars and its extensions in the context of formal language theory. An overview of different parsing techniques is provided, as well as a notion of a formal definition of a parsing procedure. Based on the concepts introduced in the first two chapters, we establish a parsing method for a typelogical formalism called the Lambek-Grishin calculus. It is yet unknown whether this system can be parsed in polynomial time. The procedure that is presented here might give more insight into the time complexity of the calculus. In the final chapter a logic programming implementation of the parsing system is presented. The code for this implementation can be found in the appendix.

Chapter 1

From rules to types

According to Chomsky (1957), a grammar for a language $\mathcal{L}$ is a formal device that generates all of the grammatical sentences of $\mathcal{L}$ and none of the ungrammatical ones. A language itself is then defined as the (finite or infinite) set of sentences that can be derived by a set of production rules (the *grammar*), given a finite set of words or symbols (an *alphabet*). Chomsky (1956) states that one of the linguist's primary concerns is to derive a properly formulated grammar describing the set of grammatical sentences for a particular language, given a finite number of observed sentences.

1.1 Rewriting grammars

In (Chomsky, 1959) the Chomsky hierarchy is described, a system that classifies formal languages based on their expressivity and complexity. This section will give an overview of several classes of the Chomsky hierarchy. Most importantly, it will show the strengths and weaknesses of each class in relation to natural languages. The grammars that are addressed in this section are all **rewriting systems**: a production rule of the form $\alpha \rightarrow \beta$ defines how a string of one or more terminal and non-terminal symbols (α) can be rewritten into a different string of terminal and non-terminal symbols (β). A grammar formalism imposes certain restrictions on the form of such a production rule. A terminal symbol is an elementary symbol of a language, whereas a non-terminal can be rewritten according to a production rule. A more in-depth overview can be found in (Kallmeyer, 2010) and (Hopcroft and Ullman, 1979).

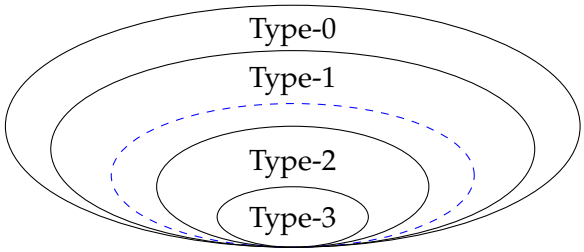


FIGURE 1.1: **The Chomsky hierarchy**. The dashed ellipse denotes the set of *mildly context-sensitive* languages.

1.1.1 Type-3: Regular Languages

The set of regular languages is the set of languages that are accepted by a Finite State Automaton (FSA). Informally, an FSA can be described as a finite set of states that are connected by a transition function. The transition from one state to another (or

to itself) is then determined by a given input string. If a final state is reached after reading all the input symbols and starting at a start state, the input string is accepted.

Chomsky (1956) shows that regular languages lack the expressivity to handle certain syntactic patterns that are found in natural languages. For example, let S_1 and S_2 be declarative sentences. Then the following pattern is a grammatical English sentence: *If* S_1, *then* S_2. Because the sentences S_1 and S_2 can be of the form of this if-else pattern too, an FSA would be unable to account for the dependency between *If* S_1 and *then* S_2. Bounding the depth of such a recursive pattern would avoid this shortcoming, but this would not offer a complete solution. We are, after all, trying to establish a formal method that is able to express more complex patterns that are found in natural languages. This does not imply that regular languages have no use in natural language processing: morphology, for example, is a field of linguistics in which regular languages play an important role.

1.1.2 Type-2: Context-free languages

As regular languages are unable to express certain phenomena that are found in natural languages, we move on to a larger class of languages called the context-free languages. This is the set of languages that can be described by a Context-Free Grammar (CFG). A CFG consists of a finite set of productions (or rewriting rules) of the form $A \rightarrow \alpha$, where A is a non-terminal symbol and α a (possibly empty) sequence of terminal and non-terminal symbols. A lot of patterns that are found in natural languages can be described by CFGs, due to the fact that the production rules permit center embedded recursion. Center embedding occurs when a production rule is of the form $A \rightarrow \alpha A \beta$, where α and β are non-empty strings of terminal and non-terminal symbols. The non-terminal A is thus *embedded* into the middle of its own production rule. For example, the *if* S_1 *then* S_2 pattern that we saw before could be described as the following CFG production rule: $S \rightarrow$ *if* S *then* S, where S denotes a production rule for a declarative sentence.

1.1.3 Type-1: Context-sensitive languages

Since the 1980s it has been known that CFGs are not powerful enough to describe all phenomena that are found in natural languages (Kallmeyer, 2010). Take, for instance, the following Swiss German example, containing a cross-serial dependency (the colours mark the dependencies between the verbs and the noun phrases):

. . . das mer	d'chind	em Hans	es huus	lönd	hälfe	aastriiche
. . . that we	the children$_{ACC}$	Hans$_{DAT}$	house$_{ACC}$	let	help	paint

'*. . . that we let the children help Hans paint the house*'

This cross-serial dependency can be described as the pattern $abcabc$. Iterating this pattern amounts to the copy language $\{ww \mid w \in \{a, b, c\}^+\}$, a language that is not context-free. This shows that the Swiss German language is not context-free, as proved by Shieber (1985). This was the first actual proof of the non-context-freeness of a natural language[1]. An explanation of this proof can be found in section 2.1.1 of (Kallmeyer, 2010).

[1] Although (Huybregts, 1984) showed that Dutch verb clusters contain cross-serial dependencies on the semantic level.

Context-sensitive languages are the next class in the Chomsky hierarchy: the set of languages that can be described by a Context-Sensitive Grammar (CSG). In a CSG production rules have the form $\alpha A\beta \rightarrow \alpha\gamma\beta$, where A is a non-terminal symbol, α and β are (possibly empty) sequences of terminal and non-terminal symbols, and γ is a non-empty sequence of terminal and non-terminal symbols. In other words, a production rule for a non-terminal A now takes its context into account, unlike a context-free production rule. CSGs are able to express patterns such as the cross-serial dependencies in Swiss German, but the expressive power of CSGs is shown to be larger than needed for natural languages. For instance, in (Hopcroft and Ullman, 1979) the language $\{a^{2^i} \mid i \geq 1\}$ is shown to be context-sensitive, but exponentially growing structures will not be found in any natural language.

1.1.4 Mildly context-sensitive languages

In 1985, Aravind Joshi introduced the notion of mild context-sensitivity, a property of a class of languages that is defined as follows in (Kallmeyer, 2010):

Definition 1.1.1 (Mildly context-sensitive)**.**

1. *A set $\mathcal{L}$ of languages is* mildly-context sensitive *iff*
   a) *$\mathcal{L}$ contains all context-free languages.*
   b) *$\mathcal{L}$ can describe cross-serial dependencies: There is an $n \geq 2$ such that $\{w^k \mid w \in T^*\} \in \mathcal{L}$ for all $k \leq n$.*
   c) *The languages in $\mathcal{L}$ are polynomially parsable, i.e., $\mathcal{L} \subset PTIME$.*
   d) *The languages in $\mathcal{L}$ have the* constant growth property*.*
2. *A formalism F is* mildly context-sensitive *iff the set $\{L|L = L(G)$ for some $G \in F\}$ is mildly context-sensitive.*

In other words, a mildly context-sensitive language $\mathcal{L}$ has to be an extension of all context-free languages. It must be able to describe patterns such as the Swiss German example. Parsing the language must take no longer than $O(n^k)$, where n is the input size and k some constant. And finally, if we would order the words of $\mathcal{L}$ according to their length, the length would grow in a linear way (the *constant growth property*).

Once Joshi formulated this notion of mild context-sensitivity, a wide range of grammar formalisms extending CFGs has been established. A clear overview can be found in (Kallmeyer, 2010). Among those formalisms are:

- **Linear Indexed Grammars (LIGs)**: LIGs are a subset of the Indexed Grammars (IGs). An IG looks like a CFG, but a stack of index symbols is attached to each non-terminal symbol. Productions have the form $A \rightarrow \alpha$, $A \rightarrow Bf$ or $Af \rightarrow \alpha$, where f is a string of index symbols. A production of $A \rightarrow \alpha$ copies the stack of A to each non-terminal in α. LIGs require a maximum of one non-terminal in α to receive the stack of A. This restriction makes LIGs belong to the mildly context-sensitive classes.
- **Multiple Context-Free Grammars (MCFGs)**: in MCFGs a non-terminal can yield multiple sequences of terminals that do not have to be concatenated adjacently to each other. This means that their span can be discontinuous in the input. A non-terminal of dimension k then derives k-tuples of terminal string by a yield function f. The class of MCFGs is a superset of the class of LIGs, but it still is a mildly context-sensitive formalism.

- **Range Concatenation Grammars (RCGs)**: RCGs are based on the yield functions of MCFGs, but these functions are added to the production rules themselves. RCGs are not mildly context-sensitive, but by imposing certain restrictions on the production rules the class of the *simple RCGs* is established. Simple RCGs are weakly equivalent to MCFGs.
- The **typelogical** and **Combinatory Categorial Grammars** that are addressed in the next section, and **Tree Adjoining Grammars** that are compared to our own system in section 3.4.

1.2 Extended Typelogical Grammars

We will now shift our focus to the Typelogical Grammars: substructural logics that are well-suited for a compositional approach of natural language processing. Syntactic constituents are now combined as functions and will be assigned a certain linguistic *category*. Contrary to the deductive system for CFGs, inferences are now made on the basis of the lexical items themselves. The formalisms that have been covered so far were all rewriting grammars: a production rule ($A \to \alpha$) defines how a non-terminal A can be rewritten to the string α. The inferences of such a production rule are independent of the non-terminal that is rewritten, i.e. the inference is solely made on the basis of the rule itself. CFGs, for example, can be defined as a deductive system, in which inferences are made by the *Cut rule* (Capelletti, 2007):

$$\frac{\Delta \to B \quad \Gamma[B] \to C}{\Gamma[\Delta] \to C} \tag{1.1}$$

Axioms have the form $\Gamma \to A$, if $\Gamma \to A$ is a production rule in the grammar. Note that Capelletti reverses the order of the production rules: $A \to \alpha$ becomes $\alpha \to A$.

1.2.1 The Lambek calculus

The typelogical grammars (TLGs) that this thesis focuses on are one of the two main varieties of the categorial grammars (Whitman, 2004), the other variety being the Combinatory Categorial Grammars that are addressed briefly in the following subsection. These typelogical grammars are based on (Lambek, 1958) and (Lambek, 1961), in which Lambek introduced the method of *parsing as deduction* in linguistics. Initially, there was not much interest for Lambek's systems: it took until the 1980's for the scientific community to develop them into a broader linguistic theory. This can be attributed to two factors (Moortgat, 2014). Firstly, van Benthem (1983) provided a clear homomorphic translation between typelogical derivations and computational semantics, based on the Curry-Howard correspondence. And secondly, the introduction of linear logic by Girard (1987) created a renewed interest in substructural logics.

In the Syntactic Calculus, or more simply the Lambek Calculus **L**, lexical items are assigned a logical formula (or type), instead of a traditional part of speech such as noun, adverb or determiner. In the Lambek systems, a type is specified by (1.2) in BNF, where p is an atomic type such as np or s. $A\backslash B$ is pronounced 'A under B' and B/A is pronounced 'B over A'. These types can best be understood as a type that expects an argument of type A to its left ($A\backslash B$) or to its right (B/A), to 'produce' a type B.

$$Types : \mathcal{A}, \mathcal{B} \ ::= p \mid \mathcal{A} \otimes \mathcal{B} \mid \mathcal{A}\backslash\mathcal{B} \mid \mathcal{A}/\mathcal{B} \tag{1.2}$$

The basic system **NL** is formed by two preorder laws (1.3: identity and transitivity) and four residuation principles (1.4). The transitivity rule is also called the cut rule. L is an extension of NL, by adding (global) associative properties to the $\otimes$ operator (1.5). Adding global commutativity (1.6) yields the calculus **NLP**, adding both commutativity and associativity yields **LP**.

$$\frac{}{A \to A}\ id \qquad ; \qquad \frac{A \to B \quad B \to C}{A \to C}\ trans \tag{1.3}$$

$$\dfrac{\dfrac{B \ \to \ A \setminus C}{A \otimes B \ \to \ C}}{A \ \to \ C \,/\, B} \tag{1.4}$$

$$(A \otimes B) \otimes C \leftrightarrow A \otimes (B \otimes C) \tag{1.5}$$

$$A \otimes B \leftrightarrow B \otimes A \tag{1.6}$$

1.2.2 Categorial Combinatory Grammars

As stated earlier, the **Combinatory Categorial Grammars (CCGs)** (Steedman, 2000) are the other main variety in the categorial framework. Similar to TLGs, inferences in CCGs are made on the basis of the lexical items themselves. CCGs are a **rule-based** approach, whereas TLGs are referred to as a **deductive** approach (Whitman, 2004). The inference rules in TLGs are defined as a logical system, with a focus on logical properties such as soundness and completeness. Inferences in CCGs are based on a finite set of rules, based on the concepts of combinatory logic. Among those rules are function application, type raising (or *lifting*), and function composition. It is possible to lie a restriction on a rule in a CCG to apply only to certain categories, a useful property of the system. This is not be possible in the classical typelogical systems. In the next section, it is shown that extending our system with the technique of multimodal control makes it possible to add structural control to our system that only applies to certain categories. CCGs are mildly context-sensitive, and can be parsed in polynomial time.

1.2.3 Extending the Lambek calculus

The systems L and NL recognize only the context-free languages, as was proved by Pentus (1997) and (Kandulski, 1988), respectively. We therefore need to extend these systems to provide a model with greater expressibility. Several extensions of the Lambek calculus have been proposed. An overview of these extensions can be found in (Moot, 2015). This thesis focuses on two different extensions:

1. The structural control modalities $\Diamond$ and $\Box$, that allow us to define specific *postulates* that only act on structures that consist of certain $\Diamond$ and $\Box$ configurations.

2. The Lambek-Grishin calculus in which the Lambek systems are extended symmetrically by an extra set of operators. The succedent side of the sequent can now consist of multiple conclusions. This allows for a concise integration of *continuation semantics* to the typelogical system.

The structural control modalities are part of the **multimodal Lambek calculus**, that allows us to define different structural rules for different modes by adding indices to each connective. This makes it possible to add *mixed associativity* and *mixed commutativity* to the calculus, thus restricting the occurrence of global associativity and commutativity.

Kurtonina and Moortgat (1997) propose the addition of a pair of unary connectives $\Diamond$ and $\Box$ with their own corresponding structural rules. Associativity and commutativity postulates can now be defined solely on structures that contain $\Diamond$ nodes. For example, one might add the following postulates to the calculus (which will be referred to as $P1$ and $P2$ later on in chapters 3 and 4). These postulates allow us to express phrases such as "*student that Alice likes*", in which the object *student* is disconnected from the object position of *likes*. This is a form of **extraction**. A derivation for a phrase that contains such an extraction is shown in figure 1.4 at the end of this chapter.

$$(A \otimes B) \otimes \Diamond C \xrightarrow{P1} A \otimes (B \otimes \Diamond C) \quad (A \otimes B) \otimes \Diamond C \xrightarrow{P2} (A \otimes \Diamond C) \otimes B$$

The logical residuation law that allow for the interaction between the two unary operators is defined as follows:

$$\frac{\Diamond A \rightarrow B}{A \rightarrow \Box B} \tag{1.7}$$

1.2.4 The Lambek-Grishin calculus

The other extension that this thesis focuses on is the Lambek-Grishin calculus (LG) (Moortgat, 2009). The sequents of the traditional Lambek systems we have seen so far do all have the form $A_1, ..., A_n \rightarrow B$, where $A_1, ..., A_n$ is a structure of formulas (a tree in the case of NL) and B is a single formula. In LG sequents are of the form $A_1, ..., A_n \rightarrow B_1, ..., B_m$. The motivation for this extension is that it allows us to express *evaluation contexts* on the succedent side of the sequent, in the form of continuations. This is achieved by enriching the product operation $\otimes$ and slash operations of NL with the dual coproduct $\oplus$ and the residual co-implications $\oslash$ and $\otimes$. LG extends the residual laws of (1.4) symmetrically, as is shown in (1.8). $A \oslash B$ is pronounced "A minus B", $B \otimes A$ "B from A". The Grishin connectives interact with the Lambek connectives by four linear *distributivity principles* (1.18). The fragment of LG without these principles is referred to as $\text{LG}_{\emptyset}$ and the fragment with them as LG_{IV}. LG without subscript usually refers to LG_{IV}. Section 1.3 provides an overview of the inference rules of LG.

$$\frac{\dfrac{C \oslash A \rightarrow B}{C \rightarrow B \oplus A}}{B \otimes C \rightarrow A} \tag{1.8}$$

1.2.5 Expressivity and complexity of typelogical grammars

As stated earlier, the systems L and NL are both context-free. Pentus (2006) shows that the global associativity of L makes its derivability problem NP-complete, whereas the derivability problem of NL is shown to be polynomial (e.g. in (Capelletti, 2007)). van Benthem (1995) shows that LP recognizes all permutation closures of context-free languages (π(CFL)). This is a class of languages that are all indexed languages (Brough, Ciobanu, and Elder, 2014), a class that offers greater expressivity than is needed for natural languages. LP is shown to be NP-complete (Kanovich, 1994). Moot (2002) showed that the multimodal Lambek calculus without restrictions on the structural rules is Turing complete, but that imposing certain restrictions let us obtain a fragment that recognizes the context-sensitive languages and that is PSPACE complete.

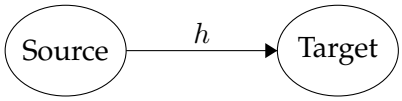
The exact recognizing capacity of LG is yet unknown. $LG_{\emptyset}$ (without interaction principles) is shown to be context-free (Bastenhof, 2010). Melissen (2009) provided a new lower bound for LG_{IV}, by showing that it recognizes all languages that are the intersection of a context-free language and the permutation closure of a context-free language. This entails that LG recognizes languages such as the language that contains any permutation of an equal number of a, b, and $c's$, or the language $a_1^n, ..., a_m^n$, where a_i is an alphabet symbol. Bransen (2012) proved that LG_{IV} is NP-complete. It is not clear yet whether LG_{IV} falls in the class of mildly context-sensitive languages.

Formalism	**Generative capacity**	**Complexity**
NL	CFL	P
L	CFL	NP-complete
LP	π(CFL)	NP-complete
$NL_{\Diamond\mathcal{R}}$	CSL	PSPACE
$LG_{\emptyset}$	CFL	P
LG_{IV}	Lower bound: CFL $\cap$ π(CFL) Upper bound: unknown	NP-complete

FIGURE 1.2: An overview of several typelogical formalisms. The upper bound of LG_{IV} is yet unknown.

1.2.6 The semantics of typelogical grammars

In the introduction of the Lambek Calculus it is stated that the computational semantics can be derived by the Curry-Howard (CH) correspondence, as introduced by van Benthem (1983). The interface between form and meaning is thereby based on the concept of **compositionality**. The CH correspondence provides a clear homomorphic mapping h from a *source language* (a syntactic system, such as the Lambek-Grishin calculus) towards a *target language* (the semantic interpretation of our syntax). A homomorphism is a mapping that is structure-preserving.



The source language can thus be interpreted in several ways, based on the homomorphism that is used to translate it. These interpretations can roughly be divided into a set theoretical approach and a distributional approach.

The set theoretical approach is based on the framework that was established by Montague (1970). Relations between phrases can be expressed using the concepts of set theory, and this approach provides a concise method to express phenomena such as quantification and scope ambiguity. In the case of interpreting fLG, a translation between fLG and a set theoretical interpretation might follow the scheme of figure 1.3, as described in (Moortgat and Moot, 2011).
Terms in Λ_{fLG} are the encoding of the logical steps of a derivation, i.e. the structural steps are ignored. The source logic is translated by the compositional interpretation $\lceil\cdot\rceil$, which sends the proof terms to a Natural Deduction proof in $\Lambda_{\mathrm{MILL}_{\otimes,\cdot\perp}}$. This is

$$fLG \xleftarrow{\text{CH}} \Lambda_{fLG} \xrightarrow{\lceil\cdot\rceil} \Lambda_{\text{MILL}_{\otimes,\cdot\perp}} \xrightarrow{(\cdot)^\ell} \Lambda_{IL_{\times,\to}}$$

FIGURE 1.3: Λ_{fLG} is a different presentation of the source logic fLG, obtained by the Curry-Howard correspondence. $\Lambda_{\text{MILL}_{\otimes,\cdot\perp}}$ and $\Lambda_{IL_{\times,\to}}$ are two target logics of fLG that are retrieved by a compositional translation.

the linear lambda calculus of $\text{MILL}_{\otimes,\cdot\perp}$ a fragment of the Multiplicative Intuitionistic Linear Logic. The lexical items in a MILL term are still represented as a constant. $(\cdot)^\ell$ provides the final translation, by substituting the constants for a lambda term that is defined in the lexicon.

This whole process is exemplified in (1.9), for the sentence *Bob likes some teacher*. The bias of np and n is positive, and s negative. In the definition for *some* and *likes* below, $\pi^i\alpha$ denotes a projection from the tuple α to its i^{th} element.

Bob	np	bob
teacher	n	teacher
some	np/n	$\lambda\alpha_0.(\exists\ \lambda x_2.((\wedge\ (\pi^1\alpha_0\ x_2))\ (\pi^0\alpha_0\ x_2)))$
likes	$(np\backslash s)/np$	$\lambda\alpha_0.(\pi^1\pi^0\alpha_0\ ((\text{LIKES}\ \pi^1\alpha_0)\ \pi^0\pi^0\alpha_0))$

$$\begin{aligned}
&\mu\alpha_0.\langle\ \mathsf{some}\uparrow(\widetilde{\mu}y_0.\langle\ \mathsf{likes}\uparrow((\mathsf{bob}\setminus\alpha_0)\ /\ y_0)\ \rangle\ /\ \mathsf{teacher})\ \rangle\\
&\lambda\alpha_0.(\lceil\mathsf{some}\rceil\ \langle\lambda y_0.(\lceil\mathsf{likes}\rceil\ \langle\langle\lceil\mathsf{bob}\rceil,\alpha_0\rangle,y_0\rangle),\lceil\mathsf{teacher}\rceil\rangle)\\
&\lambda\alpha_0.(\exists\ \lambda y_1.((\wedge\ (\text{TEACHER}\ y_1))\ (\alpha_0\ ((\text{LIKES}\ y_1)\ \text{BOB}))))
\end{aligned} \tag{1.9}$$

We stated above that the source logical could also be interpreted following a distributional approach. Current research focuses on a distributional interpretation of compositional models. Words can be described as vectors, and the compositional nature of the categorial grammars could provide a method how these vectors are to be combined. An introduction to these compositional distributional semantics can be found in (Clark, 2015). A distributional interpretation for the Lambek-Grishin calculus can be found in (Wijnholds, 2014).

1.3 Axiomatizing the Lambek-Grishin calculus

The residual laws of (1.4) and (1.8) result in the following expanding and contracting patterns. Note the symmetries between the Grishin connectives, and the original Lambek connectives.

$$\begin{aligned}
A\otimes(A\backslash B)\to B\to A\backslash(A\otimes B) \qquad (B/A)\otimes A\to B\to(B\otimes A)/A\\
(B\oplus A)\oslash A\to B\to(B\oslash A)\oplus A \qquad A\oslash(A\oplus B)\to B\to A\oplus(A\oslash B)
\end{aligned} \tag{1.10}$$

The expanding patterns of (1.10), in combination with the transitivity rule, will pose difficulties if we want to define a decision procedure for an inference $A\to B$. Lambek overcame this by recasting the Syntactic Calculus into a Gentzen-style sequent

calculus: causing the cut rule to become an admissible rule and the decision procedure to be achieved efficiently by backward chaining. Backward chaining is a proof search strategy that starts with a goal sequent, and tries to work back to the axiom premises following the logical inferences of a system.

Moortgat and Oehrle (1999) propose a system that makes the cut rule (transitivity) an *admissible rule* by adding the monotonicity rules of (1.11). This addition makes the calculus more appropriate for proof search, as is shown in section 4.6 of (Moot and Retoré, 2012).

$$\frac{A \to B \quad C \to D}{A \otimes C \to B \otimes D} \quad ; \quad \frac{A \to B \quad C \to D}{B \backslash C \to A \backslash B} \quad ; \quad \frac{A \to B \quad C \to D}{C / B \to D / A} \tag{1.11}$$

1.3.1 sLG

Moortgat and Moot (2011) introduce **sLG**, a presentation of LG that allows decidable proof search. This is done in the style of a Display Logic (Goré, 1998): for every logical connective, there is a structural counterpart. They show that sLG is equivalent to aLG (the arrow notation of LG). Sequents will be of the form $\mathcal{I} \vdash \mathcal{O}$, formed by the following grammar (with $\mathcal{F}$ being a logical formula). Note that the $\Diamond$ and $\Box$ connectives are an extension of fLG, as described in section 1.2.3.

$$\begin{aligned} \mathcal{I} ::= &\ \mathcal{F} \mid \mathcal{I} \cdot \otimes \cdot \mathcal{I} \mid \mathcal{I} \cdot \oslash \cdot \mathcal{O} \mid \mathcal{O} \cdot \otimes \cdot \mathcal{I} \mid \langle \mathcal{I} \rangle \\ \mathcal{O} ::= &\ \mathcal{F} \mid \mathcal{O} \cdot \oplus \cdot \mathcal{O} \mid \mathcal{I} \cdot \backslash \cdot \mathcal{O} \mid \mathcal{O} \cdot / \cdot \mathcal{I} \mid [\mathcal{O}] \end{aligned} \tag{1.12}$$

The rules of sLG can be divided into three groups:

1. The **identity** group: axiom and cut (1.13).
2. The **structural** group: the (dual) residuation rules (1.14), the postulates P1 and P2 (1.17), and the Distributivity Postulates (1.18).
3. The **logical** group: rewrite rules (1.15) and monotonicity principles (1.16).

Structural variables are denoted as $X, Y, \ldots$ and formula variables as $A, B, \ldots$ in the rules below. $[\cdot]$ and $\langle \cdot \rangle$ denote the structural counterparts of $\Box$ and $\Diamond$, respectively. The system allows cut elimination, i.e. the cut rule is an admissible rule. *The identity group.*

$$\frac{}{A \vdash A}\ Ax \qquad \left(\frac{X \vdash A \quad A \vdash Y}{X \vdash Y}\ Cut \right) \tag{1.13}$$

The residuation rules.

$$\frac{\dfrac{Y \vdash X \cdot \backslash \cdot Z}{X \cdot \otimes \cdot Y \vdash Z}\ rp}{X \vdash Z \cdot / \cdot Y}\ rp \qquad \frac{\dfrac{Z \cdot \oslash \cdot X \vdash Y}{Z \vdash Y \cdot \oplus \cdot X}\ drp}{Y \cdot \otimes \cdot Z \vdash X}\ drp \qquad \frac{\langle X \rangle \vdash Y}{X \vdash [Y]}\ rp \tag{1.14}$$

Rewrite rules: $\$ \in \{\otimes, \oslash, \otimes\}$, $\# \in \{\oplus, \backslash, /\}$.

$$\frac{A \cdot \$ \cdot B \vdash Y}{A\ \$\ B \vdash Y}\ \$L \qquad \frac{X \vdash A \cdot \# \cdot B}{X \vdash A\ \#\ B}\ \#R$$

$$\frac{X \vdash [A]}{X \vdash \Box A}\ \Box R \qquad \frac{\langle A \rangle \vdash Y}{\Diamond A \vdash Y}\ \Diamond L \tag{1.15}$$

The monotonicity rules for each operator.

$$\frac{X \vdash A \quad Y \vdash B}{X \cdot \otimes \cdot Y \vdash A \otimes B}\otimes R \qquad \frac{A \vdash X \quad B \vdash Y}{A \oplus B \vdash X \cdot \oplus \cdot Y}\oplus L$$

$$\frac{X \vdash A \quad B \vdash Y}{A \backslash B \vdash X \cdot \backslash \cdot Y}\backslash L \qquad \frac{X \vdash A \quad B \vdash Y}{X \cdot \oslash \cdot Y \vdash A \oslash B}\oslash R$$

$$\frac{X \vdash A \quad B \vdash Y}{B / A \vdash Y \cdot / \cdot X}/ L \qquad \frac{X \vdash A \quad B \vdash Y}{Y \cdot \otimes \cdot X \vdash B \otimes A}\otimes R$$

$$\frac{A \vdash Y}{\Box A \vdash [Y]}\Box L \qquad \frac{X \vdash A}{\langle X \rangle \vdash \Diamond A}\Diamond R \tag{1.16}$$

The structural extensions of our system.

$$\frac{A \cdot \otimes \cdot (B \cdot \otimes \cdot \Diamond C) \vdash X}{(A \cdot \otimes \cdot B) \cdot \otimes \cdot \Diamond C \vdash X}P1 \qquad \frac{(A \cdot \otimes \cdot \Diamond C) \cdot \otimes \cdot B \vdash X}{(A \cdot \otimes \cdot B) \cdot \otimes \cdot \Diamond C \vdash X}P2 \tag{1.17}$$

$$\frac{X \cdot \otimes \cdot Y \vdash Z \cdot \oplus \cdot W}{Z \cdot \otimes \cdot X \vdash W \cdot / \cdot Y}G1 \qquad \frac{X \cdot \otimes \cdot Y \vdash Z \cdot \oplus \cdot W}{Y \cdot \oslash \cdot W \vdash X \cdot \backslash \cdot Z}G3$$

$$\frac{X \cdot \otimes \cdot Y \vdash Z \cdot \oplus \cdot W}{Z \cdot \otimes \cdot Y \vdash X \cdot \backslash \cdot W}G2 \qquad \frac{X \cdot \otimes \cdot Y \vdash Z \cdot \oplus \cdot W}{X \cdot \oslash \cdot W \vdash Z \cdot / \cdot Y}G4 \tag{1.18}$$

1.3.2 fLG

Although we now have established a compact system that allows us to define a decision procedure, there still is an unwanted side-effect possible. Namely **spurious ambiguity**, a form of ambiguity where backward chaining proof search yields multiple derivations that are considered the same proof. Moortgat and Moot (2011) show that this can be overcome by introducing a focused version of sLG called **fLG**.

Focusing is a concept that has its roots in the Linear Logic, for example in (Andreoli, 1992). Sequents now have a focus: left: $\boxed{X} \vdash Y$, right: $X \vdash \boxed{Y}$ or neutral: $X \vdash Y$. The (dual) residuation rules (1.14), distributivity rules (1.18) and rewrite rules (1.15) still operate on neutral sequents, and can be taken over from sLG. The monotonicity and identity rules now involve focused sequents. (De)focusing a sequent is constrained by its polarity: the $(\otimes, \oslash, \otimes, \Diamond)$ connectives are assigned a *positive* polarity and the $(\oplus, /, \backslash, \Box)$ connectives a *negative* polarity. Atomic formulas have an arbitrary polarity *bias*. However, the configuration of this bias can have major implications on the running time. Section 4.3 of (Chaudhuri, Pfenning, and Price, 2008) shows an example of Fibonacci numbers in which a uniformly positive bias leads to an exponential running time, while a uniformly negative bias results in a linear running time.

Axiom, Co-axiom In the (Ax) case A has to be positive; in the (CoAx) case A must be negative.

$$\frac{}{A \vdash \boxed{A}}Ax \qquad \frac{}{\boxed{A} \vdash A}CoAx \tag{1.19}$$

Focusing, Defocusing For the rules in the left column A has to be negative; for those in the right column A must be positive.

$$\frac{\boxed{A} \vdash Y}{A \vdash Y} \leftharpoondown \qquad \frac{X \vdash \boxed{A}}{X \vdash A} \rightharpoondown$$
$$\frac{X \vdash A}{X \vdash \boxed{A}} \rightharpoondown \qquad \frac{A \vdash Y}{\boxed{A} \vdash Y} \leftharpoondown \tag{1.20}$$

Monotonicity rules The focus of a complex formula is shifted to its subpart(s).

$$\frac{X \vdash \boxed{A} \quad Y \vdash \boxed{B}}{X \cdot \otimes \cdot Y \vdash \boxed{A \otimes B}} \otimes R \qquad \frac{\boxed{A} \vdash X \quad \boxed{B} \vdash Y}{\boxed{A \oplus B} \vdash X \cdot \oplus \cdot Y} \oplus L$$
$$\frac{X \vdash \boxed{A} \quad \boxed{B} \vdash Y}{\boxed{A \backslash B} \vdash X \cdot \backslash \cdot Y} \backslash L \qquad \frac{X \vdash \boxed{A} \quad \boxed{B} \vdash Y}{X \cdot \oslash \cdot Y \vdash \boxed{A \oslash B}} \oslash R$$
$$\frac{X \vdash \boxed{A} \quad \boxed{B} \vdash Y}{\boxed{B/A} \vdash Y \cdot / \cdot X} /L \qquad \frac{X \vdash \boxed{A} \quad \boxed{B} \vdash Y}{Y \cdot \oslash \cdot X \vdash \boxed{B \oslash A}} \oslash R \tag{1.21}$$
$$\frac{\boxed{A} \vdash Y}{\boxed{\Box A} \vdash [Y]} \Box L \qquad \frac{X \vdash \boxed{A}}{\langle X \rangle \vdash \boxed{\Diamond A}} \Diamond R \tag{1.22}$$

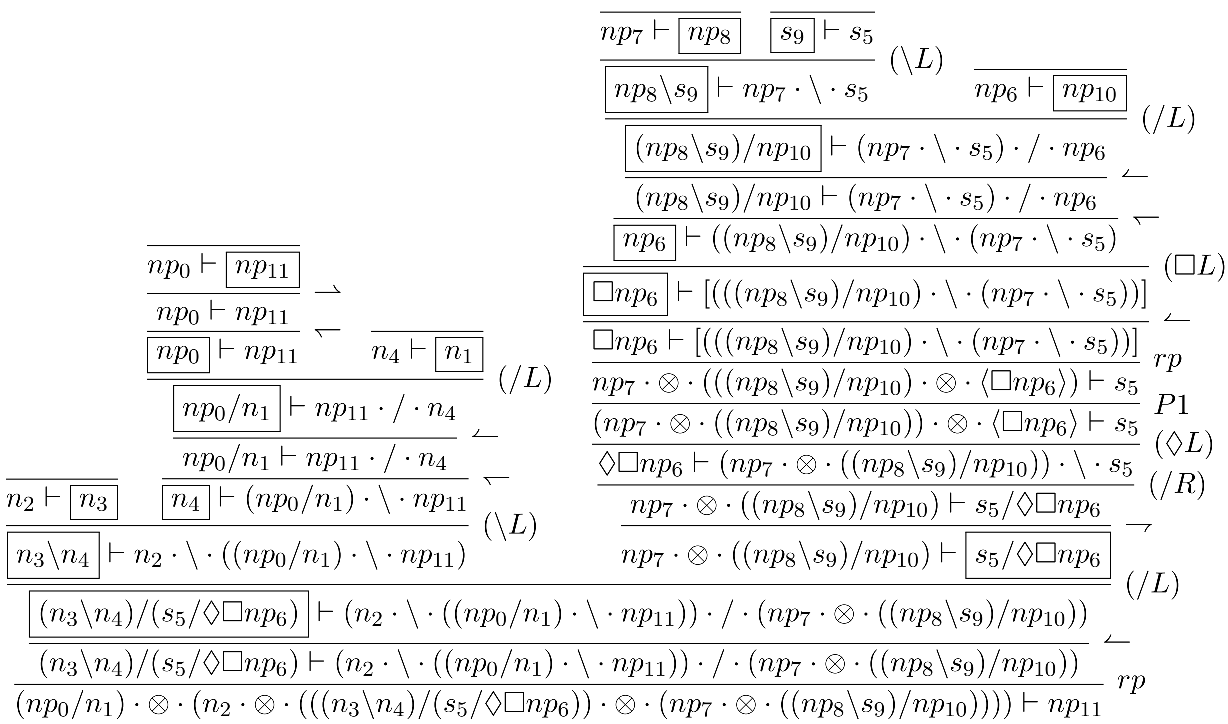


FIGURE 1.4: The proof tree for the phrase "*Some student that Alice likes*". Most of the structural residuation rules are omited, for the sake of brevity. Note the $P1$ rule halfway at the right branch of the proof tree, that inserts $\langle \Box np_6 \rangle$ next to the $(np_8 \backslash s_9)/np_{10}$ formula. np and n both have a positive bias, and s has a negative bias.

Chapter 2

Parsing as deduction for rewriting grammars

In the following chapters we will formulate a bottom-up parsing method for fLG. Because backward chaining expects its input to be structured, it does not actually *parse* its input: it solely *recognizes* it. Bottom-up parsing provides a way to correctly make a derivation for an unstructured input.

Parsing is the process of finding all **parse trees** that correspond to a given sentence and grammar. A parse tree is a hierarchical and complete description of the phrase structure of a sentence (Sikkel, 1993). Once such a parse tree is found for a sentence, we are able to *interpret* it, i.e. we can assign a corresponding meaning. Ambiguity manifests itself when multiple parse trees can be found for one sentence.

There are several approaches to parsing. For example, statistical parsing assigns a *probability* to a parse tree, while dependency parsing finds a *dependency structure* for a sentence (in contrary to a phrase structure). This thesis' approach focuses on the method of parsing as deduction: parsing algorithms are presented as deductive systems.

2.1 Parsing schemata

Sikkel (1993) introduces the concept of **parsing schemata**, a formal definition of a deductive parsing algorithm. A parsing algorithm is presented as a formally defined parsing system. This thesis will follow his definition, alongside the work of (Capelletti, 2007). Capelletti defines a parsing system as follows:

Definition 2.1.1 (Parsing system)**.**
A parsing system $\mathcal{D}$ is a triple $\langle \mathcal{I}, \mathcal{A}, \mathcal{R} \rangle$ where $\mathcal{I}$ is the set of items, $\mathcal{A}$ is the set of axioms of $\mathcal{D}$ and $\mathcal{R}$ is the set of inferences whose premises and conclusions are items. The set of items consists of a special goal item, that spans the input string with the start category.

An item is an intermediate parsing result, a concept that is central to Sikkel's parsing schemata. A parsing deduction succeeds if the goal item can be derived. Parsing schemata make it easier to reason about deductive parsing systems: they offer a high-level abstraction that enables us to separate the **declarative** and the **procedural** properties of the system. Furthermore, this allows us to easily proof certain parsing properties such as its correctness.

Parsing can essentially be done in two ways: **top-down** or **bottom-up**. In the next two subsections we will provide a short overview of these methods, based on the definitions of (Shieber, Schabes, and Pereira, 1995) and (Capelletti, 2007).

2.1.1 Pure top-down parsing

Top-down parsing is *goal-driven*: a derivation is constructed from the root symbol (the goal) towards its leaves (the axioms). **Recursive descent** parsing is a *pure* top-down parsing method. Given a context-free grammar and some input $w_i \cdots w_n$, items are of the form $[\bullet\beta, j]$, where $j \leq n$. Such an item asserts that the substring up to position j followed by a substring of type β forms a valid sentence in our language. Parsing starts with an item $[\bullet S, 0]$ and is finished if a goal item $[\bullet, n]$ is deduced. The system can be presented as a parsing system. Note that we adhere to the notion where production rules are presented in the form of $\alpha \to A$, as done by Capelletti.

$$
\begin{aligned}
\mathcal{I} = &\ \{(\bullet\beta, j) \mid \beta \in \mathcal{P}^*, 0 \leq j \leq n\} \\
\mathcal{A} = &\ \{(\bullet S, 0)\} \\
\mathcal{R} = &\begin{cases}
\dfrac{(\bullet w_{j+1}\beta, j)}{(\bullet\beta, j+1)} & \textit{Scan} \\[2ex]
\dfrac{(\bullet B\beta, j)}{(\bullet\gamma\beta, j)}\ \gamma \to A \in \mathcal{P} & \textit{Predict}
\end{cases}
\end{aligned}
$$

At each step, we either expand a non-terminal by rewriting it following a production rule in our grammar, or we read in a terminal symbol that corresponds with our input string. If our grammar is highly ambiguous then this process is quite inefficient, as many derivations are tried before the goal item is deduced.

2.1.2 Pure bottom-up parsing

Bottom-up parsing is *data-driven*: deductions are built up from premises (the input data) to conclusion. A method for pure bottom-up parsing is the Shift-Reduce algorithm. Items now have the form $[\alpha\bullet, j]$. This asserts that $\alpha \overset{*}{\Rightarrow} w_i \cdots w_j$, and can be seen as the dual of the top-down items that were shown above. Shift-Reduce works in an opposite fashion compared to the recursive descent algorithm: terminal and non-terminal symbols α are contracted into one non-terminal A if there exists a production rule $\alpha \to A$. Parsing if finished if a $[S\bullet, n]$ is deduced.

$$
\begin{aligned}
\mathcal{I} = &\ \{(\beta\bullet, j) \mid \beta \in \mathcal{P}^*, 0 \leq j \leq n\} \\
\mathcal{A} = &\ \{(\bullet, 0)\} \\
\mathcal{R} = &\begin{cases}
\dfrac{(\alpha\bullet, j)}{(\alpha w_{j+1}\bullet, j+1)} & \textit{Shift} \\[2ex]
\dfrac{(\alpha\gamma\bullet, j)}{(\alpha B\bullet, j)}\ \gamma \to A \in \mathcal{P} & \textit{Reduce}
\end{cases}
\end{aligned}
$$

2.1.3 Earley parsing

The Earley parser is based on a *mixed* parsing regime. It is based on the top-down procedure of the recursive descent algorithm, but the *Complete* rule acts in a bottom-up manner. The Earley parsing system for CFGs is defined as the following triple

$\langle \mathcal{I}, \mathcal{A}, \mathcal{R} \rangle$, given some input string $w_1, ..., w_n$:

$$
\begin{aligned}
\mathcal{I} = \;& \{(i, \Gamma \bullet \Delta \to \mathcal{C}, j) \mid \Gamma\Delta \to \mathcal{C} \in \mathcal{P}, 0 \leq i \leq j \leq n\} \\
\mathcal{A} = \;& \{(i-1, w_i, i) \mid 1 \leq i \leq n\} \\
\mathcal{R} = \;& \begin{cases}
(0, \bullet\Gamma \to S, 0) \text{ for all } \Gamma \to S \in \mathcal{P} & \textit{Init} \\
\dfrac{(i, \Delta \bullet w\Gamma \to \mathcal{C}, j) \quad (j, w, j+1)}{(i, \Delta w \bullet \Gamma \to \mathcal{C}, j+1)} & \textit{Scan} \\
\dfrac{(i, \Delta \bullet A\Gamma \to \mathcal{C}, j)}{(j, \bullet\Lambda \to A, j)} \; \Lambda \to A \in \mathcal{P} & \textit{Predict} \\
\dfrac{(k, \Lambda\bullet \to A, j) \quad (i, \Delta \bullet A\Gamma \to \mathcal{C}, k)}{(i, \Delta A \bullet \Gamma \to \mathcal{C}, j)} & \textit{Complete}
\end{cases}
\end{aligned}
$$

Items can be interpreted as a triple $(i, \Gamma \bullet \Delta \to \mathcal{C}, j)$ in which the index i denotes the start position of the production rule that is currently being parsed, the $\bullet$ the current position of the parsing process in the production rule $\Gamma\Delta \to \mathcal{C}$ (where Γ and Δ can be empty sequences), and j the index of the current position of the input string. Note that $\mathcal{I}$ denotes the set of *all* possible items: not all of these items can actually be deduced. In words the inferences of the parsing process can be described as follows.

Scan: Read in an input symbol if it corresponds with the next terminal symbol in the current production rule.

Predict: Predict the corresponding rule of the next non-terminal symbol.

Complete: Complete the parsing of a non-terminal and move on to the next symbol.

Parsing is successful if we deduce an item $(0, \Gamma\bullet \to S, n)$. The parsing process unfolds itself as a depth-first search tree, and ambiguity emerges when multiple derivations can be found that lead to the final item. It is easy to see that this process becomes increasingly more complex if our grammar contains a lot of ambiguous rules. The efficiency of Earley parsing is thus strongly dependent on the production rules of the grammar. An advantage of the system is that it works with any CFG, i.e. the productions don't have be defined in some kind of normal form.

2.1.4 CYK parsing

An influential bottom-up parser is the **CYK algorithm**. It is not purely bottom-up: just like the Earley parser it is based on a mixed regime of bottom-up and top-down methods.

The CYK parsing system for CFGs (in Chomsky Normal Form[1] without ϵ-productions) is defined as the following triple $\langle \mathcal{I}, \mathcal{A}, \mathcal{R} \rangle$, given some input string $w_1, ..., w_n$:

$$\begin{aligned} \mathcal{I} = &\ \{(i, A, j) \mid A \in \mathcal{F}, 0 \leq i \leq j \leq n\} \\ \mathcal{A} = &\ \{(i-1, A, i) \mid w_i \rightarrow A \in \text{Lex}\} \\ \mathcal{R} = &\ \left\{ \frac{(i, B, k) \quad (k, C, j)}{(i, A, j)} \text{ if } B\ C \rightarrow A \in \mathcal{P} \quad \textit{Complete} \right. \end{aligned}$$

$\mathcal{F}$ denotes the set of non-terminals, Lex denotes the lexicon of the grammar and $\mathcal{P}$ denotes the set of production rules.

Compared to the Earley parser, the inference step of CYK parsing is relatively simple: at each step two non-terminals are combined if there exists a production rule of that form in the grammar. Parsing is finished if an item $(0, S, n)$ can be deduced, and once again ambiguity arises if multiple derivations are found.

CYK parsing is usually combined with a form of *tabulation*, where intermediate results are stored in a tabel (also called a *chart*). This reduces a great amount of redundancy, because once an item is encountered that is already stored in our chart, we don't need to parse this item again.

2.2 Properties of parsing systems

If we design a parser for a certain language, it is important to ensure that this parser is able to parse all possible strings that belong to that language. This can be proved formally. A parsing algorithm is **sound** if each input that is accepted by the parser is an element of the language. It is **complete** if each element of the language is accepted by the parser. Kallmeyer (2010) proved these properties for CYK as follows. Soundness and completeness are proved by proving the following statement:

$$(i, A, j) \ \textit{iff} \ A \overset{*}{\Rightarrow} w_{i+1} \cdots w_j$$

This can be shown by induction over the length l of the span, where $l = j - i$.

$l = 1$ This case is covered by our set $\mathcal{A}$ of axiom items.

$l > 1$ This is covered by the only inference rule in the parsing system. It can be assumed that the induction claim holds, as the antecedent items have a smaller span than the consequent item. Assume that (i, A, j) was obtained by the *complete* rule. $B\ C \rightarrow A$ is then a production rule in our grammar. $B \overset{*}{\Rightarrow} w_{i+1} \cdots w_j$ and $C \overset{*}{\Rightarrow} w_{j+1} \cdots w_k$ gives us that $A \overset{*}{\Rightarrow} w_{i+1} \cdots w_k$. If $A \overset{*}{\Rightarrow} w_{i+1} \cdots w_k$, where $k > i + 1$, then there must exist a production rule $B\ C \rightarrow A$ as above, from which follows that (i, A, k).

If we would apply this to our goal item $(0, S, |w|)$, we obtain soundness and completeness.

[1] In a CFG in CNF, all productions are of the form:

- $AB \rightarrow C$, where $A \neq S$ & $B \neq S$
- $w \rightarrow A$, where w is a terminal symbol
- $\epsilon \rightarrow S$

The **complexity** of a parsing algorithm is determined by the maximal number of different rule applications that are possible (Kallmeyer, 2010). Complexity is expressed in big O notation, which describes an upper bound on the growth rate of a certain function (or algorithm, in our case). The complexity of CYK parsing is determined by the *complete* rule. In this rule there are three indices (i, j and k) that range over an input string of length n. The maximal number of possible applications that are possible then becomes $|P| \cdot n^3$, where $|P|$ is the number of production rules. If we assume that our grammar is fixed, then $|P|$ can be treated as a constant and the complexity of CYK becomes $\mathcal{O}(n^3)$.

Chapter 3

Bottom-up parsing fLG

The next two chapters will focus on a procedure for bottom-up parsing fLG. They are organized as follows: chapter 3 will provide a theoretical background and a general overview of the parsing process. Furthermore a link between this procedure and the Partial Proof Trees of (Joshi and Kulick, 1997) will be presented. Chapter 4 will discuss our computational implementation of this parser in Prolog, a logic programming language.

Recall that a bottom-up parsing deduction is built up from the premises towards a conclusion. In the case of our parser, these premises could be a list of lexical items that are assigned a logical formula by a lexicon. By 'combining' these formulas using the rules of fLG, we aim to achieve a (structured) goal sequent of the form $A_1 \otimes ... \otimes A_n \vdash B_1 \oplus ... \oplus B_m$. The procedure that is presented here consists of two phases:

1. The **unfolding** phase. All input items are assigned a logical formula according to a **lexicon**. These formulas are then *unfolded*, a form of **partial deduction**. The idea of partial deduction is that the parsing phase is preceded by a compilation of the lexicon (Hendriks, 1993): a **partial proof tree** is derived for each item according to a set of logical rules. A partial proof tree is a proof tree that contains holes: not every branch in the tree has to end with an axiom, but it can end up in an unbound variable with certain unification constraints too. These unfolded formulas will act as the items of the next phase. The unfolding phase thus acts on the *base logic* of fLG: inferences are made on the basis of the (logical) rules of fLG. This phase could be done *off-line*: instead of assigning a lexical item to a logical formula, we could assign the item to an unfolded formula that is derived beforehand by unfolding.

2. The **parsing** phase. The partial proof trees that were obtained by unfolding in the previous phase are now combined by a set of residuation rules and unification procedures in a bottom-up regime, until the goal sequent is derived and there are no unfolded formulas left. This is a form of CYK parsing. A partial proof tree is unified with another tree by inserting it on the position of a hole in the other tree. The inferences of this phase are *meta-logical*: inferences are made on the basis of the unfolded formulas, instead of the sequents themselves.

Because the two phases involve two different kind of logics, the whole system is said to make use of a *hybrid logic* (Joshi, Kulick, and Kurtonina, 1998). This is similar to the Partial Proof Trees that are described in section 3.4.

3.1 Unfolding

3.1.1 Unfolding L

The unfolding procedure presented here, is based on the concept of partial deduction as described by (Hendriks, 1993) in section 5.1. Hendriks provides a concise formal definition for the partial deduction of the fragment of **L** that consists only of the / and \ operators. The functions that are defined by Hendriks will form the basis of our system for fLG. Firstly, he defines a function A that returns the set of arguments of a category, and a function A^A that returns the set of arguments of the arguments of a category.

$$A(at) = \emptyset; A(a/b) = A(b\backslash a) = \{b\} \cup A(a)$$
$$A^A(at) = \emptyset; A^A(a/b) = A^A(b\backslash a) = A(B) \cup A^A(a)$$

For a set of categories C, the set C^+ is the set that contains C and all of the arguments (and, recursively, all their arguments) of C. This set C^+ is essential for the unfolding process, as it defines the categories that need to be unfolded. (3.1) shows the unfolding of $(s/(n\backslash s))\backslash s$, a case where C^+ contains 2 elements.

Next, Hendriks defines a function f that *'peels off the argument categories from the goal category of the sequent to which it is applied'*. This could be compared to the residuation laws of (1.4). The semantic terms are omitted, as we are primarily interested in the syntactic derivation.

$$f(T \vdash at) = T \vdash at$$
$$f(T \vdash a/b) = f(T, b \vdash a)$$
$$f(T \vdash b\backslash a) = f(b, T \vdash a)$$

Finally, the function g actually partially deduces a category (semantic terms are ignored again for the sake of clarity). In this definition, **X** and **Y** denote unstructured sequences of formulas. These sequences are unstructured, because L allows global associativity.

$$g(at) = at \vdash at$$
$$g(a/b) = \frac{f(T_m \vdash b) \quad \sigma_m \quad \cdots \quad \sigma_1}{\mathbf{X}, a/b, T_{m+1}, \mathbf{Y} \vdash at}$$
$$g(b\backslash a) = \frac{f(T_m \vdash b) \quad \sigma_m \quad \cdots \quad \sigma_1}{\mathbf{X}, T_{m+1}, b\backslash a, \mathbf{Y} \vdash at}$$

iff $g(a)$ consists of the conclusion $\mathbf{X}, a, \mathbf{Y} \vdash at$ and the premises $\sigma_m, ..., \sigma_1$ (where m is the number of arguments of a):

$$g(a) = \frac{\sigma_m \quad \cdots \quad \sigma_1}{\mathbf{X}, a, \mathbf{Y} \vdash at}$$

The whole procedure is exemplified in the following unfolding of $(s/(n\backslash s))\backslash s$.

$$\begin{aligned} \{(s/(n\backslash s))\backslash s\}^+ &= \{(s/(n\backslash s))\backslash s, n\backslash s\} \\ g((s/(n\backslash s))\backslash s) &= \frac{T_1, n\backslash s \vdash s}{T_1, (s/(n\backslash s))\backslash s \vdash s} \\ g(n\backslash s) &= \frac{T_1 \vdash n}{T_1, n\backslash s \vdash s} \end{aligned} \tag{3.1}$$

In both these cases, $g(s) = s \vdash s$, so the condition for g holds.

3.1.2 Unfolding fLG

The functions of Hendriks provide a solid basis for our unfolding procedure. In its essence, the unfolding procedure for fLG is roughly based on the function g. The Display Logic format, however, forces us to take on a slightly different approach.

First, I will give a formal definition of an unfolded formula. These unfolded formulas will behave as the items of our parser.

Definition 3.1.1 (Unfolded formula)**.**
*An unfolded formula is a tuple $\langle p, \mathcal{H} \rangle$ where p is a partial proof tree, and $\mathcal{H}$ the set of holes. An unfolded formula is **closed** if $\mathcal{H}$ is empty, and **open** otherwise.*

Note that 'unfolded formula' and 'item' are used interchangeably, but that 'partial proof tree' is not analogous to an unfolded formula, as it is a part of an unfolded formula. A hole of an unfolded formula is a triple that contains an unbound variable that is part of the partial proof tree p, that can be unified with other unfolded formulas. The other two elements of the triple define the constraints on the unification of the unbound variable in the partial proof tree. It is defined as follows:

Definition 3.1.2 (Hole)**.**
A hole h is a triple $\langle s, v, \mathcal{Y} \rangle$ where s is the sequent that corresponds with an unbound variable v in the partial proof tree of an unfolded formula, and $\mathcal{Y}$ the set of hypothesis constraints.

If some unfolded formula u_1 is unified with another unfolded formula u_2 according to some hole h_1 in u_1, a hypothesis constraint in h_1 is a partial proof tree that must be a *sub-tree* of the partial proof tree of u_2. Furthermore, the sequent of h_1 must match with the sequent of u_2. An example of an unfolded formula can be found in figure 3.1 (np and n both have a positive bias). A hypothesis constraint can be compared to the vertical dots that represent a hypothesis relation in a proof tree, such as in figure 3.2.

The unfolding phase can be described formally as a collection of functions that are formulated below. The hypotheses of a formula are unfolded too. The partial proof tree of such a hypothesis can thus be found on two different locations: as a hypothesis constraint of the original formula, and as the partial proof tree of the unfolding of the hypothesis itself.

The first function that is defined is the function u (3.2) that returns a set of unfolded formulas: the unfolded formula itself and all its unfolded hypotheses. This is a recursive process: if a hypothesis contains hypotheses, they are unfolded too. Due to the symmetric nature of LG, the side of the sequent we want to unfold is provided as a parameter. This parameter is either in, for the left-hand side (lhs) of the sequent, or out, for the right-hand side (rhs). $\{in, out\} \times \mathcal{F}$ denotes the domain of u, where

$$\left\langle\ (np/n) \otimes n \vdash X,\ \frac{x}{(np/n) \otimes n \vdash X}\ (\otimes L),\ \mathcal{H}\ \right\rangle$$

$$\mathcal{H} = \left\{\ \left\langle\ (np/n) \cdot \otimes \cdot n \vdash X,\ x,\ \left\{\ \frac{\dfrac{y}{\boxed{np} \vdash A} \leftarrow \quad \dfrac{}{B \vdash \boxed{n}}}{\dfrac{\boxed{np/n} \vdash A \cdot / \cdot B}{np/n \vdash A \cdot / \cdot B}\ \leftarrow}\ (/L)\ \right\}\ \right\rangle\ \right\}$$

FIGURE 3.1: The unfolding of $(np/n) \otimes n$, on the left-hand side of the sequent. Note that the variable x in the partial proof tree corresponds to the variable x that is the second element of the only hole of the unfolded formula.

$$\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{np \vdash A}}{\boxed{np} \vdash A} \leftarrow \quad \dfrac{}{B \vdash \boxed{n}}}{\boxed{np/n} \vdash A \cdot / \cdot B}\ (/L)}{np/n \vdash A \cdot / \cdot B}\ \leftarrow}{\dfrac{\dfrac{\vdots}{(np/n) \cdot \otimes \cdot n \vdash X}}{(np/n) \otimes n \vdash X}\ (\otimes L)}$$

FIGURE 3.2: An alternative representation of the unfolding of $(np/n) \otimes n$. Although this representation is more elegant than that of figure 3.1, it turned out to be harder to define a unification procedure in combination with the vertical dots.

$\mathcal{F}$ is the set of formulas in our lexicon, and the arguments of these formulas. $\mathcal{L}$ denotes our lexicon, and the function f the mapping from a word to its corresponding formula. Note that this restricts the function to only operate on formulas that are found in the lexicon. $\mathcal{P}(\mathcal{I})$ is the codomain of u, where $\mathcal{I}$ denotes the set of all the possible unfolded formulas given our lexicon, which is the set of items of our system. A definition for $\mathcal{I}$ will be provided later on. The function a^a (3.3) returns all the arguments of the arguments of a formula, similar to the function A^A that was defined by Hendriks. The items of a^a and our input formula x are the formulas that are to be unfolded.

$$\begin{aligned} \mathcal{F} &= \{\ x \mid w \in \mathcal{L}, x \in a(in, f(w)) \vee x \in a(out, f(w))\ \} \\ u &: (\{in, out\} \times \mathcal{F}) \to \mathcal{P}(\mathcal{I}) \\ u(io, x) &= \{\ g(io, x)\ \} \cup \{\ g(io_2, a) \mid \langle io_2, a\rangle \in a^a(io, x)\ \} \end{aligned} \tag{3.2}$$

The function a^a returns, given some formula x, the set of formulas that are hypotheses of x, paired with the side of the sequent (in or out) on which the hypothesis will be unfolded. The function a returns the input formula (and its arguments) if the in or out argument of a is the *logical* side of the formula. The logical side of a formula is the side for which its monotonicity rule is defined: the $/$ and $\backslash$ operators, for example, have their logical side on the in position, whereas

the logical side of the $\otimes$ operator is on the *out* position. A formula that does not occur on its logical side, is said to occur on its *structural* side. Note that at is an atomic formula, such as s or np. For example: $a(in, (n\backslash n)/(s/\Diamond\Box((np/n) \otimes n))) = \{ \langle in,\ (n\backslash n)/(s/\Diamond\Box((np/n) \otimes n))\rangle\ ,\ \langle in,\ \Box((np/n) \otimes n)\rangle\ ,\ \langle in,\ np/n\rangle\ \}$.

$$
\begin{aligned}
a &: \{in, out\} \times \mathcal{F} \to \mathcal{P}(\{in, out\} \times \mathcal{F}) \\
a^a &: \{in, out\} \times \mathcal{F} \to \mathcal{P}(\{in, out\} \times \mathcal{F}) \\
a(in, at) &= a(out, at) = a^a(in, at) = a^a(out, at) = \emptyset \\
a(in, A \oplus B) &= \{\langle in, A \oplus B\rangle\} \cup a^a(in, A) \cup a^a(in, B) \\
a(in, A/B) &= \{\langle in, A/B\rangle\} \cup a^a(in, A) \cup a^a(out, B) \\
a(in, B\backslash A) &= \{\langle in, B\backslash A\rangle\} \cup a^a(in, A) \cup a^a(out, B) \\
a(out, A \otimes B) &= \{\langle out, A \otimes B\rangle\} \cup a^a(out, A) \cup a^a(out, B) \\
a(out, A \oslash B) &= \{\langle out, A \oslash B\rangle\} \cup a^a(out, A) \cup a^a(in, B) \\
a(out, A \otimes B) &= \{\langle out, A \otimes B\rangle\} \cup a^a(out, A) \cup a^a(in, B) \\
a(in, \Box A) &= \{\langle in, \Box A\rangle\} \cup a^a(in, A) \\
a(out, \Diamond A) &= \{\langle out, \Diamond A\rangle\} \cup a^a(out, A) \\
a(out, A \oplus B) &= a^a(out, A \oplus B) = a(out, A) \cup a(out, B) \\
a(in, A \otimes B) &= a^a(in, A \otimes B) = a(in, A) \cup a(in, B) \\
a^a(in, A \oplus B) &= a^a(in, A) \cup a^a(in, B) \\
a^a(out, A \otimes B) &= a^a(out, A) \cup a^a(out, B) \\
a(out, A/B) &= a(out, B\backslash A) = a(in, B \oslash A) = a(in, A \oslash B) = \\
a^a(out, A/B) &= a^a(out, B\backslash A) = a^a(in, B \oslash A) = a^a(in, A \oslash B) = a(out, A) \cup a(in, B) \\
a^a(in, A/B) &= a^a(in, B\backslash A) = a^a(out, B \oslash A) = a^a(out, A \oslash B) = a(in, A) \cup a(out, B) \\
a(out, \Box A) &= a^a(out, \Box A) = a(out, A) \\
a(in, \Diamond A) &= a^a(in, \Diamond A) = a(in, A) \\
a^a(in, \Box A) &= a^a(in, A) \\
a^a(out, \Diamond A) &= a^a(out, A)
\end{aligned}
\tag{3.3}
$$

The function g actually creates an item. Recall that an item is a tuple $\langle\ p, \mathcal{H}\ \rangle$, where p is a partial proof tree, and $\mathcal{H}$ a set of holes. The function g consists of two auxiliary functions:

1. The function s (3.6) creates an input or output structure that corresponds with a formula. This structure, together with the input formula, forms the sequent that is going to be unfolded.

2. The function p (3.7) returns a tuple. The first element of the tuple is the partial proof tree of the unfolded formula. The second element is a list of all the holes in the partial proof tree that is derived by p. The functions π^1 and π^2 return the first and second element of a tuple, respectively.

$$
\begin{aligned}
g &: (\{in, out\} \times \mathcal{F}) \to \mathcal{I} \\
g(in, x) &= \langle\ \pi^1 p(x \vdash s_{out}(x)),\ \pi^2 p(x \vdash s_{out}(x))\ \rangle \\
g(out, x) &= \langle\ \pi^1 p(s_{in}(x) \vdash x),\ \pi^2 p(s_{in}(x) \vdash x)\ \rangle
\end{aligned}
\tag{3.4}
$$

The function s returns an input or output structure, given a formula and the side of the sequent we want the formula to be unfolded (*in* or *out*). The sequent for some formula A then becomes: $A \vdash s_{out}(A)$ if we unfold the in or left-hand side (lhs), and $s_{in}(A) \vdash A$ if we unfold the out or right-hand side (rhs). The definition of s is based on the structure grammar of (1.12). A and B denote formula variables, X denotes an unbound structure variable. The codomain of s is defined as $\mathcal{SV}$ (3.5). This is the set of all possible input and output structures that consist of structure variables. It is a finite set, as the depth of a structure is bounded by the number of arguments of the formula in our lexicon $\mathcal{L}$ that has the most arguments. The number of arguments of a formula is calculated by the function $\#$, based on the similar function that was defined by Hendriks. The function f maps a lexical item to its corresponding formula. The function $sv(n)$ returns all the structures that consists of n arguments or less.

$$
\begin{aligned}
\mathcal{SV} &= sv(\max_{w \in \mathcal{L}}(\#(f(w)))) \\
sv &: \mathbb{N} \to \mathcal{SV} \\
sv(0) &= \{X\} \\
sv(n) &= \{X\} \cup \{ s_1 \cdot \$ \cdot s_2, \mid s_1 \in sv(n-1),\ s_2 \in sv(n-1)\} \cup \{\langle s_1 \rangle, [s_1] \mid s_1 \in sv(n-1)\} \\
&\quad \textit{Where } \$ \in \{/, \backslash, \otimes, \oslash, \oslash, \oplus\} \\
\#(at) &= 0 \\
\#(A/B) = \#(B\backslash A) &= \#(A \oslash B) = \#(B \oslash A) = \#(\Diamond A) = \#(\Box A) = \#(A) + 1 \\
\#(A \otimes B) &= \#(A \oplus B) = \max(\#(A), \#(B)) + 1
\end{aligned}
\tag{3.5}
$$

$$
\begin{aligned}
& s : \{in, out\} \times \mathcal{F} \to \mathcal{SV} \\
& s(out, at) = s(in, at) = X \\
& s(out, A \otimes B) = X \qquad s(in, A \oplus B) = X \\
& s(out, A/B) = s(out, A) \cdot / \cdot s(in, B) \qquad s(in, A \oslash B) = s(in, A) \cdot \oslash \cdot s(out, B) \\
& s(out, B\backslash A) = s(in, B) \cdot \backslash \cdot s(out, A) \qquad s(in, B \oslash A) = s(out, B) \cdot \oslash \cdot s(in, A) \\
& s(in, A \otimes B) = s(in, A) \cdot \otimes \cdot s(in, B) \qquad s(out, A \oplus B) = s(out, A) \cdot \oplus \cdot s(out, B) \\
& s(in, A/B) = X \qquad s(out, A \oslash B) = X \\
& s(in, B\backslash A) = X \qquad s(out, B \oslash A) = X \\
& s(out, \Diamond A) = X \qquad s(in, \Diamond A) = \langle s(in, A) \rangle \\
& s(out, \Box A) = [s(out, A)] \qquad s(in, \Box A) = X
\end{aligned}
\tag{3.6}
$$

For example, $s_{out}((np\backslash s)/s) = ((X \cdot \backslash \cdot Y) \cdot / \cdot Z)$, whereas $s_{in}((np\backslash s)/s) = X$ and $s_{out}(s/(np\backslash s)) = X \cdot / \cdot Y$.

Having obtained a bottom sequent, we can start with the actual unfolding process. Recall that the rules of fLG (and sLG) can be divided into 3 groups (1.3.1): the identity, structural and logical group. For unfolding, we will only permit the rules of the identity group (without cut!) and the logical group. Furthermore, we add the focusing and defocusing rules of (1.20), as the original division of the three groups was

made for sLG. Unfolding then acts as a simple backward chaining proof search procedure: the sequent is deduced until a (co)axiom is reached, or a formula is situated on its *structural* side of the sequent.

The structural and logical sides of a formula are determined by the structure grammar of (1.12), as stated earlier. If, for example, the formula $(np/n) \otimes n$ occurs on the lhs (the *in*-position), it is situated on the wrong side of the sequent to be eligible for the monotonicity rules. By definition of s, a formula on its structural position has only one corresponding structure variable on the other side of the turnstile. If a formula is situated on its structural side, then the whole formula is rewritten to its structural counterpart according to the rules of (1.15). However, if a formula is encountered during this rewriting process that has its monotonicity rule on the side of the sequent that is rewritten, it is unfolded and added as a hypothesis constraint. In the example of $(np/n) \otimes n \vdash X$, (np/n) is a formula that has its monotonicity rule on the lhs. The proof of the unfold of (np/n) is then added as a hypothesis constraint to the initial hole, as shown in figure (3.1).

The function p returns, given a sequent, a tuple: the partial proof tree of a sequent and the set of holes that corresponds with that partial proof tree. $\mathcal{S}$ is the domain of p, and denotes the sequent space of our system, a concept that I address in the next section. The inferences that are made during the computation are based on the logical rules of fLG. Recall that the functions π^1 and π^2 return the first and second element of a tuple, respectively. The function pol returns the polarity of a formula (a structure has no polarity). A variable in a partial proof tree is represented by a lower case letter: x, y, etc.. Note that the sequent of a hole is *unfocused*. The creation of a hole consists of one auxiliary functions w. w rewrites a sequent to its structural counterpart and returns the set hypothesis constraints for a hole.

$$p : \mathcal{S} \to (\mathcal{P}r \times \mathcal{P}(\mathcal{H}))$$

$$p(S) = \begin{cases} \langle \dfrac{}{S}\ I,\ \emptyset \rangle & \text{if } \dfrac{}{S}\ I, \text{ where } I \in \{ (1.19) \} \\ \langle \dfrac{\pi^1(p(S'))}{S}\ I,\ \pi^2(p(S')) \rangle & \text{if } \dfrac{S'}{S}\ I, \text{ where } I \in \{ (1.20), (1.22) \} \\ \langle \dfrac{\pi^1(p(S'))\quad \pi^1(p(S''))}{S}\ I,\ \pi^2(p(S')) \cup \pi^2(p(S'')) \rangle & \text{if } \dfrac{S'\quad S''}{S}\ I, \text{ where } I \in \{ (1.21) \} \\ \langle x,\ \{\langle \pi^1(w(in,\ A)),\ x,\ \pi^2(w(in,\ A)) \rangle & \text{if } S = A \vdash B \text{ and } pol(A) = + \\ \langle x,\ \{\langle \pi^1(w(out,\ B)),\ x,\ \pi^2(w(in,\ B)) \rangle & \text{if } S = A \vdash B \text{ and } pol(B) = - \end{cases} \tag{3.7}$$

The function w is, like p, a function that returns a tuple. Its arguments are a sequent position (in or out) and a formula. This is a formula that is located on its *structural* side, which means that it can be rewritten to its structural counterpart following the rewrite rules of (1.15). This rewriting occurs recursively, following the structure grammar of (1.12). When a formula is encountered that has its monotonicity rule on the given side of the sequent, the proof of that formula is computed using p and added as a hypothesis constraint. The codomain of w is $(\mathcal{S}t \times \mathcal{P}(\mathcal{P}r))$, where $\mathcal{S}t$ denotes the set of possible structures in our system, and $\mathcal{P}r$ the set of possible partial proof trees.

$$w : (\{in, out\} \times \mathcal{F}) \to (\mathcal{S}t \times \mathcal{P}(\mathcal{P}r))$$

$$w(in,\ at) = w(out,\ at) = \langle\ at,\ \emptyset\ \rangle$$

$$w(in,\ X) = \begin{cases} \langle\ \pi^1 w(in,\ A) \cdot \otimes \cdot \pi^1 w(in,\ B),\ \pi^2 w(in,\ A) \cup \pi^2 w(in,\ B)\ \rangle & \text{if } X = A \otimes B \\ \langle\ \pi^1 w(in,\ A) \cdot \oslash \cdot \pi^1 w(out,\ B),\ \pi^2 w(in,\ A) \cup \pi^2 w(out,\ B)\ \rangle & \text{if } X = A \oslash B \\ \langle\ \pi^1 w(out,\ B) \cdot \otimes\!\!\!/ \cdot \pi^1 w(in,\ A),\ \pi^2 w(out,\ B) \cup \pi^2 w(in,\ A)\ \rangle & \text{if } X = B \otimes\!\!\!/ A \\ \langle\ \langle \pi^1 w(in,\ A) \rangle,\ \pi^2 w(in,\ A)\ \rangle & \text{if } X = \Diamond A \\ \langle\ X,\ \{\pi^1 p(\ X \vdash s(out,\ X)\ )\}\ \rangle & \text{if } pol(X) = - \end{cases}$$

$$w(out,\ X) = \begin{cases} \langle\ \pi^1 w(out,\ A) \cdot \oplus \cdot \pi^1 w(out,\ B),\ \pi^2 w(out,\ A) \cup \pi^2 w(out,\ B)\ \rangle & \text{if } X = A \oplus B \\ \langle\ \pi^1 w(in,\ B) \cdot \backslash \cdot \pi^1 w(out,\ A),\ \pi^2 w(in,\ B) \cup \pi^2 w(out,\ A)\ \rangle & \text{if } X = B \backslash A \\ \langle\ \pi^1 w(out,\ A) \cdot / \cdot \pi^1 w(in,\ B),\ \pi^2 w(out,\ A) \cup \pi^2 w(in,\ B)\ \rangle & \text{if } X = A / B \\ \langle\ [\pi^1 w(out,\ A)],\ \pi^2 w(out,\ A)\ \rangle & \text{if } X = \Box A \\ \langle\ X,\ \{\pi^1 p(\ s(in,\ X) \vdash X\ )\}\ \rangle & \text{if } pol(X) = + \end{cases} \tag{3.8}$$

3.2 Parsing unfolded formulas

Now that a solid definition for unfolding has been established, a method that actually parses our input can be constructed. In line with Capelletti and Sikkel, I will provide a formal definition of the procedure in the form of a parsing system (as described in section 2.1). Before I present this definition, I will provide an informal description of the parsing process.

Recall that the unfolded formulas are the items of our system, of the form $\langle p, \mathcal{H} \rangle$, with p the partial proof tree, and $\mathcal{H}$ the set of holes. Our input will have the form $A_1, ..., A_i \vdash B_j, ..., B_n$, with $1 \leq i < j \leq n$. This input is then unfolded, following the functions of the previous section. The unfoldings of atomic formulas are omitted during the parsing phase: they are unified at the end of the procedure with the unbound structure variables of the final item. The unification phase will only consist of inferences that are made from the structural group (the set $I = \{$ (1.14), (1.17), (1.18) $\}$). This is possible because the unfolding of a formula starts with an unfocused sequent, and either ends with a (co)axiom, or an unfocused sequent that is part of a hole. At each step there are three possible inferences that can be made:

1. Make a logical inference on the basis of our bottom sequent and the set I.
2. **Focus shift** our item in case there are no closed items available. This method is described below (3.2.2).
3. Unify two unfolded formulas: an unfolded formula u_1 which has an empty set of holes (*closed*) is unified with an *open* unfolded formula u_2 that contains a hole $\langle s_2, x_2, \mathcal{Y}_2 \rangle$ that can be unified with u_1. Unification is possible if the bottom sequent of the partial proof tree of u_1 can be unified with the sequent s_2, and if all hypothesis constraints $\mathcal{Y}_2$ are sub-trees of u_1. When two items are unified, the partial proof tree of u_1 is 'plugged into' the partial proof tree of u_2, by substituting the variable of x_2 in the partial proof tree of u_2.

This whole process is a form of CYK parsing. Parsing is complete once one closed item is deduced. The yield of the bottom sequent of this item (the formulas at the leaves of the input and output structures) must correspond with the initial input sequent. At this point, the atomic formulas are unified with the remaining open structure variables. I will now provide several formal definitions of concepts that are needed to arrive at a complete definition of the parsing system.

The first concept is the **unfold system**: a tuple that contains all the unfolded formulas at the start of the parsing process. They are divided into two sets, the closed and open unfolded formulas. The set of open items is defined to be all items that are not closed, by definition of the *relative complement*. Note that f is a function that maps a lexical item to a formula, given some lexicon $\mathcal{L}$.

Definition 3.2.1 (Unfold system)**.** *An unfold system U for a lexicon $\mathcal{L}$ and an input $w_1, ..., w_i \vdash w_j, ..., w_n$, with $1 \leq i < j \leq n$, is a tuple $\langle \mathcal{C}, \mathcal{O} \rangle$, where $\mathcal{C}$ is the initial set of closed unfolded formulas and $\mathcal{O}$ the initial set of open unfolded formulas:*

$$\mathcal{C} = \{\ \langle p, \emptyset \rangle \mid 1 \leq x \leq i,\ \langle p, \emptyset \rangle \in u(in, f(w_x))\ \} \cup \{\ \langle p, \emptyset \rangle \mid j \leq x \leq n,\ \langle p, \emptyset \rangle \in u(out, f(w_x))\ \}$$

$$\mathcal{O} = (\{\ u \mid 1 \leq x \leq i,\ u \in u(in, f(w_x))\ \} \cup \{\ u \mid j \leq x \leq n,\ u \in u(out, f(w_x))\ \}) \setminus \mathcal{C}$$

Focus shifting shifts the focus of a hole, to create a (co)axiom. Not every hole of an item has to be unified with another item. If the hole is *atomic*, then it can be turned into a (co)axiom. The structure variable is then substituted by the atomic formule.

Definition 3.2.2 (Focus shifting)**.** *The partial function fs is a function that turns an open atomic hole ($at \vdash X$ or $X \vdash at$, with at an atomic formula) into a (co)-axiom. The output of the function is the proof tree presentation of a focus and (co)-axiom rule in fLG.*

$$fs : \mathcal{S} \to \mathcal{P}r$$

$$fs(at \vdash X) = \dfrac{\dfrac{}{at \vdash \boxed{X}}\ Ax}{at \vdash X}\ \rightharpoonup$$

$$fs(X \vdash at) = \dfrac{\dfrac{}{\boxed{X} \vdash at}\ CoAx}{X \vdash at}\ \leftharpoondown$$

In order to define our domain of items $\mathcal{I}$, some notion of *possible* unfolded formulas needs to be established. I propose the **sequent space** S of an unfold system U. It is built up inductively from the initial set of bottom sequents. An initial sequent is depicted as $\frac{P}{s}$, that denotes that the sequent s is the bottom of a partial proof tree P. Each initial sequent contains an n-amount of open structure variables, each of which (theoretically speaking) could be unified with any other sequent in our unfold system. However, the sequent itself can not be plugged into its own sequent, as an item is always unified with another item. I ensure this by a function $\not\subseteq$, although I leave its definition implicit. It can be read as '*is not a subterm of*'. The sequent that is actually added to the set is the sequent that can be deduced from the unified sequent by zero or more applications of the residual laws, depicted as $\frac{s}{s'}\ I^{nl}$. I^{nl} is a non-looping sequence of structural rules, for which I provide a definition below. The functions v and t are auxiliary functions, that help to overcome some of the clutter that arises from the use of tuples. v returns, given a hole h, a tuple of a structure variable and the sequent side that the variable occurs on in the sequent of h. Var denotes the codomain of structure variables. The function t receives a sequent side

io and a sequent s as input, and outputs the structure on the io side of s. $\mathcal{S}t$ denotes the set of possible structures in our system, like we saw earlier at the function w.

Definition 3.2.3 (Sequent space). *The sequent space $\mathcal{S}$ of an unfold system $U = \langle \mathcal{C}, \mathcal{O} \rangle$ is the set of sequents that can be deduced from the initial set of bottom sequents. This initial set is the set S_{init}, in which each initial bottom sequent is paired with a subset of the set of structure variables in that sequent.*

$$S_{init} = \{\langle s, h\rangle \mid \langle \frac{P}{s}, \mathcal{H}_1\rangle \in \mathcal{C} \cup \mathcal{O}, h \subseteq \{v(x) \mid x \in \mathcal{H}\}\}$$

$$v : \mathcal{H} \to (\{in, out\} \times Var)$$

$$v(\langle A \vdash B, x, \mathcal{Y}\rangle) = \begin{cases} \langle in, A\rangle & \text{if } var(A) \\ \langle out, B\rangle & \text{if } var(B) \end{cases}$$

$$S^0 = \{s' \mid \langle s, \emptyset\rangle \in S_{init}, \frac{s}{s'}\ I^{nl}\}$$

$$S^1 = \{s' \mid \langle s, \{\langle io_1, v_1\rangle\}\rangle \in S_{init}, s_1 \in S^0 \backslash \{s\}, s'_1 = t(io_1, s_1), \frac{s[v_1 := s'_1]}{s'}\ I^{nl}\} \cup S^0$$

$$\vdots$$

$$\begin{aligned} S = S^n = \{s' \mid \langle s, \{\langle v_1, io_1\rangle, ..., \langle v_n, io_n\rangle\}\rangle \in S_{init}, \\ s_1 \in S^{n-1}, s \not\subseteq s_1, s'_1 = t(io_1, s_1), ..., \\ s_n \in S^{n-1}, s \not\subseteq s_n, s'_n = t(io_n, s_n), \frac{s[v_1 := s'_1, ..., v_n := s'_n]}{s'}\ I^{nl}\} \cup S^{n-1} \end{aligned}$$

$$t : \{in, out\} \times S \to \mathcal{S}t$$

$$t(in, A \vdash B) = A$$

$$t(out, A \vdash B) = B$$

A sequence $I^{\mathbb{N}}$ of length n can be described as a function $\mathbb{N} \to I$, where each $i \leq n$ maps to an element in the sequence.

Definition 3.2.4 (Structural rule sequence). *A structural rule sequence St_n is a sequence of n structural rules, such that $n > 0$ and $St_n \in I^{\mathbb{N}}$, where $I = \{$ (1.17), (1.14), (1.18) $\}$ and $I^{\mathbb{N}}$ the set of all structural sequences. A structural rule $i \in I$ can be described as a (partial) function from a sequent to a sequent $i : S \to S$.*

Definition 3.2.5 (Structural sequent sequence). *Given a structural rule sequence St_n and a sequent s, a structural sequent sequence Sq_n is the sequence of sequents that is obtained by applying each structural rule in St_n recursively and in order to s. A sequent sequence Sq_n is said to be **looping** iff $\exists i\ .\ \exists j\ .\ i < j \leq n \wedge Sq_i(s, St_n) = Sq_j(s, St_n)$. A **non-looping** structural sequent sequence is a sequent sequence that is not looping, depicted as $(\cdot)^{nl}$.*

$$Sq : \mathbb{N} \times S \times (\mathbb{N} \times I) \to S$$

$$Sq_0(s, St) = s$$

$$Sq_i(s, St) = St_i\,(Sq_{i-1}(s, St))$$

Having established a firm foundation for our parsing system, we can move on to its actual definition. A parsing system consists of a domain of items $\mathcal{I}$, a set of axioms $\mathcal{A}$ and a set of inferences $\mathcal{R}$. $\mathcal{I}$ follows directly from the sequent space S of U. $\mathcal{A}$

consists of all unfolded formulas that are either closed or can be focus shifted. $\mathcal{R}$ corresponds to the three inference rules that I described at the beginning of this section. Parsing is said to be finished once a closed item is deduced, of which the yield must correspond with the initial input sequent. Figure 4.3 shows an example of the parsing of the sentence "*Everyone left*".

Definition 3.2.6 (The parsing system fLG_{CYK})**.** *Let an unfold system $U = \langle \mathcal{C}, \mathcal{O} \rangle$ and an input $A_1, ..., A_i \vdash B_j, ..., B_n$ be given. Let S be the sequent space of U. The parsing system $fLG_{\text{CYK}} = \langle \mathcal{I}, \mathcal{A}, \mathcal{R} \rangle$ is defined as follows:*

$$\mathcal{I} = \{\langle \pi^1 p(s), \pi^2 p(s) \rangle \mid s \in S\}$$

$$\mathcal{A} = \{\langle p, \mathcal{H} \rangle \mid \langle p, \mathcal{H} \rangle \in \mathcal{C} \ \vee \ (\langle p, \mathcal{H} \rangle \in \mathcal{O} \wedge \forall \langle s', x, \mathcal{Y} \rangle \in \mathcal{H} \ . \ fs(s'))\}$$

$$\mathcal{R} = \begin{cases} \dfrac{\left\langle \dfrac{P}{\dfrac{s}{s'}} \ i', \ \mathcal{H} \right\rangle}{\left\langle \dfrac{\dfrac{P}{s}}{\dfrac{s'}{i}} \ i', \ \mathcal{H} \right\rangle} \ res & \text{if } \exists i \in I. \ \dfrac{s}{s'} \ i \text{ and } i \neq i'^{-1} \\ \dfrac{\langle P, \mathcal{H} \rangle}{\langle P[A := at, \ x := fs(s')], \emptyset \rangle} \ fs & \text{if } \forall \langle s', x, \mathcal{Y} \rangle \in \mathcal{H} \ . \ s' = \begin{cases} A \vdash at \\ at \vdash A \end{cases} \\ \dfrac{\left\langle \dfrac{P_1}{s_1}, \mathcal{H} \right\rangle \quad \left\langle \dfrac{P_2}{A_2 \vdash B_2}, \emptyset \right\rangle}{\left\langle \dfrac{P_1[x := P_2]}{s_1'}, \mathcal{H} \backslash \langle s_2, x, \mathcal{Y} \rangle \right\rangle} \ un & \text{if } \exists \langle s_2, x, \mathcal{Y} \rangle \in \mathcal{H} \ . \ \forall y \in \mathcal{Y} \ . \ y \subseteq P_2 \\ & \text{and } \ s_1' = \begin{cases} s_1[A := A_2] & \text{if } \ s_2 = A \vdash B_2 \\ s_1[B := B_2] & \text{if } \ s_2 = A_2 \vdash B \end{cases} \end{cases}$$

3.3 Indexed fLG

Our definition so far does not take the input order of the sequent into account. The only constraint that is laid on the goal sequent is that its yield must correspond with the initial input (in the correct order). This method is quite inefficient, because many derivations are tried before it turns out the order of the final yield is not correct. For example, the non-indexed unfolded formulas of the sentences *Bob likes some teacher* and *Some teacher likes Bob* are equivalent. By keeping track of the *span* of an unfold during the parsing process, the running time can be reduced by a great amount. I will present a system here that is based on the **position pairs** that are used in (Moot, 2014).

A string w can be represented as tuple $\langle i, j \rangle$ of string positions, which means that w is the string that spans position i to j. This approach can be applied to the rules of fLG: inference rules are now manipulations of indices. A hole in an unfolded formula is then paired with an index tuple that acts as a constraint on that hole.

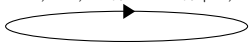
Definition 3.3.1 (Indexed structure)**.** *An indexed structure ${}^{i}A^{j}$ is a structure A that spans position i to j in the input string.*

Definition 3.3.2. *We say a grammar is* simple in the input string *if for each input string $w_1, ..., w_n$ we have that w_i spans positions $i, i+1$* (Moot, 2014)*.*

The symmetrical nature of LG forces us to take on a slightly different approach than that of Moot. A sequent in NL is linear in its input and in terms of position pairs

can be read as follows: ${}^0A_1^1, ..., {}^{n-1}A_n^n \vdash {}^0B^n$. The antecedent structure thus spans the same position as the succedent formula. In the approach for fLG, both sides of the sequent still span the same index (from 0 to n), but the lhs is split up in different sub-spans than the rhs. A sequent (that is simple in the input string) then becomes ${}^0A_1^1, ..., {}^{n-1}A_n^n \vdash {}^0B_{n+1}^1, ..., {}^{n-1}B_{2n}^n$. In a sequent that is not simple in its input string, the number of formulas on each side of the turnstile does not have to be equal, as long as the entire span of the antecedent and succedent are of the form ${}^0A^n \vdash {}^0B^n$. We could also choose for a symmetry that puts A_n adjacent to B_{n+1}:

$$A_1, ..., A_n \vdash B_{n+1}, ..., B_m$$



However, this would pose difficulties for the monotonicity rules of $\otimes$ and $\oplus$, as I show below. The co(axiom), (de)focusing, rewrite, and unary operator rules do not shift the string positions and are left implicit.

Monotonicity rules The focus of a complex formula is shifted to its subparts.

$$\frac{{}^iX^j \vdash \boxed{{}^iA^j} \quad {}^jY^k \vdash \boxed{{}^jB^k}}{{}^i\big(X \cdot \otimes \cdot Y\big)^k \vdash \boxed{{}^i\big(A \otimes B\big)^k}} \otimes R \qquad \frac{\boxed{{}^iA^j} \vdash {}^iX^j \quad \boxed{{}^jB^k} \vdash {}^jY^k}{\boxed{{}^i\big(A \oplus B\big)^k} \vdash {}^i\big(X \cdot \oplus \cdot Y\big)^k} \oplus L$$

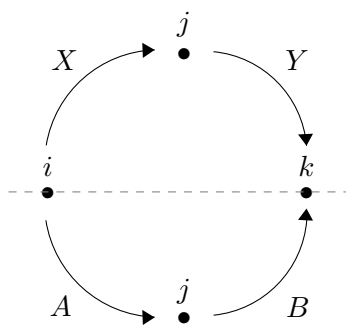
$$\frac{{}^iX^j \vdash \boxed{{}^iA^j} \quad \boxed{{}^iB^k} \vdash {}^iY^k}{\boxed{{}^j\big(A \backslash B\big)^k} \vdash {}^j\big(X \cdot \backslash \cdot Y\big)^k} \backslash L \qquad \frac{{}^iX^k \vdash \boxed{{}^iA^k} \quad \boxed{{}^jB^k} \vdash {}^jY^k}{{}^i\big(X \cdot \oslash \cdot Y\big)^j \vdash \boxed{{}^i\big(A \oslash B\big)^j}} \oslash R$$

$$\frac{{}^jX^k \vdash \boxed{{}^jA^k} \quad \boxed{{}^iB^k} \vdash {}^iY^k}{\boxed{{}^i\big(B / A\big)^j} \vdash {}^i\big(Y \cdot / \cdot X\big)^j} / L \qquad \frac{{}^iX^k \vdash \boxed{{}^iA^k} \quad \boxed{{}^iB^j} \vdash {}^iY^j}{{}^j\big(Y \cdot \oslash \cdot X\big)^k \vdash \boxed{{}^j\big(B \oslash A\big)^k}} \oslash R \tag{3.9}$$

$$\dfrac{\dfrac{{}^jY^k \vdash {}^j\big(X \cdot \backslash \cdot Z\big)^k}{{}^i\big(X \cdot \otimes \cdot Y\big)^k \vdash {}^iZ^k}\, rp}{{}^iX^j \vdash {}^i\big(Z \cdot / \cdot Y\big)^j}\, rp \qquad \dfrac{\dfrac{{}^i\big(Z \cdot \oslash \cdot X\big)^j \vdash {}^iY^j}{{}^iZ^k \vdash {}^i\big(Y \cdot \oplus \cdot X\big)^k}\, drp}{{}^j\big(Y \cdot \oslash \cdot Z\big)^k \vdash {}^jX^k}\, drp \tag{3.10}$$

The rules $\otimes R$ and $\oplus L$ act as string concatenation. Since the monotonicity rules for both these operators combine the left argument on the lhs with the left argument on the rhs (and vice versa), a cyclic reading that placed the last formula of the lhs adjacent to the first formula of the rhs would raise difficulties how the subparts of an $\otimes$ or $\oplus$ were to be combined. Graphically, the monotonicity rule of $\otimes$ could be represented as follows, where the dashed line denotes the transition from the antecedent (top part) to the succedent (bottom part).

$$\frac{{}^iX^j \vdash \boxed{{}^iA^j} \quad {}^jY^k \vdash \boxed{{}^jB^k}}{{}^i\big(X \cdot \otimes \cdot Y\big)^k \vdash \boxed{{}^i\big(A \otimes B\big)^k}} \otimes R =$$



The Grishin interaction postulates turned out to be problematic. As they add some notion of mixed associativity to our system, determining the indices of the resulting formulas is less trivial than we have seen above. $G1$ and $G3$ fit nicely into the current model, and the symmetry is maintained. As $\oslash, \otimes, /, \backslash$ can be seen as the string subtraction of the argument from the functor, $Z \cdot \otimes \cdot X$ (in the case of $G1$) then becomes the subtraction of $\langle i, l \rangle$ from $\langle i, j \rangle$ which yields $\langle l, j \rangle$. $G3$ is similar. $G2$ and $G4$ are more problematic, as Z and Y, and X and W are not adjacent in our current reading. For example, subtracting ${}^{i}Z^{l}$ from ${}^{j}Y^{k}$ (from the left with $\cdot \otimes \cdot$) forces us to either assume that $i := j$ to span $\langle l, k \rangle$, which entails that X is empty (as X spans $\langle i, j \rangle$), or that there is some non-empty part $\langle i, j \rangle$ to the left of Z. In the case of $G2$ the other side of the sequent then becomes ${}^{l}\big(X \cdot \backslash \cdot W\big)^{k}$, because X was considered to be empty:

$$\frac{{}^{i}\big({}^{i}X^{i} \cdot \otimes \cdot {}^{i}Y^{k}\big)^{k} \vdash {}^{i}\big({}^{i}Z^{l} \cdot \oplus \cdot {}^{l}W^{k}\big)^{k}}{{}^{l}\big(Z \cdot \otimes \cdot Y\big)^{k} \vdash {}^{l}\big(X \cdot \backslash \cdot W\big)^{k}}\ G2$$

We could also assume that Z is empty and that $i := l$, which would result in the following inference step:

$$\frac{{}^{i}\big({}^{i}X^{j} \cdot \otimes \cdot {}^{j}Y^{k}\big)^{k} \vdash {}^{i}\big({}^{i}Z^{i} \cdot \oplus \cdot {}^{i}W^{k}\big)^{k}}{{}^{j}\big(Z \cdot \otimes \cdot Y\big)^{k} \vdash {}^{j}\big(X \cdot \backslash \cdot W\big)^{k}}\ G2$$

X and Z could also be both empty. The problem of $G2$ and $G4$ is that is not possible in our current reading that X and Z are both non-empty. $G4$ acts in a similar manner as $G2$. We therefore have to conclude that $G2$ and $G4$ lie certain restriction on the composition of its structures. I therefore prefer to leave the indexing of the interaction postulates open for discussion. The techniques that (Moot, 2014) raises for the Displacement calculus might provide a solution for the issues that are encountered here. Figure 3.3 gives an overview of the indexed postulates. Keep in mind that $G2$ and $G4$ are subject to the assumptions I stated above.

When unfolding a hypothesis, its position in the sentence is not fixed. I therefore chose to give a hypothesis a span of two open variables. When the hypothesis is unified with another unfold, the variable indices are then unified with the indices of the other unfold. The unfolding for $(np/n) \otimes n$ then becomes:

$$\dfrac{\dfrac{\dfrac{\vdots}{{}^{i}np_6^k \vdash {}^{i}E^{k}}}{\boxed{{}^{i}np_6^k} \vdash {}^{i}E^{k}}\leftarrow \quad \dfrac{}{{}^{j}n_D^k \vdash \boxed{{}^{j}n_7^k}}}{\boxed{{}^{i}\big(np_6/n_7\big)^{j}} \vdash {}^{i}\big(E \cdot / \cdot n_D\big)^{j}}\ (/L) \qquad \leftarrow$$

$$\dfrac{{}^{i}\big(np_6/n_7\big)^{j} \vdash {}^{i}\big(E \cdot / \cdot n_D\big)^{j}}{\vdots}$$

$${}^{0}\big((np_6/n_7) \cdot \otimes \cdot n_8\big)^{1} \vdash {}^{0}A^{1}$$

$$\frac{{}^{i}\big({}^{i}X^{j}\ \cdot\otimes\cdot\ {}^{j}Y^{k}\big)^{k} \vdash {}^{i}\big({}^{i}Z^{l}\ \cdot\oplus\cdot\ {}^{l}W^{k}\big)^{k}}{{}^{l}\big(Z\cdot\oslash\cdot X\big)^{j} \vdash {}^{l}\big(W\cdot/\cdot Y\big)^{j}}\ G1 \qquad \frac{{}^{i}\big({}^{i}X^{j}\ \cdot\otimes\cdot\ {}^{j}Y^{k}\big)^{k} \vdash {}^{i}\big({}^{i}Z^{l}\ \cdot\oplus\cdot\ {}^{l}W^{k}\big)^{k}}{{}^{j}\big(Y\cdot\oslash\cdot W\big)^{l} \vdash {}^{j}\big(X\cdot\backslash\cdot Z\big)^{l}}\ G3$$

$$\frac{{}^{i}\big({}^{i}X^{j}\ \cdot\otimes\cdot\ {}^{j}Y^{k}\big)^{k} \vdash {}^{i}\big({}^{i}Z^{l}\ \cdot\oplus\cdot\ {}^{l}W^{k}\big)^{k}}{{}^{l}\big(Z\cdot\oslash\cdot Y\big)^{k} \vdash {}^{j}\big(X\cdot\backslash\cdot W\big)^{k}}\ G2 \qquad \frac{{}^{i}\big({}^{i}X^{j}\ \cdot\otimes\cdot\ {}^{j}Y^{k}\big)^{k} \vdash {}^{i}\big({}^{i}Z^{l}\ \cdot\oplus\cdot\ {}^{l}W^{k}\big)^{k}}{{}^{i}\big(X\cdot\oslash\cdot W\big)^{l} \vdash {}^{i}\big(Z\cdot/\cdot Y\big)^{j}}\ G4$$

(3.11)

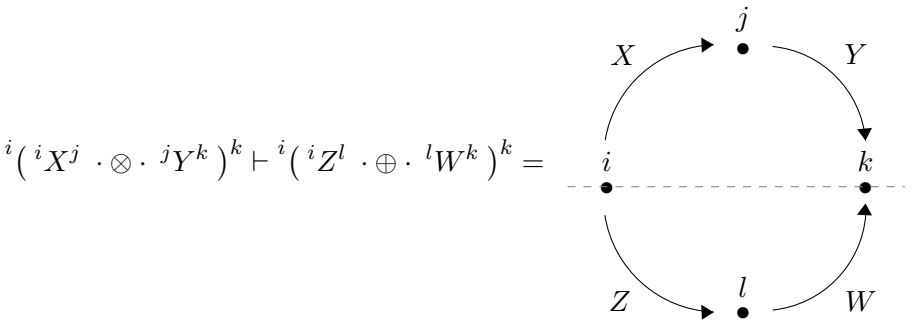


FIGURE 3.3: The Grishin interaction postulates. Note that j does not have to be equal to l, as the span of the antecedent might be divided differently than the succedent.

The postulate $P2$ gives rise to similar issues that were encountered by the Grishin interaction postulates. $P1$ is not problematic, as the order of the structures is not changed:

$$\frac{{}^{i}\big({}^{i}A^{j} \cdot\otimes\cdot {}^{j}\big({}^{j}B^{k} \cdot\otimes\cdot {}^{k}\Diamond C^{l}\big)^{l}\big)^{l} \vdash {}^{i}X^{l}}{{}^{i}\big({}^{i}\big({}^{i}A^{j} \cdot\otimes\cdot {}^{j}B^{k}\big)^{k} \cdot\otimes\cdot {}^{k}\Diamond C^{l}\big)^{l} \vdash {}^{i}X^{l}}\ P1$$

With $P2$ the $\Diamond C$ part is no longer adjacent to B: the structure has become discontinuous. This faces us again with two options: we either assume that $\Diamond C$ is a hypothesis and therefore has an empty span, or we need to adapt our indexed model to handle discontinuous structures. In my implementation I assume the former, and leave the second option open for discussion.

$$\frac{{}^{i}\big({}^{i}\big({}^{i}A^{j} \cdot\otimes\cdot {}^{x}\Diamond C^{x}\big)^{j} \cdot\otimes\cdot {}^{j}B^{k}\big)^{k} \vdash {}^{i}X^{k}}{{}^{i}\big({}^{i}\big({}^{i}A^{j} \cdot\otimes\cdot {}^{j}B^{k}\big)^{k} \cdot\otimes\cdot {}^{x}\Diamond C^{x}\big)^{k} \vdash {}^{i}X^{k}}\ P1$$

3.3.1 Parsing indexed fLG

Because the bottom sequent s of an item now contains a span ($\langle i, s, j\rangle$), our items become the triple $\langle\langle i, s, j\rangle, p, \mathcal{H}\rangle$. Our parsing system remains roughly the same as it was without the indices, except the unfolding steps now follow the updated indexed rules. I will refrain from rewriting each function that was established in the previous sections, and leave the indices implicit. The goal item of our parsing system is now defined as $\langle\langle 0, s, n\rangle, p, \emptyset\rangle$, where $\langle 0, n\rangle$ is the span of the entire initial input. Our parsing system ifLG$_{\text{CYK}}$ is defined as follows:

Definition 3.3.3 (The parsing system i-fLG$_{\text{CYK}}$)**.** *Let an unfold system $U = \langle\mathcal{C}, \mathcal{O}\rangle$ and an input $A_1, ..., A_i \vdash B_j, ..., B_n$ be given. Let S be the unfold space of U. The parsing system i-fLG$_{\text{CYK}} = \langle\mathcal{I}, \mathcal{A}, \mathcal{R}\rangle$ is defined as follows:*

$$\mathcal{I} = \ \{\langle\pi^1 p(s), \pi^2 p(s)\rangle \mid s \in S\}$$

$$\mathcal{A} = \ \{\langle p, \mathcal{H}\rangle \mid \langle p, \mathcal{H}\rangle \in \mathcal{C} \ \vee\ (\langle p, \mathcal{H}\rangle \in \mathcal{O} \wedge \forall\langle s', x, \mathcal{Y}\rangle \in \mathcal{H} \,.\, fs(s'))\}$$

$$\mathcal{R} = \begin{cases} \dfrac{\big\langle \dfrac{P}{s}\ i',\ \mathcal{H}\big\rangle}{\big\langle \dfrac{\frac{P}{s}}{s'}\ \dfrac{i'}{i},\ \mathcal{H}\big\rangle}\ res & \text{if } \exists i \in I.\ \dfrac{s}{s'}\ i \text{ and } i \neq i'^{-1} \\[2ex] \dfrac{\langle P, \mathcal{H}\rangle}{\langle P[A := at,\ x := fs(s')], \emptyset\rangle}\ fs & \text{if } \forall\langle s', X, \mathcal{Y}\rangle \in \mathcal{H} \,.\, s' = \begin{cases}\langle i, A, j\rangle \vdash \langle i, at, j\rangle \\ \langle i, at, j\rangle \vdash \langle i, A, j\rangle\end{cases} \\[2ex] \dfrac{\big\langle \dfrac{P_1}{\langle i, s_1, l\rangle}, \mathcal{H}\big\rangle \quad \big\langle \dfrac{P_2}{\langle j, A_2 \vdash B_2, k\rangle}, \emptyset\big\rangle}{\big\langle \dfrac{P_1[x := P_2]}{\langle i, s_1', l\rangle}, \mathcal{H}\backslash\langle\langle j, s_2, k\rangle, x, \mathcal{Y}\rangle\big\rangle}\ un & \text{if } \exists\langle\langle j, s_2, k\rangle, x, \mathcal{Y}\rangle \in \mathcal{H} \,.\, \forall y \in \mathcal{Y} \,.\, y \subseteq P_2 \\[2ex] & \text{and } s_1' = \begin{cases} s_1[A := A_2] & \text{if } s_2 = A \vdash B_2 \\ s_1[B := B_2] & \text{if } s_2 = A_2 \vdash B\end{cases} \end{cases}$$

3.4 Partial Proof Trees

This section provides a link between the parsing method of fLG and the Partial Proof Trees (PPTs) of Joshi and Kulick (1997). PPTs are based on lexicalized tree adjoining grammars (LTAGs) (Joshi and Schabes, 1997), a formalism I will elaborate on briefly.

Tree adjoining grammars (TAGs) are tree rewriting systems. They consist of a finite set of *elementary* trees that are combined to form larger trees. This can be done by substitution (where a leaf is substituted by another tree) or adjunction (where a node in a tree is replaced by another tree). In order to adjoin a tree into another tree, it must be an *auxiliary* tree. This means that it contains exactly one leaf that is marked as the foot node (marked by an *), which has the same label as the root node. A tree without such a foot node is called an *initial* tree. In an LTAG, each tree is associated with a lexical item, which is called the *anchor* of the tree. LTAGs are able to describe several phenomena that are found in natural languages and can be parsed in polynomial time.

PPTs are based on several principles of LTAGs, such as the adjunction operation. PPTs, however, perform manipulations on proof trees, instead of simple phrase structure trees. This is the first similarity between PPTs and the parsing system for fLG. PPTs are divided into 2 distinct phases that together form a hybrid logic: a construction phase (unfolding) and a combination phase (unification). This is similar to what we have seen before.

A PPT is a λ-term that ranges over a proof tree. A variable denotes an unfilled assumption in the proof tree, and can be compared to the open structure variables in an unfolded formula. λ-abstractions are annotated with a label l or r, to constraint the position of the argument (left or right). PPTs are paired with a type $\texttt{judg}\ \alpha$, specifying the category of the proof. I will illustrate this with two examples, and compare them with similar unfoldings in the system for fLG.

PPTs are manipulated by three different operations. I will describe them briefly, and compare each operation with a similar method for fLG.

1. **Application**: The conclusion of a PPT is linked with the assumption of another PPT. This is similar to the basic unification of fLG. PPTs restrict the direction of its arguments, a feature that is comprised by the fLG system by the introduction of indices.

2. **Stretching**: An interior node of a PPT is stretched up by inserting another PPT; this is very much alike to the adjunction of TAGs. Not only the labels of the node and the auxiliary PPT must match: they must also coincide on any undischarched assumptions (Joshi, Kulick, and Kurtonina, 1998). The stretching of nodes is not an operation in the fLG system: this is handled slightly different. In general, a PPT that is stretched by another PPT on a node of type A, would be equivalent to the unification of an unfolded formula that contains A as (sub)type with an unfolded formula of type A/A or $A\backslash A$. In the case of fLG where an unfolded formula u_1 is unified with another unfolded formula u_2 that contains hypotheses, all the proofs of the hypotheses of u_2 must be a subterm of u_1. This is, however, the only constraint on unification. It is shown that a sentence such as "*...book that John wrote and Bob read Ulysses*" is not valid in the PPT system (due to undischarged assumptions), whereas the (current) fLG system lacks the expressivity to disallow such a sentence.

In the PPT case of `likes [NP] passionately`, the `NP\S` node of `likes` would be stretched, to insert the proof tree of `passionately` . Items in our system for fLG do not need te be stretched open in this case: to handle `likes [NP] passionately`, the conclusion of `likes` (after residuating) is simply unified with the hole of passionately (figures 3.4 & 3.5). Note that the unfolded formulas are presented as partial proof trees in which a hole is represented by vertical dots.

Joshi et al. describe that phrases with two adjectives such as "*beautiful blue sky*" would give rise to a form of spurious ambiguity. The PPT of *beautiful* could be stretched by the PPT of *blue* first, or the the PPT of *sky* could just be applied to the PPT of *blue* and then to the PPT of *beautiful*. This ambiguity would occur only at the composition of the trees: the two trees itself are equivalent. This kind of spurious ambiguity is not possible in fLG, as $(n/n) \cdot \otimes \cdot (n/n) \not\vdash n/n$.

3. **Interpolation**: Similar to stretching, but this operation is already added to the PPT during the unfolding phase and therefore an obligatory inference. Furthermore, the PPT that is inserted by stretching has the same conclusion and argument node, whereas the interpolated node can have a different conclusion.

   To achieve this in the fLG system, we would *lift* the type that is interpolated. If we were to interpolate a type A to a type B, the type in fLG would then become $(B/A)\backslash B$, or $B/(A\backslash B)$, depending on the direction of the argument. A would then be a hypothesis of this lifted type. Figure 3.6 shows an example of a PPT that contains interpolation, and an equivalent fLG unfold.

Trace assumptions in PPTs are represented similar to hypothetical reasoning. A trace assumption must be discharged within the PPT in which it is assumed. An assumption that is discharged can be expected on either side of the formula, which is an example of the *extended domain of locality* of the PPTs. In our fLG system, assumptions are created during the unfolding phase, and their unfolding must be a subterm of the complete proof once it is completely unified. Figure 3.7 shows the unfolding of $(s/(np\backslash s))\backslash s$, an example which was shown earlier during the unfolding method of Hendriks.
The link that has been shown between the two formalisms is an interesting connection, as it might provide an insight into the complexity of parsing fLG, as the PPTs inherit polynomial parsing properties from the LTAG system. This link could be studied in more detail. For example, the way PPTs constrain stretching on the basis of undischarged hypotheses might be implemented into the framework for fLG.

$$\lambda_{l1}x.\lambda_{r1}y.\left(\dfrac{x \quad \dfrac{\dfrac{\texttt{likes}}{\texttt{(NP\textbackslash S)/NP}} \quad y}{\texttt{NP\textbackslash S}}}{\texttt{S}}\right) : \{l1 \Rightarrow \texttt{(judg NP)}, r1 \Rightarrow \texttt{(judg NP)}\} \rightarrow \texttt{(judg S)}$$

$$\dfrac{\dfrac{\dfrac{\overline{X \vdash \boxed{np_0}} \quad \overline{\boxed{s_1} \vdash Y}}{\boxed{np_0 \backslash s_1} \vdash X \cdot \backslash \cdot Y}\,(\backslash L) \quad \overline{Z \vdash \boxed{np_2}}}{\boxed{(np_0 \backslash s_1)/np_2} \vdash (X \cdot \backslash \cdot Y) \cdot / \cdot Z}\,(/L)}{(np_0 \backslash s_1)/np_2 \vdash (X \cdot \backslash \cdot Y) \cdot / \cdot Z}\,\leftharpoondown$$

FIGURE 3.4: The unfoldings for `likes`. Note that the fLG unfolding does not contait any holes: atomic formulas are only unified at the end of the parsing process.

$$\lambda_{l1}x.\left(\dfrac{x \quad \dfrac{\texttt{passionately}}{\texttt{(NP\textbackslash S)\textbackslash(NP}^{*}\texttt{\textbackslash S)}}}{\texttt{(NP}^{*}\texttt{\textbackslash S)}}\right) : \{l1 \Rightarrow \texttt{(judg NP\textbackslash S)}\} \rightarrow \texttt{(judg NP\textbackslash S)}$$

$$\dfrac{\dfrac{\dfrac{\vdots}{X \vdash np_0 \backslash s_1}}{X \vdash \boxed{np_0 \backslash s_1}}\,\rightharpoondown \quad \dfrac{\overline{Y \vdash \boxed{np_2}} \quad \overline{\boxed{s_3} \vdash Z}}{\boxed{np_2 \backslash s_3} \vdash Y \cdot \backslash \cdot Z}\,(\backslash L)}{\boxed{(np_0 \backslash s_1)\backslash(np_2 \backslash s_3)} \vdash X \cdot \backslash \cdot (Y \cdot \backslash \cdot Z)}\,(\backslash L)}{(np_0 \backslash s_1)\backslash(np_2 \backslash s_3) \vdash X \cdot \backslash \cdot (Y \cdot \backslash \cdot Z)}\,\leftharpoondown$$

FIGURE 3.5: The unfoldings for `passionately`. The * in the PPT unfolding indicates that this category is not unfolded. Note the similarity between the holes in both unfoldings.

$$\lambda_{i1}z.\lambda_{l1}x.\left(\dfrac{x \qquad \dfrac{\texttt{walk(inf)}}{\dfrac{\texttt{NP\textbackslash Sinf}}{\texttt{NP\textbackslash S}}\,(z)}}{\texttt{S}}\right) : \begin{cases} i1 \Rightarrow (\{r1 \Rightarrow \texttt{(judg NP\textbackslash Sinf)}\} \rightarrow \texttt{(judg NP\textbackslash S)}), \\ l1 \Rightarrow \texttt{(judg NP)} \end{cases} \rightarrow \texttt{(judg S)}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{A \vdash (np_0 \cdot \backslash \cdot s_1) \cdot / \cdot (np_2 \backslash s_{inf3})}}{A \vdash \boxed{(np_0 \backslash s_1)/(np_2 \backslash s_{inf3})}}\rightharpoonup \quad \dfrac{\overline{np_B \vdash \boxed{np_4}} \quad \overline{\boxed{s_5} \vdash s_C}}{\boxed{np_4 \backslash s_5} \vdash np_B \cdot \backslash \cdot s_C}\,(\backslash L)}{\boxed{((np_0 \backslash s_1)/(np_2 \backslash s_{inf3})) \backslash (np_4 \backslash s_5)} \vdash A \cdot \backslash \cdot (np_B \cdot \backslash \cdot s_C)}\,(\backslash L)}{((np_0 \backslash s_1)/(np_2 \backslash s_{inf3})) \backslash (np_4 \backslash s_5) \vdash A \cdot \backslash \cdot (np_B \cdot \backslash \cdot s_C)}\leftharpoondown$$

$$\dfrac{\dfrac{\overline{np_E \vdash \boxed{np_2}} \quad \dfrac{\dfrac{\vdots}{s_{inf3} \vdash G}}{\boxed{s_{inf3}} \vdash G}\leftharpoondown}{\boxed{np_2 \backslash s_{inf3}} \vdash np_E \cdot \backslash \cdot G}\,(\backslash L)}{np_2 \backslash s_{inf3} \vdash np_E \cdot \backslash \cdot G}\leftharpoondown$$

FIGURE 3.6: The unfolding for `walk`. In the fLG unfolding `walk` is considered to be the type $np\backslash s_{inf}$ that is lifted by $np\backslash s$.

$$\lambda_{l1}x.\left(\dfrac{x \qquad \dfrac{\dfrac{[\texttt{NP\textbackslash S}]^1}{\texttt{S}}}{\texttt{S/(NP\textbackslash S)}}\,[1] \qquad \dfrac{}{\texttt{(S/(NP\textbackslash S))\textbackslash S}}}{\texttt{S}}\right) : \{l1 \Rightarrow \texttt{(judg NP)}\} \rightarrow \texttt{(judg S)}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{A \vdash s_0 \cdot / \cdot (np_1 \backslash s_2)}}{A \vdash \boxed{s_0/(np_1 \backslash s_2)}}\rightharpoonup \qquad \overline{\boxed{s_3} \vdash s_B}}{\boxed{(s_0/(np_1 \backslash s_2)) \backslash s_3} \vdash A \cdot \backslash \cdot s_B}\,(\backslash L)}{(s_0/(np_1 \backslash s_2)) \backslash s_3 \vdash A \cdot \backslash \cdot s_B}\leftharpoondown \quad \dfrac{\overline{np_D \vdash \boxed{np_1}} \quad \overline{\boxed{s_2} \vdash s_E}}{\boxed{np_1 \backslash s_2} \vdash np_D \cdot \backslash \cdot s_E}\,(\backslash L)}{np_1 \backslash s_2 \vdash np_D \cdot \backslash \cdot s_E}\leftharpoondown$$

FIGURE 3.7: The unfolding for $(s/(np\backslash s))\backslash s$. The structure variables that are part of a (co)axiom are already unified with their corresponding atomic formula. Note that `NP\S` is the hypothesis in both unfoldings.

Chapter 4

A logic programming implementation

This chapter provides an implementation of the parsing system of chapter 3, written in the logic programming language Prolog. The declarative nature of Prolog turned out to be perfectly suited for the deductive parsing method, and the way unification is handled in Prolog is simple and elegant. Most of the programming methods that are used can be found in (Pereira and Shieber, 2002) and (O'Keefe, 1990).

4.1 Overview of the parser

Parsing is once again divided into two distinct phases: the **unfolding** phase and the **parsing** phase. These two phases are combined in the predicate `parse/3`, that takes an initial focus and a list of formulas or lexical items as input and output: the symmetrical nature of fLG permits us to parse both sides of the sequent. All input formulas are unfolded, and added to a list of closed and open items. A goal is kept separate if the input or output list contains only one item. The unfolding of this goal is only added to the parsing system if all the other items have been unified, and one closed item has been obtained: this reduces the running time by quite an amount. Parsing itself is established by the complete traversal of the proof space that is created by Prolog's backtracking mechanism. This will be addressed more extensively in section 4.3. All possible proofs are collected in a list by the built-in `findall` predicate, and this list of proofs is converted to a LaTeX-proof. The code for this final step is omitted in the appendix, as it is largely a copy of the code by Moortgat.

4.2 Unfolding

Recall that an unfolded formula in the indexed system was defined as a tuple $\langle p, \mathcal{H} \rangle$, with p the partial proof tree, and $\mathcal{H}$ the set of holes. In our implementation, proof trees are represented as combinator proof terms. A combinator proof is a simple term representation of the deductive steps of a parse tree, where each inference is denoted by a single term. An inference that consists of two premises becomes a term with two arguments. The sequents in a proof tree are not represented at each step in a combinator proof, but can be obtained by forward chaining from the axioms of a proof towards a goal. In our system, the bottom sequent of a proof is therefore added to an unfolded formula as a separate argument. In Prolog a tuple is represented by a `-` symbol. An unfolded formula then becomes an object of the form `VDash-Proof-Holes`, where `VDash` is the bottom sequent of the proof.

The indices of the sequent are attached to the antecedent and the succedent of a sequent: `vdash(F, I-A-J, I-B-J)`, where `F` is the focus (0, l, or r). A hole was defined as a triple $\langle\langle i, s, j\rangle, v, \mathcal{Y}\rangle$, with $\langle i, s, j\rangle$ the open sequent, v the open variable in the proof p and $\mathcal{Y}$ the set of hypothesis constraints. In Prolog this becomes `VDash-Holes-Hyps`. Sets are simply represented as lists. Unfolding is done by the predicate `unfold/4`, that takes the sequent side (*in* or *out*) and a formula as input, and outputs the unfolded formula, its unfolded hypotheses, and a list of axiom links. An axiom link is a pair of atom labels that are part of a (co)axiom in the proof tree.

Unfolding closely follows the mathematical definition that was established in chapter 3. The predicate `unfold/4` is similar to the function u (3.2). The function s (3.6) is defined as the predicate `init_struct`. The structure variables that are instantiated at this point are represented as normal Prolog variables. This predicate also labels the atomic formulas, using the dynamic predicate `at_label/1`. Labelling could be done non-dynamically, but this method provides clarity by not having to pass on an argument of the current label.

The functions g (3.4) and p (3.7) are combined in the predicate `g/6`. `g` is deterministic, i.e. once the unfolding process is completed it is not possible to backtrack to find other possible unfoldings. `g` takes a the sequent side that is unfolded and a sequent as input arguments. It outputs the proof and a list of holes (similar to the function p), a list of axiom links and a list of unfolded hypotheses (which is part of the function u). The rules of fLG are encoded as predicate rules in the program. The left focusing rule, for example, is represented as follows:

$$\frac{\boxed{A} \vdash Y}{A \vdash Y} \swarrow \quad \text{Where } A \text{ is negative}$$

```
g(Pol, vdash(0, A, Y),fl1(Proof),Links, Holes, HypUnfolds) :-
    \+var(A),pol(A,-),
    g(Pol, vdash(l, A, Y),Proof, Links, Holes, HypUnfolds).
```

Once a (co)axiom is reached, the labels of the atoms are added to the list of axiom links. Furthermore, if one of the formulas is some structure variable `X`, this variable is unified with the term `At/N`, where `At` is the atomic formula and `N` the (yet unbound) label variable that is added to the axiom links. This ensures that the original variable `X` in the bottom sequent is unified with `At/N` too, which prevents a lot of unnecessary unifications later in the procedure.

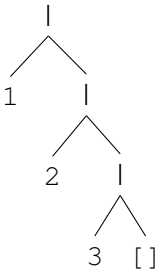
If at some point a formula is encountered that has its monotonicity rule on the other side of the sequent, it is transformed into its structural counterpart. If during this rewriting process any formula is encountered that is situated on its logical it is unfolded and its unfolding is added to a list of hypotheses. This is similar to the function w, and is done by the predicate `findall_unfolds`. A hole is created by the following line, where `HypConsts` is a list of combinator proofs:

```
make_hyp_hole(Struct, ProofHole, HypConsts, [Struct-ProofHole-
    HypConsts|H]-H).
```

4.3 Unification of unfolded formulas

In the above predicate, `ProofHole` is a Prolog variable and is instantiated on two different positions: as a variable in the proof tree, and as the variable in the hole that corresponds with the variable in the proof tree. This is powerful technique, as these two variables are now linked during the computation. If at any point one of the variables is unified, the other is too. This technique stems from the concept of **difference lists**. A list can be viewed as a binary tree:

`[1,2,3] = [1|[2,3]] = [1|[2|[3]]] = [1|[2|[3|[]]]] =`



The empty list acts as the neutral element of the list building operator `|`. In a difference list, this neutral element is replaced by a variable. By keeping a separate instance of this variable it is possible to unify this variable by another difference list. List concatenation then simply becomes the unification of two lists, and the variable of the second list is now the new variable of the tail of the list. The predicate `conc` exemplifies this concept.

```
conc(A-B, B-C, A-C).
?- conc([1, 2|X]-X, [3, 4|Y]-Y, Z).
X = [3, 4|Y],
Z = [1, 2, 3, 4|Y]-Y.
```

This concept can be applied to other data structures too, and is essential to our partial proof trees. A hole was defined as a triple, in which the second element is a variable that also occurs in the proof tree. Unifying this variable with some other proof automatically plugs that proof into the position of the variable.

4.4 Parsing

The parsing of the items is done by the predicate `unify_all_open/5`. Its first argument is an index that is used for focus shifting, a feature that I address later. The other three input arguments are the closed items, the open items, and the final goal unfolding. Closed items are represented as tuples of the form `Bottom-Proof`: the hole argument is omitted because a closed item does not contain any holes. The unfolded atomic formulas are not part of the parsing process: they are unified at the end once a final item is reached. Figure 4.3 showcases the process for the sentence "*everybody left*".

The parsing process can be divided into several distinct cases, based on the list of closed and open items. There are three different recursive cases:

1. The basic recursive case takes the head element from the list of closed items, and residuates the bottom sequent of this item until it can be unified with an item in the list of open items. Because each closed item has to be unified with some open item at some point in the computation, and the number of holes

is decreasing (i.e. no new holes are created during the parsing phase), we can necessitate the unification of an item as soon as it is closed. Once the closed item is unified with an open item, it is added to the list of closed items if there are no remaining holes. Otherwise it is appended to the end of the list of open items.

The residuation and unification of an item is done by the predicate `unify_open`. Unification is done by the built-in `select/3` predicate. This is quite elegant: by selecting the closed item that is to be unified, it is instantly unified with a corresponding hole from a list of holes. The unification step is therefore done simultaneously with the selection of a corresponding hole, a concept that is based on the partial execution as described by (Pereira and Shieber, 2002) in section 6.4. If no open item is found that can be unified, the closed item is residuated following one of the structural rules that are defined by the `rule` predicate.

2. It is possible that at some point during the computation there are no closed items available. This is caused by the polarity bias of the atomic formulas. See for example the unfolding of a transitive verb with s positive and np negative. If the polarity would be reversed, the item would not contain any holes at all.

$$\dfrac{\dfrac{\dfrac{\vdots}{A \vdash np_0}}{A \vdash \boxed{np_0}} \rightarrow \quad \dfrac{\dfrac{\vdots}{s_1 \vdash B}}{\boxed{s_1} \vdash B} \leftarrow}{\boxed{np_0 \backslash s_1} \vdash A \cdot \backslash \cdot B} (\backslash L) \quad \dfrac{\dfrac{\vdots}{C \vdash np_2}}{C \vdash \boxed{np_2}} \rightarrow}{\dfrac{\boxed{(np_0 \backslash s_1)/np_2} \vdash (A \cdot \backslash \cdot B) \cdot / \cdot C}{(np_0 \backslash s_1)/np_2 \vdash (A \cdot \backslash \cdot B) \cdot / \cdot C} \leftarrow} (/L)$$

Focus shifting is a method that has been described in chapter 3. It is defined as the predicate `focus_shift` in the code. Not every open item that can be focus shifted, needs to be completely focus shifted! The open holes can be unified with another item too, after all. This leads us to the final recursive step, that is closely connected to focus shifting.

3. If there are no closed items available, we might want to skip the focus shifting of an open item. However, to prevent spurious ambiguity we want to keep track of the items that have been skipped. In the code this is done by the index `N`. Updating this index turned out to be surprisingly complex. At the start of parsing it is instantiated to be `0`. This indicates that the 0^{th} open item can still be focus shifted. Once an open item is skipped, the index is incremented and the next open item is selected using the built-in predicate `nth0/4`. If at some point a focus shifted item is unified with an open item that contains multiple holes and has been skipped from focus shifting earlier on, we need to add this item to the end of the list of open items! This is due to the fact that focus shifting is only possible if all holes are shifted, and now that one hole has been unified with another item we might want to focus shift the remaining holes. Figure 4.1 exemplifies this whole procedure.

The following six cases all mark a final step of the unfolding process. Each case is based on the composition of the lists of open and closed items.

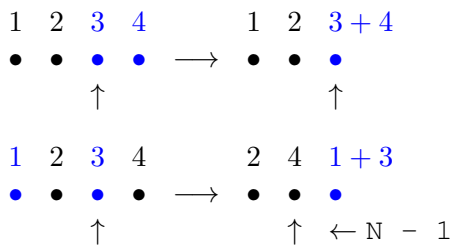


FIGURE 4.1: Two different cases for updating the index of the list of open items that have been focus shifted. ↑ denotes the current index. In the first case, the item that is focus shifted (3) is unified with an open item that still can be focus shifted. In the second case, (3) is unified with (1), an item that has been skipped for focus shifting. However, we allow this new item to be focus shifted again, as the remaining holes might need to be shifted. The index is decremented, as the first item of the list is now removed.

1. Both the lists are empty. This occurs when parsing a sequent such as $np \vdash np$, because unfolded atomic formulas are omitted from the parsing process.

2. One closed item, an empty list of open items and no goal item. When parsing two lists that both contain more than one item, there is no goal formula that is kept separate. The final closed item is residuated until a sequent of the form $X \cdot \otimes \cdot Y \vdash Z \cdot \oplus \cdot W$ is retrieved.

3. No closed items, one open item and no goal item. Similar to the previous case, but now the open item must be focus shifted before we can residuate it. I will address focus shifting in a later case.

4. One closed item, no open item and an open goal. The standard final step: the closed item is simply unified with the open goal.

5. No closed items, one open item and one goal. In this case either the open item or the open goal is focus shifted. The closed item that is returned by focus shifting is then unified with the remaining open item.

6. No closed items, one open item, and one closed goal. The closed goal is unified with the open item.

The final item is represented as `vdash(0,0-A-n,0-B-n)-FProof`. Once such an item has been obtained, the predicates `unify_in` and `unify_out` walk through the proof term and unify the atomic formulas that were omitted from the parsing process.

The entire unification phase is performed in a *cut-free* manner. The cut is a powerful tool, but if we want our program to rely on the backtracking mechanism of Prolog, we can't use it at all times. By never using a cut at any time during unification, a proof search space is created automatically by Prolog's backtracking mechanism, in a depth-first search fashion. This search space can be optimized by a great amount, as it can be seen easily that a lot of unnecessary partial parses are tried during parsing. I propose several ways to optimize the parsing process.

4.5 Parsing efficiency

4.5.1 Loop detection

The first method concerns the way loop detection is handled. Since the residuation rules are invertible, it is possible to get stuck in a loop. We could maintain a list of visited sequents, and at each step check whether we have already visited that sequent. This is a time-consuming task, and I propose a more efficient method. Simply by checking whether the previous rule is not the inverse of the rule that is currently checked, it is impossible to get stuck in a loop. This can be proved, based on the definition of structural sequent sequence that is described in section 3.2.

Lemma 4.5.1. *A structural sequent sequence $Sq_n(s, St_n)$ is looping on positions i and j, where $i < j$, if and only if the sequence in St_n from position $i + 1$ to j is a pattern of nested rules, i.e. each rule is closed by its inverse in a nested manner.*

Proof. Let St_n be a structural rule sequence of length n. Let s be a sequent and Sq_n the structural sequent sequence for s and St_n. The inverse of the inverse of a structural rule is considered to be that structural rule itself, i.e. $I^{-1^{-1}} = I$. The postulates P1 and P2 can not be inverted. The Grishin interaction postulates are non-invertible too. Therefore, any structural sequence that contains at least one of these postulates is non-looping, as there aren't any other rules in our system that contain associativity.

$\Longrightarrow$ As Sq_n is looping, there exist an i and j, such that $i < j$ and $Sq_i(s, St_n) = Sq_j(s, St_n)$.

The structural rules can be divided into 2 distinct groups: (1) the rules that shift a structure from the lhs to the rhs, and (2) the rules that do this in the opposite way. A rule from group (1) can only be followed by its own inverse, or a rule from group (1); for group (2) vice versa. Each rule from either group peels off a structure from one side of the sequent and adds it to the other, and the only way a rule from one group can be followed by a rule from the other group is by its own inverse. Therefore, the only way to obtain the same sequent after n steps is by n applications of the inverses of the first n rules, in a reversed order. During the application of these inverted rules it is possible to apply a rule that is not an inverse, but this rule must then be inverted later on too to arrive at the original sequent. This creates a nested pattern of rules and their inverses:

$$
\begin{aligned}
S &::= L \mid R \mid S\ S \mid \epsilon \\
R &::= r\ R\ r^{-1} \mid \epsilon \qquad \textit{where } r \in \textit{group } (1) \\
L &::= l\ L\ l^{-1} \mid \epsilon \qquad \textit{where } l \in \textit{group } (2)
\end{aligned}
$$

$\Longleftarrow$ St_n forms a nested pattern of rules from position $i + 1$ to j. That means that for each rule there is an inverted rule later in the sequence. If we were to apply each rule in the sequence to $Sq_i(s, St_n)$, it would either shift a structure to the rhs or the lhs. As the pattern is nested, each structure shift is undone later in the sequence by its inverse, and after applying all the rules, we would obtain the same sequent on position j, and thus the sequent is looping from position i to j.

□

Corollary 4.5.1.1. *If a structural sequent sequence is looping, then there is at least one point in the corresponding structural rule sequence where a rule is immediately followed by its inverse.*

This follows directly from the nested nature of a looping sequence: a left-shifting pattern ends when $L ::= \epsilon$, which yields a pattern $...l\ l^{-1}...$. A right-shifting pattern yields at its centre: $...r\ r^{-1}...$.

To prevent a loop, we can therefore simply check if the previous rule was the inverse of the current rule.

4.5.2 Unification constraints

I implemented a method for constraining the unification of the atom labels. Take, for example, the unfolding of $(np\backslash s)/np$ (with s negative and np positive):

$$\dfrac{\dfrac{\dfrac{\overline{np_A \vdash \boxed{np_3}} \quad \overline{\boxed{s_4} \vdash s_B}}{\boxed{np_3\backslash s_4} \vdash np_A \cdot \backslash \cdot s_B}(\backslash L) \quad \overline{np_C \vdash \boxed{np_5}}}{\boxed{(np_3\backslash s_4)/np_5} \vdash (np_A \cdot \backslash \cdot s_B) \cdot / \cdot np_C}(/L)}{(np_3\backslash s_4)/np_5 \vdash (np_A \cdot \backslash \cdot s_B) \cdot / \cdot np_C}\leftarrow$$

A priori we can already tell that A will be unified with some i that is smaller than 3, and C with some j that is larger than 5. If we were to combine this item with the unfolding of np_0/n_1 in a sentence such as "*Some teacher likes Bob*", then np_0 can not be unified with np_5, as this would yield the sentence "*Bob likes some teacher*". This constraint can be added easily using the built-in `freeze` predicate that takes a variable and some Prolog goal as its arguments. Once the variable is unified, the goal is called. By adding the constraints to `freeze`, and not to some list that is passed on, the *extra-logical* nature of such constraints is kept separate from the logical flow.

This method improved the running time by a significant factor. For example, the parsing of the sentence "*Everyone thinks some teacher likes Bob*" (with three different proofs) costs 13,463 inferences without the label constraints, and only 3,505 inferences with constraints[1]. Unfortunately, these constraints bring quite some issues with them. First of all, if we were to add the postulates P1 and P2 to our system, it is now possible to manipulate the structure of our input. It is then no longer necessary for an atom to come from the expected direction. This can be solved by attaching a special label to an argument of a $\Diamond\Box$ construction, for which a constraint is ignored. The case that made me convinced that this method is not optimal is the proof of

[1]For comparison, backward chaining this sentence (as a structured sentence tree) costs 1,718 inferences.

$(s_0/(np_1/n_2)) \cdot \otimes \cdot (np_3/n_4) \vdash s_5$:

$$\dfrac{\dfrac{\boxed{s_0} \vdash s_5 \qquad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{np_3 \vdash \boxed{np_1}}\rightarrow}{np_3 \vdash np_1}\leftarrow}{\boxed{np_3} \vdash np_1} \qquad \dfrac{}{n_2 \vdash \boxed{n_4}}}{\boxed{np_3/n_4} \vdash np_1 \cdot / \cdot n_2}(/L)}{np_3/n_4 \vdash np_1 \cdot / \cdot n_2}\leftarrow}{np_3/n_4 \vdash np_1/n_2}(/R)}{np_3/n_4 \vdash \boxed{np_1/n_2}}\rightarrow}{\boxed{s_0/(np_1/n_2)} \vdash s_5 \cdot / \cdot (np_3/n_4)}(/L)}{s_0/(np_1/n_2) \vdash s_5 \cdot / \cdot (np_3/n_4)}\leftarrow$$

The unfolding of np_3/n_4 would impose a constraint on the label that is matched with n_4. n_4 is, however, matched with n_2, as the whole np_3/n_4 is an argument of $(s_0/(np_1/n_2))$. This could be solved again by adding special labels to (np_1/n_2), but that would only push us further from an elegant and generalized solution for our constraints.

4.5.3 Indexed fLG

The indexed fLG system that was introduced in section 3.3 provided a major improvement in the running time of the parsing algorithm. Without the restriction on the span of a hole, any possible combination is checked: only the `unify_in` and `unify_out` predicates restricted the order of the final goal item. Adding the indices to my code turned out to be an easy task. The structure of the parser remained intact: only the unfolding and residuation rules had to be adjusted. Unfolding now takes an extra argument that represents the span of the formula that is unfolded. When unfolding a hypothesis, this span consists of two unbound variables. Figure 4.2 shows the unfolding of *thinks*, generated by the parser. Figure 4.3 shows the unfolding and unification of the sentence "*everyone left*".

$$\dfrac{\dfrac{\dfrac{\dfrac{}{np_C \vdash \boxed{np_0}} \quad \dfrac{}{\boxed{s_1} \vdash s_D}}{\boxed{{}^{A}np_0^3 \setminus {}^{A}s_1^B} \vdash {}^{A}np_C^3 \cdot \setminus \cdot {}^{A}s_D^B}(\setminus L) \quad \dfrac{\vdots}{{}^{4}E^B \vdash {}^{4}s_2^B}}{E \vdash \boxed{s_2}}\rightarrow}{\boxed{{}^{3}\big(({}^{A}np_0^3 \setminus {}^{A}s_1^B)\big)^B / {}^{4}s_2^B} \vdash {}^{3}\big(({}^{A}np_C^3 \cdot \setminus \cdot {}^{A}s_D^B)\big)^B \cdot / \cdot {}^{4}E^B}(/L)}{{}^{3}\big(({}^{A}np_0^3 \setminus {}^{A}s_1^B)\big)^B / {}^{4}s_2^B \vdash {}^{3}\big(({}^{A}np_C^3 \cdot \setminus \cdot {}^{A}s_D^B)\big)^B \cdot / \cdot {}^{4}E^B}\leftarrow$$

FIGURE 4.2: The unfolding of *thinks*, with type $(np\backslash s)/s$. np is positive, s is negative.

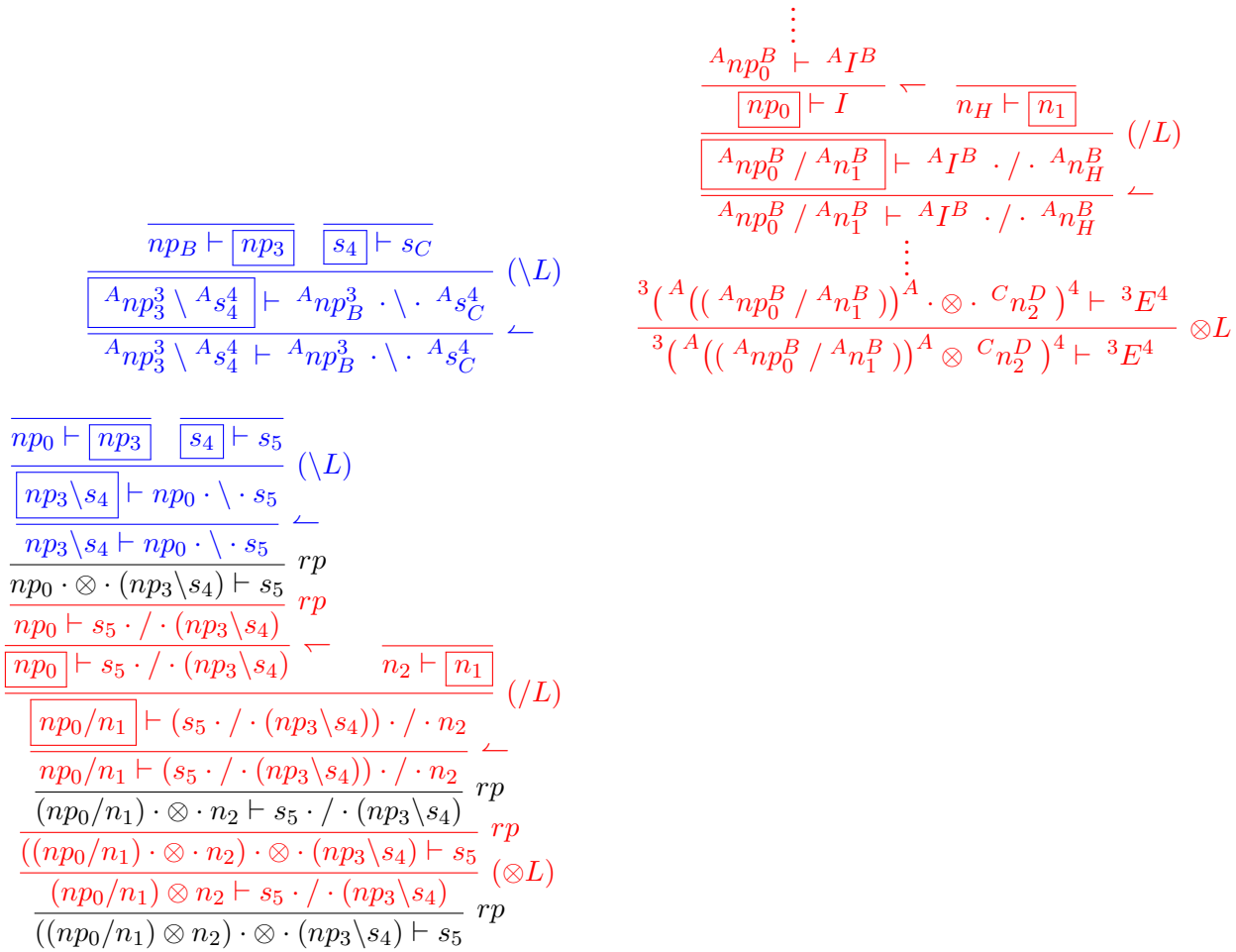


FIGURE 4.3: The unfolding and unification for the sentence "*everyone left*", where *everyone* has the type $(np/n) \otimes n$ and *left* $np\backslash s$. Note that the unfoldings (in red and blue) are connected by residual rules (in black). The hypothesis of *everyone* is connected by a residual rule too.

Chapter 5

Conclusion

I have provided a parsing method for the focused Lambek-Grishin calculus that is able to parse an unstructured input sequent in a bottom-up fashion. We achieved this by dividing the parsing procedure into an unfolding and a parsing phase. The whole procedure was presented as a parsing system, a formal notion that enabled us to define the parsing steps in a clear and abstract manner. In the unfolding phase each logical formula is transformed into a partial proof tree: a proof tree in which some branches can be unified with other proof trees. This unfolding phase can be computed independently of the parsing phase. The unfolding of a formula is a deterministic process, which means that each formula can only be assigned to one partial proof tree. After each formula has been unfolded, the obtained items are combined in the parsing phase. This is done bottom-up: at each step we either combine two items if they are eligible for unification, or we manipulate the partial proof tree of an item by an inference rule in our logical system.

The method that was presented in section 3.2 did not take the input order into account during parsing. Only at the end of the whole procedure it was checked whether the derived logical structure corresponded with the initial input string. Therefore I presented a system in which the span of a formula in the input string was part of the unfolding and parsing process. This approach lead to a significant amelioration of the efficiency of our parser.

The parsing system of chapter 3 was implemented in the logic programming language Prolog. This implementation roughly follows the same parsing steps that were defined formally. The declarative nature of Prolog turned out to be well-suited for our deductive parsing approach, and the unification mechanisms of the language provided an elegant solution to the combination of two unfolded formulas.

There is, however, a considerable amount of work left that needs to be finished. First of all, an analysis of the time complexity of our system would provide more insight into the complexity of a parsing procedure for the Lambek-Grishin calculus. The soundness and completeness of the system should both be proved. The presentation of our parsing method as a formally defined parsing system will provide a solid foundation for proving the correctness of our system. The link between the Partial Proof Trees of section 3.4 might provide an insight into the properties of parsing system. The parsing procedure was presented as a form of CYK parsing, which means that a form of tabulation might be added to the system to increase its efficiency. Extending the inferences of the calculus with the indexed span of a formula left some issues open for discussion. The non-continuous nature of LG turned out to be problematic for our indexed extension. By raising these issues, I aim to facilitate the development of a complete indexed system.

Bibliography

Andreoli, Jean-Marc (1992). "Logic Programming with Focusing Proofs in Linear Logic". In: *Journal of Logic and Computation* 2.3, p. 297.

Bastenhof, Arno (2010). "Tableaux for the Lambek-Grishin calculus". In: *CoRR* abs/1009.3238.

Benthem, Johan van (1983). "The semantics of variety in categorial grammar". In: *Technical Report* 83.29. Revised version in W. Buszkowski *et al*. (1988).

Bransen, Jeroen (2012). "The Lambek-Grishin calculus is NP-complete". In: *Formal Grammar*. Springer, Berlin, Heidelberg, pp. 33–49.

Brough, Tara, Laura Ciobanu, and Murray Elder (2014). "Permutations of context-free and indexed languages". In: *CoRR* abs/1412.5512.

Capelletti, Matteo (2007). *Parsing with Structure-preserving Categorial Grammars*. LOT international series. LOT. ISBN: 9789078328339.

Chaudhuri, Kaustuv, Frank Pfenning, and Greg Price (2008). "A Logical Characterization of Forward and Backward Chaining in the Inverse Method". In: *Journal of Automated Reasoning* 40.2, pp. 133–177. ISSN: 1573-0670.

Chomsky, Noam (1956). "Three models for the description of language". In: *IRE Transactions on Information Theory* 2.3, pp. 113–124.

— (1957). *Syntactic Structures*. The Hague: Mouton and Co.

— (1959). "On certain formal properties of grammars". In: *Information and Control* 2.2, pp. 137 –167.

Clark, Stephen (2015). "Vector Space Models of Lexical Meaning". In: *The Handbook of Contemporary Semantic Theory*. John Wiley and Sons, Ltd, pp. 493–522. ISBN: 9781118882139.

Dost, Sjoerd N. (2013). "Typelogical Proof Nets in Python-Graphical Lambek-Grishin Calculus". B.Sc. thesis.

Girard, Jean-Yves (1987). "Linear Logic". In: *Theoretical Computer Science* 50.1, pp. 1–102. ISSN: 0304-3975.

Goré, Rajeev (1998). "Substructural logics on display". In: *Logic Journal of IGPL* 6.3, pp. 451–504.

Hendriks, H. (1993). *Studied Flexibility: Categories and Types in Syntax and Semantics*. ILLC dissertation series. Institute for Logic, Language and Computation, Universiteit van Amsterdam. ISBN: 9789074795012.

Hopcroft, John E. and Jeff D. Ullman (1979). *Introduction to Automata Theory, Languages, and Computation*. 1st. Addison-Wesley Publishing Company.

Huybregts, Rini (1984). "The weak inadequacy of context-free phrase structure grammars". In: *Van Periferie naar Kern*. Ed. by G. de Haan, M. Trommelen, and W. Zonneveld.

Joshi, Aravind K. and Seth Kulick (1997). "Partial Proof Trees as Building Blocks for a Categorial Grammar". In: *Linguistics and Philosophy* 20.6, pp. 637–667. ISSN: 1573-0549.

Joshi, Aravind K., Seth Kulick, and Natasha Kurtonina (1998). "An LTAG Perspective on Categorial Inference". In: *LACL*. Vol. 2014. Lecture Notes in Computer Science. Springer, pp. 90–105.

Joshi, Aravind K. and Yves Schabes (1997). "Handbook of Formal Languages, Vol. 3". In: ed. by Grzegorz Rozenberg and Arto Salomaa. New York, NY, USA: Springer-Verlag New York, Inc. Chap. Tree-adjoining Grammars, pp. 69–123. ISBN: 3-540-60649-1.

Kallmeyer, Laura (2010). *Parsing Beyond Context-Free Grammars*. 1st. Springer Publishing Company, Incorporated.

Kandulski, Maciej (1988). "Phrase Structure Languages Generated by Categorial Grammars With Product". In: *Mathematical Logic Quarterly* 34.4, pp. 373–383. ISSN: 1521-3870.

Kanovich, Max I. (1994). "The Complexity of Horn Fragments of Linear Logic". In: *Ann. Pure Appl. Logic* 69.2-3, pp. 195–241.

Kurtonina, Natasha and Michael Moortgat (1997). "Specifying Syntactic Structures". In: ed. by Patrick Blackburn and Maarten de Rijke. Stanford, CA, USA: Center for the Study of Language and Information. Chap. Structural Control, pp. 75–113. ISBN: 1-57586-085-6.

Lambek, Joachim (1958). "The Mathematics of Sentence Structure". In: *Americal Mathematical Monthly* 65, pp. 154–170.

Lambek, Joachim (1961). "On the Calculus of Syntactic Types". In: *Structure of Language and its Mathematical Aspects*. Ed. by R. Jacobsen. Proceedings of Symposia in Applied Mathematics, XII. American Mathematical Society.

Melissen, Matthijs (2009). "The Generative Capacity of the Lambek-Grishin Calculus: A New Lower Bound". In: *FG*. Vol. 5591. Lecture Notes in Computer Science. Springer, pp. 118–132.

Montague, Richard (1970). "Pragmatics and Intensional Logic". In: *Synthese* 22.1/2, pp. 68–94. ISSN: 00397857, 15730964.

Moortgat, Michael (2009). "Symmetric Categorial Grammar". In: *J. Philosophical Logic* 38.6, pp. 681–710.

— (2014). "Typelogical Grammar". In: *The Stanford Encyclopedia of Philosophy*. Ed. by Edward N. Zalta. Spring 2014. Metaphysics Research Lab, Stanford University.

Moortgat, Michael and Richard Moot (2011). "Proof nets for the Lambek-Grishin calculus". In: *CoRR* abs/1112.6384.

Moortgat, Michael and Richard Oehrle (1999). "Proof nets for the grammatical base logic". In: *Dynamic Perspectives in Logic and Linguistics*. Ed. by V.M. Abrusci, C. Casadio, and G. Sandri.

Moot, Richard (2002). "Proof Nets for Linguistic Analysis". PhD Thesis. Utrecht University.

— (2014). "Extended Lambek Calculi and First-Order Linear Logic". In: *Categories and Types in Logic, Language, and Physics: Essays Dedicated to Jim Lambek on the*

Occasion of His 90th Birthday. Ed. by Claudia Casadio et al. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 297–330. ISBN: 978-3-642-54789-8.

Moot, Richard (2015). "Comparing and evaluating extended Lambek calculi". In: *CoRR* abs/1506.05561.

Moot, Richard and Christian Retoré (2012). *The Logic of Categorial Grammars: A Deductive Account of Natural Language Syntax and Semantics*. FoLLI-LNCS. Springer, p. 322.

O'Keefe, Richard A. (1990). *The Craft of Prolog*. Cambridge, MA, USA: MIT Press. ISBN: 0-262-15039-5.

Pentus, Mati (1997). "Product-Free Lambek Calculus and Context-Free Grammars". In: *Journal of Symbolic Logic* 62.2, pp. 648–660.

— (2006). "Lambek calculus is NP-complete". In: *Theor. Comput. Sci.* 357.1-3, pp. 186–201.

Pereira, Fernando CN and Stuart M Shieber (2002). *Prolog and natural-language analysis*. Microtome Publishing.

Shieber, Stuart M. (1985). "Evidence against the context-freeness of natural language". In: *Linguistics and Philosophy* 8.3, pp. 333–343.

Shieber, Stuart M, Yves Schabes, and Fernando CN Pereira (1995). "Principles and implementation of deductive parsing". In: *The Journal of logic programming* 24.1, pp. 3–36.

Sikkel, N. (1993). "Parsing Schemata". PhD Thesis. University of Twente, p. 412. ISBN: 90-9006688-8.

Steedman, Mark (2000). *The Syntactic Process*. Cambridge, MA, USA: MIT Press. ISBN: 0-262-19420-1.

van Benthem, Johan (1995). *Language in Action: Categories, Lambdas and Dynamic Logic*. North-Holland. ISBN: 9780262720243.

Whitman, Neil (2004). *Category neutrality: A type-logical investigation*. Routledge.

Wijnholds, Gijs (2014). "Categorical foundations for extended compositional distributional models of meaning". M.Sc. thesis. Universiteit van Amsterdam.

Appendix A

Prolog parser

Parsing

```
% A bottom-up parser for the focused Lambek-Grishin Calculus (fLG)
% Bachelor thesis by Jaap Jumelet (2017)
% Under supervision of prof. dr. Michael Moortgat

:- dynamic at_label/1,constraints/1.
:- style_check(-discontiguous).
:- [unification],[unfolding],[latex],[lexicon].


:-  retractall(at_label(_)),
    assert(at_label(0)),
    retractall(constraints(_)),
    assert(constraints(false)).

reset_labels :-
    retractall(at_label(_)),assert(at_label(0)).

no_constraints :-
    retractall(constraints(_)),
    assert(constraints(false)).
constraints :-
    retractall(constraints(_)),
    assert(constraints(true)).

/*=====================================================
                    Bottom-up Parsing
=====================================================*/

% example input:
% ?- sen(S), parse(0,S,s). "everyone thinks some teacher likes bob"
% ?- parse(0,dia(box(dia(box(a)))),dia(box(a))). <- 2 derivations

%%% parse(Focus, In, Out)
%%% =====================
%%%
%%%     Focus  ==> 1,0, r (left, neutral, right)
%%%     In, Out ==> 2 lists of lexical items or formulas, or a single structure/formula
%%%

parse(F, In, Out) :-
    reset_labels,

```

```
    % Create all unfolded formulas
    get_in_out(In, Out, In2, Out2, Sentence, IO-Goal),
    get_unfolds(in, 0, In2, InUnfolds, InLinks-[]),

    (IO = in ->
        make_goal(IO-Goal, GoalUnfold, OpenHypGoals, ClosedHypGoals, GoalLinks-InLinks),
        get_unfolds(out, 0, Out2, OutUnfolds, Links-GoalLinks)
    ;
        get_unfolds(out, 0, Out2, OutUnfolds, OutLinks-InLinks),
        make_goal(IO-Goal, GoalUnfold, OpenHypGoals, ClosedHypGoals, Links-OutLinks)),

    get_labeled_forms(in  , InUnfolds  , InLabeledForms  ),
    get_labeled_forms(out , OutUnfolds , OutLabeledForms ),
    get_labeled_form(IO, GoalUnfold-_, LabeledGoal),

    conc(InUnfolds,OutUnfolds,AllUnfolds-[]),
    open_closed(AllUnfolds, OpenUnfolds-OpenHypGoals, ClosedUnfolds-ClosedHypGoals),

    !, % Unfolding phase is deterministic
    time(
    % Unify all open gaps by linking them
    findall(term(FinalProof,FinalLinks,UnifiedIn,UnifiedOut,SentenceTree),
    ((
    unify_all_open(0, ClosedUnfolds,
                    OpenUnfolds,
                    GoalUnfold,
                    Links,
                    vdash(0, FinalIn, FinalOut)-CompleteProof-FinalLinks),

    % Fill in the gaps that are left by atomic formulas, word order is preserved
    unify_in(FinalIn, InLabeledForms, LabeledGoal, Sentence, UnifiedIn, SentenceTree,
    []-[]),
    unify_out(FinalOut, OutLabeledForms, LabeledGoal, UnifiedOut, []),
    add_init_focus(F, FinalIn, FinalOut, CompleteProof, FinalProof))),[P|Proofs])),!,

    print_latex([P|Proofs]),tex_display.



% get_in_out(InitIn, InitOut, In, Out, Sentence, IOGoal-Goal)
% ==========
%
% For efficiency reasons, the final goal is separated from the other unfolds.
% IOGoal is used for the unfolding of the goal formula.

get_in_out([In], Out , In2 , Out2 , Sen , Goal    ) :-
    !,get_in_out(In, Out, In2, Out2, Sen, Goal).
get_in_out(In ,[Out], In2 , Out2 , Sen , Goal    ) :-
    !,get_in_out(In, Out, In2, Out2, Sen, Goal).
get_in_out(In , Out , In2 , Out2 , In  , _-[]    ) :- is_list(In), is_list(Out),!,
    get_forms(In, In2),
    get_forms(Out, Out2).
get_in_out(In , Out , In2 , []   , In  , out-Out ) :- is_list(In) , \+is_list(Out),!,
    get_forms(In, In2).
get_in_out(In , Out , []  , Out2 , [In], in-In   ) :- is_list(Out),!,
    get_forms(Out, Out2).
```

```
get_in_out(In , Out , []  ,[Out2], [In], in-In2  ) :-
    get_form(In, In2)  ,
    (pol(In2,+) ;atomic(In2),atom_pol(In2/_ ,+)),
    get_form(Out, Out2).
get_in_out(In , Out , [In2], []  , [In], out-Out2) :- get_form(Out, Out2), (pol(Out2,-);
   atomic(Out2),atom_pol(Out2/_,-)), get_form(In, In2).


% get_unfolds(InOut, Focus, Input, Unfolds, AxiomLinks)
% ===========
%
% unfolds a list of formulas, and returns a difference list of unfolds and axiom links.

get_unfolds(_, _, [], U-U, L-L).
get_unfolds(IO, I, [Form], [Unfold|U]-U, Links) :-
    unfold(IO, I-n, Form, Unfold, Links).
get_unfolds(IO, I, [Form|Forms], [Unfold|Unfolds]-U, LinksA-LinksC ) :-
    J is I+1,
    unfold(IO, I-J, Form, Unfold, LinksA-LinksB),
    get_unfolds(IO, J, Forms, Unfolds-U, LinksB-LinksC).


% get_forms(Input, Formula)
% =========
%
% Returns a formula if the input is in the lexicon, otherwise it returns the input itself
   .

get_forms(In, Forms) :-
    maplist(get_form, In, Forms).
get_form(Word, Formula) :-
    lex(Word, Formula).
get_form(Formula, Formula) :-
    \+lex(Formula, _).


% get_labeled_forms(IO, Unfolds, LabeledForms)
% =================
%
% Returns a list of labeled formulas, given a list of unfolded formulas.

get_labeled_forms(IO, [F|Fs]-_, [L|Ls]) :-
    \+var(F),
    get_labeled_form(IO, F, L),
    get_labeled_forms(IO, Fs-_, Ls).
get_labeled_forms(_, U-U, []).


get_labeled_form(in, vdash(_, F, _)-_-_-_, F).
get_labeled_form(out, vdash(_, _, F)-_-_-_, F).
get_labeled_form(_, []-_ , [] ).
get_labeled_form(_, I-(F/M)-J, I-(F/M)-J).


% open_closed(Unfolds, OpenUnfolds, ClosedUnfolds)
% ===========
%
```

```
154 % Returns 2 difference lists of open & closed unfolds.
    % Closed formulas have an empty set of holes and no hypotheses.
156 % Atomic formulas are discarded at this point.

158 open_closed([], O-O, C-C).
    open_closed([A-B-[]-[]|Unfolds], Open, [A-B|ClosedRest]-C) :-
160     open_closed(Unfolds, Open, ClosedRest-C).

162 open_closed([_-(_A/_M)-_|Unfolds], Open, Closed) :-
        open_closed(Unfolds, Open, Closed).
164
    open_closed([U-Hyps|Unfolds], [U|Open]-O, Closed-C) :-
166     open_closed(Hyps, OpenHyps-O, ClosedHyps-C),
        open_closed(Unfolds, Open-OpenHyps, Closed-ClosedHyps).
168


170 % make_goal(IO-Goal, GoalUnfold, GoalOpen, GoalClosed, GoalLinks)
    % =========
172 %
    % Creates the unfold of the goal formula.
174 % Unfolding depends on whether the goal is compound or atomic.

176 make_goal(_-[], [], [], [], L-L).
    make_goal(IO-Goal, GoalUnfold, GoalOpen, GoalClosed, GoalLinks) :-
178     compound(Goal),
        make_compound_goal(IO, Goal, GoalUnfold, GoalOpen, GoalClosed, GoalLinks).
180 make_goal(IO-Goal, GoalUnfold, [], [], L-L) :-
        atomic(Goal),
182     make_atomic_goal(IO, Goal, GoalUnfold).

184 make_compound_goal(IO, Goal, GoalUnfold, OpenGoals, ClosedGoals, GoalLinks) :-
        unfold(IO, 0-n, Goal, GoalUnfold-Hyps, GoalLinks),
186     open_closed(Hyps, OpenGoals-[], ClosedGoals-[]).

188 make_atomic_goal(out, Goal, vdash(0, 0-X-n, 0-Goal/M-n)-H-[vdash(0, 0-X-n, 0-Goal/M-n)-H
        -[]] ) :-
        get_label(M).
190 make_atomic_goal(in,  Goal, vdash(0, 0-Goal/M-n, 0-Y-n)-H-[vdash(0, 0-Goal/M-n, 0-Y-n)-H
        -[]] ) :-
        get_label(M).
192


194 % add_init_focus(Foc, In, Out, Proof, FinalProof)
    % ==============
196 %
    % Adds the initial focus of the sequent at the bottom of the proof.
198 % An initial focus for a proof that is brought into that focus already,
    % means we can remove that step in the derivation.
200
    add_init_focus(0, _In, _Out, Proof, Proof).
202 add_init_focus(l, _, _, fl1(Proof),Proof).
    add_init_focus(r, _, _, fr1(Proof),Proof).
204
    add_init_focus(l, In, _Out, Proof, fl(Proof)) :- pol(In,+).
206 add_init_focus(r, _In, Out, Proof, fr(Proof)) :- pol(Out,-).


208
```

```
% unify_in(InStruct, InForms, Goal, SenList, FinalInStruct, SenStruct, Rest)
% ========
%
% unify_in/7 unifies the atomic formulas of the input (InForms) with the
% structure that is derived during the parsing process. It also builds up
% the sentence structure: e.g. alice*(likes*bob).

% If the input formula is the final goal, it is separated from the input list at
% the beginning of the parsing process.
unify_in(I-Goal-J, [], I-Goal-J, [W], Goal2, W, []-[]) :-
    remove_indices(Goal,Goal2).

unify_in(I-In-J, [I-In-J|InRest], _, [W|SenRest], In2, W, InRest-SenRest) :-
    remove_indices(In,In2).
unify_in(_-otimes0(X, Y)-_,I, _, S, otimes0(IX, IY), SX*SY, InRest2-SenRest2) :-
    unify_in(X, I, _, S, IX, SX, InRest-SenRest),
    unify_in(Y, InRest, _, SenRest, IY, SY, InRest2-SenRest2).

% unify_out(OutStruct, OutForms, Goal, FinalOutStruct, Rest)
% =========
%
% Similar to unify_in/7, but for out-structures (on the rhs).

unify_out(I-Goal-J, [], I-Goal-J, Goal2, []) :-
    remove_indices(Goal,Goal2).
unify_out(I-Out-J, [I-Out-J|OutRest], _, Out2, OutRest) :-
    remove_indices(Out,Out2).
unify_out(_-oplus0(X, Y)-_,O, _, oplus0(OX, OY), OutRest2) :-
    unify_out(X, O, _, OX, OutRest),
    unify_out(Y, OutRest, _, OY, OutRest2).


remove_indices(At/N, At/N).
remove_indices(_-F-_, F2) :-
    remove_indices(F,F2).

% This could be simplified by =.., but that is heavier on computation.
remove_indices(otimes(A,B), otimes(A2,B2)) :-
    remove_indices(A, A2),
    remove_indices(B, B2).
remove_indices(over(A,B), over(A2,B2)) :-
    remove_indices(A, A2),
    remove_indices(B, B2).
remove_indices(under(A,B), under(A2,B2)) :-
    remove_indices(A, A2),
    remove_indices(B, B2).

remove_indices(oplus(A,B), oplus(A2,B2)) :-
    remove_indices(A, A2),
    remove_indices(B, B2).
remove_indices(oslash(A,B), oslash(A2,B2)) :-
    remove_indices(A, A2),
    remove_indices(B, B2).
remove_indices(obslash(A,B), obslash(A2,B2)) :-
    remove_indices(A, A2),
    remove_indices(B, B2).
```

```
remove_indices(box(A), box(A2)) :-
    remove_indices(A, A2).
remove_indices(dia(A), dia(A2)) :-
    remove_indices(A, A2).


% aux preds

lift(F, under(over(s, F),s)).
form(1, under(over(s, under(np, s)),s)).
form(2, over(under(np, s),np)).
form(3, over(under(n, n),over(s, dia(box(np))))).
sen([everyone, thinks, some, teacher, likes, bob]).

conc(A-B,B-C,A-C).

printlist([]).
printlist([H|T]) :-
        print(H),nl, nl, printlist(T).

% polarity: negative: mono left ; positive: mono right

% Polarity for indexed formulas
pol(_-X-_,P) :- pol(X,P).

pol(box(_),-).
pol(oplus(_, _),-).
pol(over(_, _),-).
pol(under(_, _),-).

pol(dia(_),+).
pol(otimes(_, _),+).
pol(oslash(_, _),+).
pol(obslash(_, _),+).

pol(A/_, Pol) :- atom_pol(A/_, Pol).

% atom_pol/2: choose bias for atoms (adapt to your own liking)

atom_pol(A/_, Pol) :- A=s -> Pol=(-) ; Pol=(+).

% Prints the latex output of unfolding In.
% Note: does not work for a hole with more than one hypothesis.
latex_unfold(IO,In) :-
  %  reset_labels,
    (lex(In,Form);Form = In),!,
    unfold(IO, 3-4, Form, U, _),
    print_unfold(U),
    tex_display,!.
```

Unfolding

```
/*=========================================================
                        Unfolding
=========================================================*/

:- style_check(-discontiguous).

%%% unfold(InOut, Formula, VDash-Proof-Holes-Hyps, Links)
%%% ============================================================
%%%
%%%     InOut   ==> in, out: determines which side of the sequent is unfolded
%%%     Formula ==> The formula that is unfolded
%%%
%%%     VDash   <== The bottom sequent with labeled atoms
%%%     Proof   <== The proof of the unfolded formula
%%%     Holes   <== List of holes in the proof.
%%%                 Holes are of the form of: VDash-Hole-Hyp,
%%%                 where Hyp is a list of hypothesis constraints.
%%%     Hyps    <== List of unfolded hypotheses
%%%     Links   <== A difference list of axiom links

unfold(_, I-J, Atom, I-Atom/M-J, L-L ) :-
    atomic(Atom),!,
    get_label(M).

unfold(in, I-J, Formula, vdash(0, LabeledFormula, Structure)-UnfoldedProof-Holes-
    HypUnfolds, Links) :-
    compound(Formula),
    init_struct(in, I-Formula-J, LabeledFormula, Structure),!,
    g(in, vdash(0, LabeledFormula, Structure),UnfoldedProof, Links, Holes-[], HypUnfolds
    -[]).

unfold(out, I-J, Formula, vdash(0, Structure, LabeledFormula)-UnfoldedProof-Holes-
    HypUnfolds, Links) :-
    compound(Formula),
    init_struct(out, I-Formula-J, LabeledFormula, Structure),!,
    g(out, vdash(0, Structure, LabeledFormula),UnfoldedProof, Links, Holes-[], HypUnfolds
    -[]).

% Labeling of atoms uses a dynamic predicate at_label/1.
get_label(M) :-
    at_label(M),
    N is M+1,
    retractall(at_label(_)),
    assert(at_label(N)).

is_struct(otimes0(_,_)).
is_struct(over0(_,_)).
is_struct(under0(_,_)).
is_struct(oplus0(_,_)).
is_struct(oslash0(_,_)).
is_struct(obslash0(_,_)).

% init_struct(InOut , Symbol , Formula , LabeledFormula , Structure)
% ===========
%
```

```
% Creates an input or output structure.
% The symbol is used later in the computation during the creation of the constraints.

% When a hypothesis is unfolded, its atoms are already labeled
init_struct(_, I-(I-At/M-J)-J, I-At/M-J, I-_VAR-J).

init_struct(IO, I-(I-Form-J)-J, LabeledForm, Struct) :-
    init_struct(IO, I-Form-J, LabeledForm, Struct).

init_struct(IO, I-Form-J, I-LabeledForm-J, I-Struct-J) :-
    get_bi_struct(IO, IOA-IOB, (I-Form-J)-A-B, LabeledForm-L_A-L_B, Struct-X-Y),
    init_struct(IOA, A, L_A, X),
    init_struct(IOB, B, L_B, Y).

init_struct(IO, I-Form-J, I-LabeledForm-J, I-Struct-J) :-
    get_un_struct(IO, (I-Form-J)-A, LabeledForm-L_A, Struct-X),
    init_struct(IO, A, L_A, X).

% Atom labeling
init_struct(_, I-At-J, I-At/M-J, I-_VAR-J) :-
    atomic(At),
    get_label(M).

% Structures matching an in-formula
get_bi_struct(in, in-in , (_-otimes(A, B)-_)-(_-A-_)-(_-B-_) , otimes(L_A, L_B)-L_A-L_B ,
    _VAR-_-_          ).
get_bi_struct(in, in-out, (I-over(A, B)-J)-(I-A-K)-(J-B-K)  , over(L_A, L_B)-L_A-L_B   ,
   over0(X, Y)-X-Y  ).
get_bi_struct(in, out-in, (J-under(A, B)-K)-(I-A-J)-(I-B-K) , under(L_A, L_B)-L_A-L_B  ,
   under0(X, Y)-X-Y ).

get_bi_struct(in, in-in , (I-oplus(A, B)-K)-(I-A-J)-(J-B-K)   , oplus(L_A, L_B)-L_A-L_B
    , oplus0(X, Y)-X-Y ).
get_bi_struct(in, in-out, (_-oslash(A, B)-_)-(_-A-_)-(_-B-_)  , oslash(L_A, L_B)-L_A-L_B
    , _VAR-_-_).
get_bi_struct(in, out-in, (_-obslash(A, B)-_)-(_-A-_)-(_-B-_) , obslash(L_A, L_B)-L_A-L_B
    , _VAR-_-_).

get_un_struct(in, (_-dia(A)-_)-(_-A-_) , dia(L_A)-L_A , _VAR-_    ).
get_un_struct(in, (I-box(A)-J)-(I-A-J) , box(L_A)-L_A , box0(X)-X ).

% Structures matching an out-formula
get_bi_struct(out, out-out, (I-otimes(A, B)-K)-(I-A-J)-(J-B-K) , otimes(L_A, L_B)-L_A-L_B
    , otimes0(X, Y)-X-Y ).
get_bi_struct(out, out-in , (_-over(A, B)-_)-(_-A-_)-(_-B-_)   , over(L_A, L_B)-L_A-L_B
    , _VAR-_-_).
get_bi_struct(out, in-out , (_-under(A, B)-_)-(_-A-_)-(_-B-_)  , under(L_A, L_B)-L_A-L_B
    , _VAR-_-_).

get_bi_struct(out, out-out, (_-oplus(A, B)-_)-(_-A-_)-(_-B-_)   , oplus(L_A, L_B)-L_A-L_B
      , _VAR-_-_ ).
get_bi_struct(out, out-in , (I-oslash(A, B)-J)-(I-A-K)-(J-B-K)  , oslash(L_A, L_B)-L_A-
   L_B  , oslash0(X, Y)-X-Y  ).
get_bi_struct(out, in-out , (J-obslash(A, B)-K)-(I-A-J)-(I-B-K) , obslash(L_A, L_B)-L_A-
   L_B , obslash0(X, Y)-X-Y ).

get_un_struct(out, (I-dia(A)-J)-(I-A-J) , dia(L_A)-L_A , dia0(X)-X ).
```

```
get_un_struct(out, (_-box(A)-_)-(_-A-_) , box(L_A)-L_A , _VAR-_    ).


%%% g(InOut, Sequent, Proof, Symbol-Links, Holes, HypUnfolds)
%%% ========================================================
%%%
%%% g is (originally) based on Herman Hendriks (1993) unfolding procedure.
%%% It creates a complete unfold for a given sequent, i.e. a proof,
%%% a dlist of axiom list, a dlist of open holes and a list of unfolded hypotheses.

    %%% MONOTONICITY RULES %%%
g(InOut, vdash(F, Ant, Suc), Proof, LinksA-LinksC, HolesA-HolesC, HypUnfoldsA-HypUnfoldsC
    ) :-
    \+var(Ant),\+var(Suc),

    g_LGf(InOut, vdash(F, Ant, Suc), InOutA-A, InOutB-B, Proof, ProofA-ProofB ),

    g(InOutA, A, ProofA, (LinksA-LinksB), HolesA-HolesB, HypUnfoldsA-HypUnfoldsB ),
    g(InOutB, B, ProofB, (LinksB-LinksC), HolesB-HolesC, HypUnfoldsB-HypUnfoldsC ).

g_LGf(in, vdash(l, I-oplus(A, B)-K, I-oplus0(X, Y)-K) , in-vdash(l, A, X)  , in-vdash(l,
   B, Y)  , oplus(ProofA, ProofB), ProofA-ProofB).
g_LGf(in, vdash(l, I-over(B, A)-J , I-over0(Y, X)-J ) , in-vdash(l, B, Y)  , out-vdash(r,
    X, A) , over(ProofA, ProofB) , ProofA-ProofB).
g_LGf(in, vdash(l, J-under(A, B)-K, J-under0(X, Y)-K) , out-vdash(r, X, A) , in-vdash(l,
   B, Y)  , under(ProofA, ProofB), ProofA-ProofB).

g_LGf(out, vdash(r, I-otimes0(X, Y)-K , I-otimes(A, B)-K ) , out-vdash(r, X, A) , out-
   vdash(r, Y, B) , otimes(ProofA, ProofB)  , ProofA-ProofB).
g_LGf(out, vdash(r, I-oslash0(X, Y)-J , I-oslash(A, B)-J ) , out-vdash(r, X, A) , in-
   vdash(l, B, Y)  , oslash(ProofA, ProofB)  , ProofA-ProofB).
g_LGf(out, vdash(r, J-obslash0(Y, X)-K, J-obslash(B, A)-K) , in-vdash(l, B, Y)  , out-
   vdash(r, X, A) , obslash(ProofA, ProofB) , ProofA-ProofB).

% box L, dia R
g(in, vdash(l, I-A-J, I-Y-J), box(Proof),Links, Holes, HypUnfolds) :-
    \+var(A)    , \+var(Y)     ,
    A = box(A2) , Y = box0(Y2) ,
    g(in, vdash(l, A2, Y2),Proof, Links, Holes, HypUnfolds).

g(out, vdash(r, I-X-J, I-A-J), dia(Proof), Links, Holes, HypUnfolds) :-
    \+var(X)     , \+var(A)    ,
    X = dia0(X2) , A = dia(A2) ,
    g(out, vdash(r, X2, A2),Proof, Links, Holes, HypUnfolds).

    %%% (DE)FOCUSING %%%
g(Pol, vdash(0, A, Y), fl1(Proof), Links, Holes, HypUnfolds) :-
    not_var(A), pol(A,-),
    g(Pol, vdash(l, A, Y),Proof, Links, Holes, HypUnfolds).
g(Pol, vdash(r, X, A),fr(Proof), Links, Holes, HypUnfolds) :-
    not_var(A), pol(A,-),
    g(Pol, vdash(0, X, A),Proof, Links, Holes, HypUnfolds).

g(Pol, vdash(0, X, A),fr1(Proof),Links, Holes, HypUnfolds) :-
    not_var(A), pol(A,+),
    g(Pol, vdash(r, X, A),Proof, Links, Holes, HypUnfolds).
g(Pol, vdash(l, A, Y),fl(Proof), Links, Holes, HypUnfolds) :-
```

```
147     not_var(A), pol(A,+),
        g(Pol, vdash(0, A, Y),Proof, Links, Holes, HypUnfolds).
149
151     %%% (CO)AXIOMS %%%
    % Ax - N is unbound
153 g(in, vdash(l, I-(At/M)-J, I-(At/N)-J), idl(At, M, N), [M/N|L]-L, H-H, Hyps-Hyps) :-
        \+var(M),
155     atom_pol(At/M,-).

157 % CoAx - M is unbound
    g(out, vdash(r, I-(At/M)-J, I-(At/N)-J), idr(At, M, N), [M/N|L]-L, H-H, Hyps-Hyps) :-
159     \+var(N),
        atom_pol(At/N,+).
161

163     %%% OPEN SUBPROOFS / HYPOTHESES %%%
    % Negative structure on an out-position (RHS of the sequent)
165 g(out, vdash(0, X, Y), ProofHole, Links, Hole, HypUnfolds) :-
        is_var(X),
167     pol(Y,-),
        findall_unfolds(out, Y, StructY, Links, HypUnfolds, HypConsts-[]),
169     make_hyp_hole(vdash(0, X, StructY), ProofHole, HypConsts, Hole).

171 % Positive formula on an in-position (LHS of the sequent)
    g(in, vdash(0, X, Y), ProofHole, Links, Hole, HypUnfolds) :-
173     is_var(Y),
        pol(X,+),
175     findall_unfolds(in, X, StructX, Links, HypUnfolds, HypConsts-[]),
        make_hyp_hole(vdash(0, StructX, Y), ProofHole, HypConsts, Hole).
177

179 % make_hyp_hole(Sequent, ProofHole, Consts, Hole)
    % =============
181 %
    % Adds the hypotheses constraints to the complete hole.
183
    make_hyp_hole(Struct, ProofHole, HypConsts, [Struct-ProofHole-HypConsts|H]-H).
185

187 % findall_unfolds(InOut, Sym, Sequent, Structure, Links, Unfolds, Consts)
    % ===============
189 %
    % Finds and unfolds all formulas that are situated on a position (in/out) on which
191 % they can be unfolded. Also returns an updated list of axiom links and a list of
    % hypotheses constraints that are added to the hole of the initial unfold.
193
    findall_unfolds(in, I-dia(A)-J, I-dia0(X)-J, Links, Unfolds, Consts ) :-
195     findall_unfolds(in , A, X, Links, Unfolds, Consts ).
    findall_unfolds(out, I-box(A)-J, I-box0(X)-J, Links, Unfolds, Consts ) :-
197     findall_unfolds(out, A, X, Links, Unfolds, Consts ).
    findall_unfolds(IO, I-Form-J, I-Struct-J, LinksA-LinksC, UnfoldsA-UnfoldsC, ConstsA-
       ConstsC ) :-
199     \+var(Form),
        get_arg(IO, Form-A-B, Struct-X-Y, IOA, IOB),
201
        findall_unfolds(IOA , A, X, LinksA-LinksB, UnfoldsA-UnfoldsB , ConstsA-ConstsB ),
```

```
203     findall_unfolds(IOB , B, Y, LinksB-LinksC, UnfoldsB-UnfoldsC , ConstsB-ConstsC ).

205 findall_unfolds(out, I-Form-J, I-Form-J, HypLinks, [Bottom-Proof-Holes-Hyps|U]-U, [Proof|
    P]-P ) :-
     is_compound(Form),pol(Form,+),
207     unfold(out, I-I, Form, Bottom-Proof-Holes-Hyps, HypLinks).

209 findall_unfolds(in, I-Form-J, I-Form-J, HypLinks, [Bottom-Proof-Holes-Hyps|U]-U, [Proof|P
    ]-P) :-
     is_compound(Form),pol(Form,-),
211     unfold(in, I-I, Form, Bottom-Proof-Holes-Hyps, HypLinks).

213 findall_unfolds(_, I-At/M-J, I-At/M-J, L-L, U-U, P-P).


215
get_arg(in, otimes(A, B)-A-B  , otimes0(X, Y)-X-Y  , in  , in  ).
217 get_arg(in, oslash(A, B)-A-B  , oslash0(X, Y)-X-Y  , in  , out ).
get_arg(in, obslash(A, B)-A-B , obslash0(X, Y)-X-Y , out , in  ).
219
get_arg(out, oplus(A, B)-A-B , oplus0(X, Y)-X-Y , out , out ).
221 get_arg(out, under(A, B)-A-B , under0(X, Y)-X-Y , in  , out ).
get_arg(out, over(A, B)-A-B  , over0(X, Y)-X-Y  , out , in  ).
223
% Indexed structure vars.
225 is_var(_-X-_)  :-   var(X).
not_var(_-X-_) :- \+var(X).
227
% Labeled atoms are compound too (e.g. np/0)
229 is_compound(Compound) :-
     compound(Compound),
231     Compound \= _/_.
```

Unification

```
/*=======================================================
                        Unification
=======================================================*/

:- style_check(-discontiguous).

%%% unify_all_open(N, Closed, Open, Goal, FinalBottom-FinalProof-FinalLinks)
%%% ============================================================
%%%
%%%     N      ==> Index to prevent spurious ambiguity, and adds efficiency
%%%     Closed ==> List of closed unfolds
%%%     Open   ==> List of open unfolds
%%%     Goal   ==> The unfolded goal formula. Only added when
%%%                there is just one closed or open unfold left.
%%%
%%%     FinalBottom-FinalProof-FinalLinks
%%%            <== Tuple of the completely unified proof,
%%%                final bottomsequent and list of axiom links.

% Case for atomic axioms like np |- np.
unify_all_open(_, [], [], vdash(0,0-A-n,0-B-n)-FProof-GHoles, Links, vdash(0,0-A-n,0-B-n)
    -FProof-FLinks) :-
     focus_shift(vdash(0,0-A-n,0-B-n)-FProof-GHoles, Links, FLinks).

% The cases when there isn't one unfolded goal, but two lists as input/output.
unify_all_open(_, [CBottom-CProof], [], [], FLinks, vdash(0,0-A-n,0-B-n)-FProof-FLinks)
    :-
     residuate(CBottom-CProof, vdash(0,0-A-n,0-B-n)-FProof).

unify_all_open(_, [], [Bottom-Proof-OHoles], [], Links, vdash(0,0-A-n,0-B-n)-FProof-
    FLinks) :-
     focus_shift(Bottom-Proof-OHoles, Links, FLinks),
     residuate(Bottom-Proof,vdash(0,0-A-n,0-B-n)-FProof).

% Keeps resididuating until a sequent of the form
% otimes0(A,B) |- oplus0(C,D) is obtained.
residuate(vdash(0, 0-otimes0(A,B)-n, 0-oplus0(X,Y)-n)-P, vdash(0, 0-otimes0(A,B)-n, 0-
    oplus0(X,Y)-n)-P).
residuate(vdash(0, A, B)-Proof, vdash(0,0-A-n,0-B-n)-FProof) :-
     rule(vdash(0, A, B),vdash(0, A2, B2),Proof, NewProof),
     residuate(vdash(0, A2, B2)-NewProof, vdash(0,0-A-n,0-B-n)-FProof).

% Final step with one closed unfold left, which is unified with the open Goal.
unify_all_open(N, [CBottom-CProof], [], Goal, FLinks, vdash(0,0-A-n,0-B-n)-FProof-FLinks)
     :-
     open_goal(Goal),
     unify_open(N, _, CBottom, [Goal], CProof, FLinks, vdash(0,0-A-n,0-B-n)-FProof-[]-[]).

% One open unfold left, that is focus shifted.
unify_all_open(N, [], [OBottom-OProof-Holes], Goal, Links, vdash(0,0-A-n,0-B-n)-FProof-
    FLinks) :-
     open_goal(Goal),
     focus_shift(OBottom-OProof-Holes, Links, FLinks),
     unify_open(N, _, OBottom, [Goal], OProof, FLinks, vdash(0,0-A-n,0-B-n)-FProof-[]-[]).

```

```
% An open goal has a non-empty set of holes.
open_goal(_-_-[_|_]).

% The goal formula can contain holes too, and can be focus shifted.
unify_all_open(N, [], [Open], GBottom-GProof-[GH|GHoles], Links, FinalBottom-FinalProof-
    FinalLinks) :-
    focus_shift(GBottom-GProof-[GH|GHoles], Links, FinalLinks),
    unify_open(N, _, GBottom, [Open], GProof, FinalLinks, FinalBottom-FinalProof-[]-[]).

% Closed goal that is unified with an open unfold.
unify_all_open(N, [], [Open], GBottom-GProof-[], FinalLinks, FinalBottom-FinalProof-
    FinalLinks) :-
    unify_open(N, _, GBottom, [Open], GProof, FinalLinks, FinalBottom-FinalProof-[]-[]).

% Recursive case: the closed unfold is unified and the updated list of unfolds is passed
    on.
unify_all_open(N, [ClosedBottom-ClosedProof|ClosedRest], [O1|AllOpen], Goal, Links,
    Unified) :-
    unify_open(N, N1, ClosedBottom, [O1|AllOpen], ClosedProof, Links, NewUnified-
    NewOpenRest),
    new_closed_open(NewUnified, ClosedRest, NewOpenRest, NewClosed, NewOpen),
    unify_all_open(N1, NewClosed, NewOpen, Goal, Links, Unified).

% If there are no closed unfolds left, an open unfold is chosen using the index N.
%  If this unfold can be focus shifted (ergo it becomes a closed unfold),
%  it is then unified with the other open unfolds.
unify_all_open(N, [], [O1, O2|AllOpen], Goal, Links, Unified) :-

    nth0(N, [O1, O2|AllOpen], Bottom-Proof-Holes, OpenRest),

    focus_shift(Bottom-Proof-Holes, Links, NewLinks),

    unify_open(N, N1, Bottom, OpenRest, Proof, NewLinks, (NewBottom-NewProof-NewHoles)-
    NewOpenRest),

    new_closed_open(NewBottom-NewProof-NewHoles, [], NewOpenRest, NewClosed, NewOpen),

    unify_all_open(N1, NewClosed, NewOpen, Goal, NewLinks, Unified).

% Not every open sequent can be focus shifted, and not every open sequent
% that can be shifted, has to be shifted.
unify_all_open(N, [], [O1, O2|AllOpen], Goal, Links, Unified) :-
    N1 is N+1, length([O1, O2|AllOpen], L),N1<L,
    unify_all_open(N1, [], [O1, O2|AllOpen], Goal, Links, Unified).

% New unfolds are added to the end of the list of open unfolds
% if they still contain holes.
new_closed_open(Bottom-Proof-[], Closed, Open, [Bottom-Proof|Closed], Open).
new_closed_open(Bottom-Proof-[H|Holes], Closed, Open, Closed, NewOpen) :-
    append(Open, [Bottom-Proof-[H|Holes]], NewOpen).


% focus_shift(Unfold, OldLinks, NewLinks)
% ===========
%
% Focus shifts all holes of an open unfold, and updates the links.
```

```
focus_shift(_-_-[], L, L).
focus_shift(Bottom-Proof-[vdash(0, _-At/M-_, _-At/N-_)-fr1(idr(At, M, N))-[]|Holes],
    Links, [M/N|NewLinks]) :-
    \+var(M),
    atom_pol(At/M,+),
    focus_shift(Bottom-Proof-Holes, Links, NewLinks).
focus_shift(Bottom-Proof-[vdash(0, _-At/M-_, _-At/N-_)-fl1(idl(At, M, N))-[]|Holes],
    Links, [M/N|NewLinks]) :-
    \+var(N),
    atom_pol(At/M,-),
    focus_shift(Bottom-Proof-Holes, Links, NewLinks).


%%% unify_open(N, N1, Sequent, Open, SubProof, Constraints, NewUnfold-OpenRest)
%%% ============================================================
%%%
%%%     N, N1       ==> Indices used for efficient focus shifting
%%%     Sequent     ==> Sequent that is unified to an open unfold
%%%     Open        ==> List of open unfolds
%%%     SubProof    ==> Proof that corresponds to the Sequent that is unified
%%%     Constraints ==> Difference list of constraints
%%%
%%%     NewUnfold-OpenRest
%%%             <== The unified unfold and the list of open unfolds that are left.

unify_open(N, N1, vdash(F, A, B), Open, SubProof, _Links, VDash-Proof-NewHoles-OpenRest)
    :-
    nth0(I, Open, VDash-Proof-ProofHoles, OpenRest),
    select(vdash(F, A, B)-SubProof-Hyps, ProofHoles, NewHoles),
    check_hyps(Hyps, SubProof),
    update_index(I, N, N1).


update_index(I, N, N) :- I >= N.
update_index(I, N, N1) :- I < N, N1 is N-1.

check_hyps([], _).
check_hyps([Hyp|Hyps], SubProof) :-
    contains_hyp(SubProof, Hyp),
    check_hyps(Hyps, SubProof).

% contains_term/2 in the occurs library is not sufficient:
%  contains_term(test(1),over(A)) is true, as the variable A is simply unified with the
    subterm.
%  A hypothesis must be a concrete subterm of its parent argument, so I wrote my own
    version.
contains_hyp(P, Hyp) :-
    \+var(P),
    P =.. [_|Args],
    contains_hyp2(P, Args, Hyp).

contains_hyp2(X, _Args, Hyp) :- Hyp = X.
contains_hyp2(X, Args, Hyp)  :-
    Hyp \= X,
    contains_hyp3(Args, Hyp).
contains_hyp3([Arg], Hyp) :-
    contains_hyp(Arg, Hyp).
```

```
contains_hyp3([Arg1, Arg2], Hyp) :-
    contains_hyp(Arg1, Hyp) ->
        true
    ;
        contains_hyp(Arg2, Hyp).


% If unification fails, the actual structural rules of LGs are applied.
% Sequents are 'varred', to prevent undesirable unification.
unify_open(N, N1, vdash(0, A, B),Open, Proof, Links, NewUnfold) :-
    make_var(A, A2),
    make_var(B, B2),
    rule(vdash(0, A2, B2),vdash(0, A3, B3),Proof, NewProof),
    unify_open(N, N1, vdash(0, A3, B3),Open, NewProof, Links, NewUnfold).

make_var(I-X-J, I-var(X)-J) :-    var(X).
make_var(I-X-J, I-X-J)      :- \+ var(X).

unvar(I-var(X)-J,I-X-J).
unvar(I-X-J, I-X-J) :- X \= var(_).

% rp1
% variables are 'unvarred' after each rule
rule(vdash(0, X2, I-over0(I-Z-K, Y)-_), vdash(0, I-otimes0(X, Y)-K,I-Z-K), F, beta1(F))
    :-
    F \= beta(_),
    unvar(X2, X).
rule(vdash(0, Y2, _-under0(X, I-Z-K)-K), vdash(0, I-otimes0(X, Y)-K,I-Z-K), F, gamma1(F))
    :-
    F \= gamma(_),
    unvar(Y2, Y).

% drp1
rule(vdash(0, _-obslash0(Y, I-Z-K)-K, X2), vdash(0, I-Z-K, I-oplus0(Y, X)-K), F,
    gammaplus1(F)) :-
    F \= gammaplus(_),
    unvar(X2, X).
rule(vdash(0, I-oslash0(I-Z-K, X)-_, Y2), vdash(0, I-Z-K, I-oplus0(Y, X)-K), F, betaplus1
    (F)) :-
    F \= betaplus(_),
    unvar(Y2, Y).

% Postulates P1 & P2 (for rightward extraction)
rule(vdash(0, I-otimes0(I-A-J, Y)-K, Z2),vdash(0, I-otimes0(I-otimes0(I-A-J, J-B-K)-K,X-
    dia0(C)-X)-K,Z),F, xr1(F)) :-
    \+var(Y),
    Y = J-otimes0(J-B-K, X-dia0(C)-X)-K,
    unvar(Z2, Z).
rule(vdash(0, I-otimes0(Y, J-B-K)-K,Z2),vdash(0, I-otimes0(I-otimes0(A, B)-K,X-dia0(C)-X)
    -K,Z),F, xr2(F)) :-
    \+var(Y),
    Y = I-otimes0(I-A-J, X-dia0(C)-X)-J,
    unvar(Z2, Z).

 % Display postulates for box/diamond.
rule(vdash(0, X2, I-box0(Y)-J),vdash(0, I-dia0(X)-J,Y), F, alpha1(F)) :-
    F \= alpha(_),
```

```
    unvar(X2, X).
rule(vdash(0, I-dia0(X)-J, Y2),vdash(0, X, I-box0(Y)-J),F, alpha(F)) :-
    F \= alpha1(_),
    unvar(Y2, Y).

% rp
rule(vdash(0, I-otimes0(I-X-J, Y)-_, Z2), vdash(0, I-X-J, I-over0(Z, Y)-J), F, beta(F))
   :-
    F \= beta1(_),
    unvar(Z2, Z).
rule(vdash(0, _-otimes0(X, J-Y-K)-K, Z2), vdash(0, J-Y-K, J-under0(X, Z)-K), F, gamma(F))
    :-
    F \= gamma1(_),
    unvar(Z2, Z).

% drp
rule(vdash(0, Z2, I-oplus0(I-Y-J, X)-_),vdash(0, I-oslash0(Z, X)-J,I-Y-J), F, betaplus(F)
   ) :-
    F \= betaplus1(_),
    unvar(Z2, Z).
rule(vdash(0, Z2, _-oplus0(Y, J-X-K)-K),vdash(0, J-obslash0(Y, Z)-K,J-X-K),F, gammaplus(F
   )) :-
    F \= gammaplus1(_),
    unvar(Z2, Z).

% g1, g2, g3, g4
rule( vdash(0, I-otimes0(I-X-J, J-Y-K)-K , I-oplus0(I-Z-L, L-W-K)-K),
      vdash(0, L-obslash0(I-Z-L, I-X-J)-J, L-over0(L-W-K, J-Y-K)-L ), F, gr1(F) ).
rule( vdash(0, I-otimes0(I-X-J, J-Y-K)-K , I-oplus0(I-Z-L, L-W-K)-K),
      vdash(0, L-obslash0(I-Z-L, J-Y-K)-K, J-under0(I-X-J, L-W-K)-K), F, gr2(F) ) :-
    I = J;
    I = L.
rule( vdash(0, I-otimes0(I-X-J, J-Y-K)-K , I-oplus0(I-Z-L, L-W-K)-K),
      vdash(0, J-oslash0(J-Y-K, L-W-K)-L , J-under0(I-X-J, I-Z-L)-L), F, gr3(F) ).
rule( vdash(0, I-otimes0(I-X-J, J-Y-K)-K , I-oplus0(I-Z-L, L-W-K)-K),
      vdash(0, I-oslash0(I-X-J, L-W-K)-L , I-over0(I-Z-L, J-Y-K)-J ), F, gr4(F) ) :-
    J = K;
    L = K.
```

Sample lexicon

```
/*=======================================================
                    Sample lexicon
=======================================================*/
% Semantic terms are ignored, but could be added here.

lex(alice, np).
lex(bob, np).

lex(that, over(under(n, n),over(s, dia(box(np))))).
lex(about, over(under(n, n),np)).
lex(everyone, otimes(over(np, n),n)).
lex(someone, otimes(over(np, n),n)).
lex(some, over(np, n)).
lex(every, over(np, n)).
lex(teacher, n).
lex(student, n).
lex(unicorn, n).
lex(left, under(np, s)).
lex(likes, over(under(np, s),np)).
lex(thinks, over(under(np, s),s)).
lex(someone2, obslash(oslash(s, s),np)).
lex(everyone2, over(s, under(np, s))).
lex(today,under(under(np,s),under(np,s))).
lex(and,over(under(s,s),s)).
lex(beautiful,over(n,n)).
lex(blue,over(n,n)).
lex(sky,n).

lex(seems_to,over(under(np,s),under(np,sinf))).
lex(walk,under(over(under(np,s),under(np,sinf)),under(np,s))).
lex(walk2,under(np,sinf)).
```